



Effective Modern C++

42 SPECIFIC WAYS TO IMPROVE YOUR USE OF C++11 AND C++14

Scott Meyers

1 **Contents**

2	Introduction	4
3	Chapter 1 Deducing Types	11
4	Item 1: Understand template type deduction.	11
5	Item 2: Understand <code>auto</code> type deduction.	21
6	Item 3: Understand <code>decltype</code> .	26
7	Item 4: Know how to view deduced types.	34
8	Chapter 2 <code>auto</code>	42
9	Item 5: Prefer <code>auto</code> to explicit type declarations.	42
10	Item 6: Be aware of the typed initializer idiom.	48
11	Chapter 3 From C++98 to C++11 and C++14	56
12	Item 7: Distinguish <code>()</code> and <code>{}</code> when creating objects.	57
13	Item 8: Prefer <code>nullptr</code> to <code>0</code> and <code>NULL</code> .	67
14	Item 9: Prefer alias declarations to <code>typedefs</code> .	72
15	Item 10: Prefer scoped <code>enums</code> to unsscoped <code>enums</code> .	77
16	Item 11: Prefer deleted functions to private undefined ones.	84
17	Item 12: Declare overriding functions <code>override</code> .	89
18	Item 13: Prefer <code>const_iterators</code> to <code>iterators</code> .	96
19	Item 14: Use <code>constexpr</code> whenever possible.	101
20	Item 15: Make <code>const</code> member functions thread-safe.	107
21	Item 16: Declare functions <code>noexcept</code> whenever possible.	113

1	Item 17: Consider pass by value for cheap-to-move parameters that are always copied.	120
3	Item 18: Consider emplacement instead of insertion.	128
4	Item 19: Understand special member function generation.	136
5	Chapter 4 Smart Pointers	145
6	Item 20: Use <code>std::unique_ptr</code> for exclusive-ownership resource management.	147
8	Item 21: Use <code>std::shared_ptr</code> for shared-ownership resource management.	154
10	Item 22: Use <code>std::weak_ptr</code> for <code>std::shared_ptr</code> -like pointers that can dangle.	164
12	Item 23: Prefer <code>std::make_unique</code> and <code>std::make_shared</code> to direct use of <code>new</code> .	170
14	Item 24: When using the Pimpl Idiom, define special member functions in the implementation file.	179
16	Chapter 5 Rvalue References, Move Semantics, and Perfect Forwarding	189
17	Item 25: Understand <code>std::move</code> and <code>std::forward</code> .	190
18	Item 26: Distinguish universal references from rvalue references.	197
19	Item 27: Use <code>std::move</code> on rvalue references, <code>std::forward</code> on universal references.	202
21	Item 28: Avoid overloading on universal references.	210
22	Item 29: Familiarize yourself with alternatives to overloading on universal references.	217
24	Item 30: Understand reference collapsing.	228

1	Item 31: Assume that move operations are not present, not cheap, and not used.	234
3	Item 32: Familiarize yourself with perfect forwarding failure cases.	238
4	Chapter 6 Lambda Expressions	249
5	Item 33: Avoid default capture modes.	251
6	Item 34: Use init capture to move objects into closures.	258
7	Item 35: Use <code>decltype</code> on <code>auto&&</code> parameters to <code>std::forward</code> them.	264
8	Item 36: Prefer lambdas to <code>std::bind</code> .	267
9	Chapter 7 The Concurrency API	275
10	Item 37: Prefer task-based programming to thread-based.	275
11	Item 38: Specify <code>std::launch::async</code> if asynchronicity is essential.	280
12	Item 39: Make <code>std::threads</code> unjoinable on all paths.	286
13	Item 40: Be aware of varying thread handle destructor behavior.	293
14	Item 41: Consider <code>void</code> futures for one-shot event communication.	299
15	Item 42: Use <code>std::atomic</code> for concurrency, <code>volatile</code> for special memory.	308
17		

1 **Introduction**

2 If you're an experienced C++ programmer and are anything like me, you initially
3 approached C++11 thinking, "Yes, yes, I get it. It's C++, only more so." But as your
4 knowledge of the revised language increased, you were surprised by the scale of
5 the changes. `auto` objects, range-based `for` loops, lambda expressions, and rvalue
6 references change the very face of C++, to say nothing of the new concurrency fea-
7 tures. And then there are the idiomatic changes! `0` and `typedefs` are out, `nullptr`
8 and alias declarations are in. Enums should now be scoped. Smart pointers are
9 now preferable to built-in ones. Moving objects is normally better than copying
10 them. There's a lot to learn about C++11.

11 The adoption of C++14 hardly made things easier.

12 So...a lot to learn. More importantly, a lot to learn about making *effective* use of the
13 new capabilities. If you need information on the basic syntax or semantics of fea-
14 tures in "modern" C++, resources abound, but if you're looking for guidance on
15 how to employ the features to create software that's correct, efficient, maintaina-
16 ble, and portable, the search is more challenging. That's where this book comes in.
17 It's devoted not to describing the features of C++11 and C++14, but rather to their
18 effective application.

19 The information in the book is broken into guidelines called *Items*. Want to under-
20 stand the various forms of type deduction? Or to know when (and when not) to
21 declare objects using `auto`? Are you interested in why `const` member functions
22 should be thread-safe, how to implement the Pimpl Idiom using
23 `std::unique_ptr`, why you should avoid default capture modes in lambda ex-
24 pressions, or the differences between (and proper uses of) `std::atomic` and
25 `volatile`? The answers are all here. Furthermore, they're platform-independent,
26 Standards-conformant answers. This is a book about *portable* C++.

27 Items comprise guidelines, not rules, because guidelines have exceptions. The
28 most important part of each Item is not the advice it offers, but the rationale be-
29 hind the advice. Once you've read that, you'll be in a position to determine whether
30 the circumstances of your project justify disregarding the Item's guidance. The

1 true goal of this book isn't to tell you what to do or what to avoid doing, but to
2 convey a deeper understanding of how things work in C++11 and C++14.

3 **Terminology and Conventions**

4 To make sure we understand one another, it's important to agree on some termino-
5 nology, beginning, ironically, with "C++." There have been four standardized ver-
6 sions of C++, each named after the year in which the corresponding ISO Standard
7 was adopted: *C++98*, *C++03*, *C++11*, and *C++14*. C++98 and C++03 differ only in
8 subtle technical details, so in this book, I refer to both as C++98.

9 When I mention C++98, I mean only that version of the language. Where I refer to
10 C++11, I mean both C++11 and C++14, because C++14 is effectively a superset of
11 C++11. When I write C++14, I mean specifically C++14. And if I simply mention
12 C++, I'm making a broad statement that pertains to all language versions. As a re-
13 sult, I might say that C++ places a premium on efficiency (true for all versions),
14 that C++98 lacks support for concurrency (true for C++98 only), that C++11 sup-
15 ports lambda expressions (true for C++11 and C++14), and that C++14 offers gen-
16 eralized function return type deduction (true for C++14 only).

17 C++11's most pervasive feature is probably move semantics, and the foundation of
18 move semantics is distinguishing expressions that are *rvalues* from those that are
19 *lvalues*. That's because rvalues indicate objects eligible for move operations, while
20 lvalues generally don't. In concept (though not always in practice), rvalues corre-
21 spond to anonymous temporary objects returned from functions, while lvalues
22 correspond to objects you can refer to, either by name or by following a pointer or
23 reference.

24 A useful heuristic to determine whether an expression is an lvalue is to ask if you
25 can take its address. If you can, it typically is. If you can't, it's usually an rvalue. A
26 nice feature of this heuristic is that it helps you remember that the type of an ex-
27 pression is independent of whether the expression is an lvalue or an rvalue. That
28 is, given a type T, you can have both lvalues of type T and rvalues of type T. It's es-
29 pecially important to remember this when dealing with a parameter of rvalue ref-
30 erence type, because the parameter itself is an lvalue:

```
1 class Widget {
2 public:
3     Widget(Widget&& rhs); // rhs is an lvalue, though it has
4     ... // an rvalue reference type
5 };
```

6 Here, it'd be perfectly valid to take `rhs`'s address inside `Widget`'s move constructor, so `rhs` is an lvalue, even though its type is an rvalue reference. (By similar reasoning, all parameters are lvalues.)

9 This code example demonstrates several conventions I typically follow:

- 10 • The class name is `Widget`. I use `Widget` whenever I want to refer to an arbitrary user-defined type. Unless I need to show specific details of the class, I use `Widget` without declaring it.
- 13 • I use the parameter name `rhs`, which stands for “right-hand side.” It’s my preferred parameter name for the *move operations* (i.e., move constructor and move assignment operator) and the *copy operations* (i.e., copy constructor and copy assignment operator), though I also employ it for the right-hand parameter of binary operators:

```
18 Matrix operator+(const Matrix& lhs, const Matrix& rhs);
```

19 It’s no surprise, I hope, that `Lhs` stands for “left-hand side.”

- 20 • I highlight parts of code or parts of comments to draw your attention to them. In the code above, I’ve highlighted the declaration of `rhs` and the part of the comment noting that `rhs` is an lvalue.
- 23 • I use “...” to indicate “other code could go here.” This narrow ellipsis is different from the wide ellipsis (“...”) that’s used in the source code for C++11’s variadic templates. That sounds confusing, but it’s not. For example:

```
26 template<typename... Ts> // these are C++
27 void processVals(const Ts&... params) // source code
28 {
29     ... // ellipses
30         // this means "some
31 } // code goes here"
```

1 The declaration of `processVals` shows that I use `typename` when declaring
2 type parameters in templates, but that's merely a personal preference; the
3 keyword `class` would work just as well. On those few occasions where I show
4 code excerpts from a C++ Standard, I declare type parameters using `class`, be-
5 cause that's what the Standards do.

6 When an object is initialized with another object of the same type, the new object
7 is said to be a *copy* of the initializing object, even if the copy was created via the
8 move constructor. Regrettably, there's no terminology in C++ that distinguishes
9 between an object that's a copy-constructed copy and one that's a move-
10 constructed copy:

```
11 void someFunc(Widget w);           // someFunc's parameter w
12                           // is passed by value
13
14 Widget wid;                   // wid is some Widget
15 someFunc(wid);                // in this call to someFunc,
16                           // w is a copy of wid that's
17                           // created via copy construction
18 someFunc(std::move(wid));      // in this call to SomeFunc,
19                           // w is a copy of wid that's
20                           // created via move construction
```

21 Copies of rvalues are generally move-constructed, while copies of lvalues are typi-
22 cally copy-constructed. An implication is that if you know only that an object is a
23 copy of another object, it's not possible to say how expensive it was to construct
24 the copy. In the code above, for example, there's no way to say how expensive it is
25 to create the parameter `w` without knowing whether rvalues or lvalues are passed
26 to `someFunc`. (You'd also have to know the cost of moving and copying `Widgets`.)

27 In a function call, the expressions passed at the call site are the function's *argu-*
28 *ments*. The arguments are used to initialize the function's *parameters*. In the first
29 call to `someFunc` above, the argument is `wid`. In the second call, the argument is
30 `std::move(wid)`. In both calls, the parameter is `w`. The distinction between argu-
31 ments and parameters is important, because parameters are lvalues, but the ar-
32 guments with which they are initialized may be rvalues or lvalues. This is especi-
33 ally relevant during the process of *perfect forwarding*, whereby an argument passed

1 to a function is passed to a second function such that the original argument's rvalue-
2 ness or lvalue-ness is preserved. (Perfect forwarding is discussed in more detail
3 in Item 32.)

4 Well-designed functions are *exception-safe*, meaning they offer at least the basic
5 exception safely guarantee (i.e., the *basic guarantee*). Such functions assure callers
6 that even if an exception is emitted, program invariants remain intact (i.e., no data
7 structures are corrupted) and no resources are leaked. Functions offering the
8 strong exception safety guarantee (i.e., the *strong guarantee*) assure callers that if
9 an exception is emitted, the state of the program remains as it was prior to the call.
10 As Item 16 explains, C++98 Standard Library functions offering the strong guarantee
11 constrain the applicability of move semantics in the C++11 Standard Library.

12 I use the term *callable entity* to refer to anything that can be called using the syntax
13 of a call to a non-member function, i.e., the syntax "*functionName(arguments)*".
14 Functions, function pointers, and function objects are all callable entities. Function
15 objects created through lambda expressions are known as *closures*. It's seldom
16 necessary to distinguish between lambda expressions and the closures they create,
17 so I often refer to both as *lambdas*.

18 In the same vein, I rarely distinguish between *function templates* (i.e., templates
19 that generate functions) and *template functions* (i.e., the functions generated from
20 function templates). Ditto for *class templates* and *template classes*.

21 Many things in C++ can be both declared and defined. *Declarations* introduce
22 names without giving details, such as where storage is located or how the entity is
23 implemented:

```
24 extern int x;                      // object declaration
25 class Widget;                     // class declaration
26 int func(const Widget& w);        // function declaration
27 enum class Color;                 // scoped enum declaration
28 // (see Item 10)
```

29 *Definitions* provide the storage location or implementation details:

```
30 int x;                           // object definition
```

```
1 class Widget { // class definition
2 ...
3 };

4 int func(const Widget& w) // function definition
5 { return w.size(); }

6 enum class Color // scoped enum definition
7 { Yellow, Red, Blue };

8 A definition also qualifies as a declaration, so unless it's really important that
9 something is a definition, I tend to refer to declarations.

10 New C++ Standards generally preserve the validity of code written under older
11 ones, but occasionally the Standardization Committee deprecates features. That's a
12 warning that the features may be removed from future Standards. You should
13 avoid deprecated features. (The reason for deprecation is usually that newer fea-
14 tures offer the same functionality, but with fewer restrictions or drawbacks.
15 std::auto_ptr is deprecated in C++11, for example, because std::unique_ptr
16 does the same job, only better.)

17 Sometimes the Standard says that the result of an operation is undefined behavior.
18 That means that runtime behavior is unpredictable, and it should go without say-
19 ing that you want to steer clear of such uncertainty. Examples of actions with un-
20 defined behavior include using square brackets ("[ ]") to index beyond the bounds
21 of a std::vector, dereferencing an uninitialized iterator, or engaging in a data
22 race (i.e., having two or more threads, at least one of which is a writer, simultane-
23 ously access the same memory location).

24 In source code comments, I sometimes abbreviate "constructor" as ctor and "de-
25 structor" as dtor.
```

26 Reporting Bugs and Suggesting Improvements

27 I've done my best to fill this book with clear, accurate, useful information, but sure-
28 ly there are ways to make it better. If you find errors of any kind (technical, exposi-
29 tory, grammatical, typographical, etc.), or if you have suggestions for how the book
30 could otherwise be improved, please email me at emc++@aristeia.com. New

1 printings give me the opportunity to revise *Effective Modern C++*, and I can't ad-
2 dress issues I don't know about!

3 To view the list of the issues I do know about, consult the book's errata page,
4 <http://www.aristeia.com/BookErrata/emc++-errata.html>.

1 **Chapter 1 Deducing Types**

2 C++98 had a single set of rules for type deduction: the one for function templates.
3 C++11 modifies those rules a bit and adds two more, one for `auto` and one for
4 `decltype`. C++14 then extends the usage contexts in which `auto` and `decltype`
5 may be employed. The increasingly widespread application of type deduction frees
6 programmers from the tyranny of spelling out types that are obvious or redundant.
7 It makes C++ software more adaptable, because changing a type at one point
8 in the source code automatically propagates through type deduction to other locations.
9 However, it can render code more difficult to reason about, because the
10 types deduced by compilers may not be as apparent as we'd like.

11 Effective programming in modern C++ isn't possible without a solid understanding
12 of how type deduction operates. There are just too many contexts where it takes
13 place: in calls to function templates, in most situations where `auto` appears, in
14 `decltype` expressions, and, as of C++14, where the enigmatic `decltype(auto)`
15 construct is employed.

16 This chapter provides the foundational information about type deduction that every C++ developer requires. It explains how template type deduction works, how
17 `auto` builds on that, and how `decltype` goes its own way. It even explains how
18 you can force compilers to make the results of their type deductions visible, thus
19 enabling you to ensure that compilers are deducing the types you want them to.

21 **Item 1: Understand template type deduction.**

22 It's said that imitation is the sincerest form of flattery, but blissful ignorance can be
23 an equally heartfelt accolade. When users of a complex system are ignorant of how
24 it works, yet happy with what it does, that says a lot about the design of that system. By this measure, template type deduction in C++ is a tremendous success. Millions of programmers have passed arguments to template functions with completely satisfactory results, even though many of those programmers would be hard-pressed to give more than the haziest description of how the types used by those functions were deduced.

1 If that group includes you, I have good news and bad news. The good news is that
2 the type deduction process for `auto`-declared variables is essentially the same as
3 for templates (see Item 2), so when it comes to `auto`, you're on familiar ground.
4 The bad news is that when the template type deduction rules are employed by `au-`
5 `to`, you're more likely to be surprised by what happens. If you want to use `auto`
6 (and you certainly should—see Item 5), you'll need a reasonable understanding of
7 the rules that drive template type deduction. They're generally straightforward, so
8 this poses little challenge. It's just that they worked so naturally in C++98, you
9 probably never had to think much about them.

10 If you're willing to overlook a pinch of pseudocode, we can think of a function
11 template as looking like this:

```
12 template<typename T>
13 void f(ParamType param);
```

14 A call can look like this:

```
15 f(expr); // call f with some expression
```

16 During compilation, compilers use *expr* to deduce two types: one for `T` and one for
17 `ParamType`. These types are frequently different, because `ParamType` often con-
18 tains adornments, e.g., `const`- or reference qualifiers. For example, if the template
19 is declared like this,

```
20 template<typename T>
21 void f(const T& param); // ParamType is const T&
```

22 and we have this call,

```
23 int x = 0;
24 f(x); // call f with an int
```

25 `T` is deduced to be `int`, but `ParamType` is deduced to be `const int&`.

26 It's natural to expect that the type deduced for `T` is the same as the type of the ar-
27 gument passed to the function, i.e., that `T` is the type of *expr*. In the above example,
28 that's the case: `x` is an `int`, and `T` is deduced to be `int`. But it doesn't always work
29 that way. The type deduced for `T` is dependent not just of the type of *expr*, but also
30 on the form of `ParamType`. There are three cases:

- *ParamType* is a pointer or reference type, but not a universal reference. (Universal references are described in Item 26. At this point, all you need to know is that they exist.)
 - *ParamType* is a universal reference.
 - *ParamType* is neither a pointer nor a reference.

6 We therefore have three type deduction scenarios to examine. Each will be based
7 on our general form for templates and calls to it:

```
8 template<typename T>
9 void f(ParamType param);
10 f(expr); // deduce T and ParamType from expr
```

11 Case 1: *ParamType* is a Pointer or Reference, but not a Universal Reference

12 The simplest situation is when *ParamType* is a pointer type or a reference type,
13 but not a universal reference. In that case, type deduction works like this:

- If expr 's type is a reference, ignore the reference part.
 - Pattern-match expr 's type against ParamType to determine T.

16 For example, if this is our template,

```
17 template<typename T>
18 void f(T& param);      // param is a reference
```

19 and we have these variable declarations.

```
20 int x = 27;           // x is an int
21 const int cx = x;     // cx is a const int
22 const int& rx = x;   // rx is a read-only view of x
```

23 the deduced types for param and T in various calls are as follows:

1 In the second and third calls, notice that because `cx` and `rx` designate `const` val-
2 ues, `T` is deduced to be `const int`, thus yielding a parameter type of `const int&`.
3 That's important to callers. When they pass a `const` object to a reference parame-
4 ter, they expect that object to remain unmodifiable, i.e., for the parameter to be a
5 reference-to-`const`. That's why passing a `const` object to a template taking a `T&`
6 parameter is safe: the `constness` of the object becomes part of the type deduced
7 for `T`.

8 In the third example, note that even though `rx`'s type is a reference, `T` is deduced to
9 be a non-reference. That's because `rx`'s reference-ness is ignored during type de-
10 duction. If this were not the case (i.e., if `T` were deduced to be `const int&`),
11 `param`'s type would be `const int& &`, i.e., a reference to a reference. References to
12 references aren't permitted in C++, and one way they're avoided is by ignoring the
13 reference-ness of expressions during type deduction.

14 These examples all show lvalue reference parameters, but type deduction works
15 exactly the same way for rvalue reference parameters. Of course, only rvalue ar-
16 guments may be passed to rvalue reference parameters, but that restriction has
17 nothing to do with type deduction.

18 If we change the type of `f`'s parameter from `T&` to `const T&`, things change a little,
19 but not in any really surprising ways. The `constness` of `cx` and `rx` continues to be
20 respected, but because we're now assuming that `param` is a reference-to-`const`,
21 there's no longer a need for `const` to be deduced as part of `T`:

```
22 template<typename T>
23 void f(const T& param); // param is now a ref-to-const
24
25 int x = 27; // as before
26 const int cx = x; // as before
27 const int& rx = x; // as before
28
29 f(x); // T is int, param's type is const int&
30 f(cx); // T is int, param's type is const int&
31 f(rx); // T is int, param's type is const int&
```

30 As before, `rx`'s reference-ness is ignored during type deduction.

1 If `param` were a pointer (or a pointer to `const`) instead of a reference, things
2 would work essentially the same way:

```
3 template<typename T>
4 void f(T* param);           // param is now a pointer
5 int x = 27;                 // as before
6 const int *px = &x;          // px is a ptr to a read-only view of x
7
8 f(&x);                    // T is int, param's type is int*
9 f(px);                     // T is const int,
10                         // param's type is const int*,
```

11 At this point, you may find yourself yawning and nodding off, because C++'s type
12 deduction rules work so naturally for reference and pointer parameters, seeing
13 them in written form is really dull. Everything's just obvious! Which is exactly
14 what you want in a type deduction system.

15 Case 2: *ParamType* is a Universal Reference

16 Things are less obvious for templates taking universal reference parameters (i.e.,
17 “`T&&`” parameters), because lvalue arguments get special treatment. The complete
18 story is told in Item 26, but here's the headline version:

- 19 • If `expr` is an lvalue, both `T` and *ParamType* are deduced to be lvalue refer-
20 ences.
- 21 • If `expr` is an rvalue, the usual type deduction rules apply.

22 For example:

```
23 template<typename T>
24 void f(T&& param);        // param is now a universal reference
25 int x = 27;                 // as before
26 const int cx = x;           // as before
27 const int& rx = x;          // as before
28 f(x);                      // x is lvalue, so T is int&,
29                         // param's type is also int&
30 f(cx);                     // cx is lvalue, so T is const int&,
31                         // param's type is also const int&
```

```
1  f(rx);           // rx is lvalue, so T is const int&,
2                  // param's type is also const int&
3  f(27);          // 27 is rvalue, so T is int,
4                  // param's type is therefore int&&
```

5 Item 26 explains exactly why these examples play out the way they do, but the key
6 point is that the type deduction rules for parameters that are universal references
7 are different from those for parameters that are lvalue references or rvalue refer-
8 ences. In particular, when universal references are in use, type deduction distin-
9 guishes between lvalue arguments and rvalue arguments. That never happens for
10 non-universal (i.e., “normal”) references.

11 Case 3: *ParamType* is Neither a Pointer nor a Reference

12 When *ParamType* is neither a pointer nor a reference, we’re dealing with pass-by-
13 value:

```
14 template<typename T>
15 void f(T param);      // param is now passed by value
```

16 That means that *param* will be a copy of whatever is passed in—a completely new
17 object. The fact that *param* will be a new object motivates the rules that govern
18 how *T* is deduced from *expr*:

- 19 • As before, if *expr*’s type is a reference, ignore the reference part.
- 20 • If, after ignoring *expr*’s reference-ness, *expr* is *const*, ignore that, too. If it’s
21 *volatile*, also ignore that. (*volatile* objects are uncommon. They’re gener-
22 ally used only for implementing device drivers. For details, see Item 42.)

23 Hence:

```
24 int x = 27;           // as before
25 const int cx = x;     // as before
26 const int& rx = x;   // as before
27 f(x);                // T and param are both int
28 f(cx);               // T and param are again both int
29 f(rx);               // T and param are still both int
```

1 Note that even though `cx` and `rx` represent `const` values, `param` isn't `const`. That
2 makes sense. `param` is an object that's completely independent of `cx` and `rx`—a
3 *copy* of `cx` or `rx`. The fact that `cx` and `rx` can't be modified says nothing about
4 whether `param` can be. That's why `expr`'s `constness` (if any) is ignored when de-
5 ducing a type for `param`: just because `expr` can't be modified doesn't mean that a
6 copy of it can't be.

7 It's important to recognize that `const` is ignored only for by-value parameters. As
8 we've seen, for parameters that are references-to- or pointers-to-`const`, the `con-`
9 `stness` of `expr` is preserved during type deduction. But consider the case where
10 `expr` is a `const` pointer to a `const` object, and `expr` is passed to a by-value `param`:

```
11 template<typename T>
12 void f(T param);           // param is still passed by value
13 const char* const ptr =   // ptr is const pointer to const object
14   "Fun with pointers";
15 f(ptr);                  // pass arg of type const char * const
```

16 Here, the `const` to the right of the asterisk declares `ptr` to be `const`: `ptr` can't be
17 made to point to a different location, nor can it be set to null. (The `const` to the left
18 of the asterisk says that what `ptr` points to—the character string—is `const`, hence
19 can't be modified.) When `ptr` is passed to `f`, the bits making up the pointer are
20 copied into `param`. As such, *the pointer itself (ptr) will be passed by value*. In accord
21 with the type deduction rule for by-value parameters, the `constness` of `ptr` will be
22 ignored, and the type deduced for `param` will be `const char*`, i.e., a modifiable
23 pointer to a `const` character string. The `constness` of what `ptr` points to is pre-
24 served during type deduction, but the `constness` of `ptr` itself is ignored when
25 copying it to create the new pointer, `param`.

26 Array Arguments

27 That pretty much covers it for mainstream template type deduction, but there's a
28 sidestream case that is worth knowing about. It's that array types are different
29 from pointer types, even though they sometimes seem to be interchangeable. A
30 primary contributor to this illusion is that, in many contexts, an array *decays* into a
31 pointer to its first element. This decay is what permits code like this to compile:

```
1 const char name[] = "J. P. Briggs"; // name's type is
2 // const char[13]
3 const char * ptrToName = name; // array decays to pointer
4 Here, the const char* pointer ptrToName is being initialized with name, which is
5 a const char[13], i.e., a 13-element array of const char. These types (const
6 char* and const char[13]) are not the same, but because of the array-to-pointer
7 decay rule, the code compiles.
```

8 But what if an array is passed to a template taking a by-value parameter? What
9 happens then?

```
10 template<typename T>
11 void f(T param); // template with by-value parameter
12 f(name); // what types are deduced for T and param?
13 We begin with the observation that there is no such thing as a function parameter
14 that's an array. Yes, yes, the syntax is legal,
15 void myFunc(int param[]);
```

16 but the array declaration is treated as a pointer declaration, meaning that `myFunc`
17 could be equivalently declared like this:

```
18 void myFunc(int* param); // same function as above
19 This equivalence of array and pointer parameters is a bit of foliage springing from
20 the C roots at the base of C++, and it fosters the illusion that array and pointer
21 types are the same.
```

22 Because array parameter declarations are treated as if they were pointer parameter
23 declarations, the type of an array that's passed to a template function by value
24 is deduced to be a pointer type. That means that in the call to the template `f`, its
25 type parameter `T` is deduced to be `const char*`:

```
26 f(name); // name is array, but T deduced as const char*
27 But now comes a curve ball. Although functions can't declare parameters that are
28 truly arrays, they can declare parameters that are truly references to arrays! So if
29 we modify the template f to take its argument by reference,
```

```
1 template<typename T>
2 void f(T& param);           // template with by-reference parameter
3 and we pass an array to it,
4 f(name);                  // pass array to f
5 the type deduced for T is the actual type of the array! That type includes the size of
6 the array, so in this example, T is deduced to be const char [13], and the type of
7 f's parameter (a reference to this array) is const char (&)[13]. Yes, the syntax
8 looks toxic, but on the plus side, knowing it will score you mondo points with those
9 rare souls who care.
```

10 Interestingly, the ability to declare references to arrays enables creation of a tem-
11 plate to deduce the number of elements that an array contains:

```
12 template<typename T, std::size_t N>      // return size of
13 constexpr std::size_t arraySize(T (&)[N]) // an array as a
14 {                                         // compile-time
15     return N;                           // constant
16 }
```

17 Note the use of `constexpr` (see Item 14) to make the result of this function avail-
18 able during compilation. That makes it possible to declare, say, an array with the
19 same number of elements as a second array whose size is computed from list of
20 initializers:

```
21 int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };      // keyVals has
22                                         // 7 elements
23 int mappedVals[arraySize(keyVals)];             // so does
24                                         // mappedVals
```

25 Of course, as a modern C++ developer, you'd naturally prefer a `std::array` to a
26 built-in array:

```
27 std::array<int, arraySize(keyVals)> mappedVals; // mappedVals'
28                                         // size is 7
```

29 **Function Arguments**

30 Arrays aren't the only entities in C++ that can decay into pointers. Function types
31 can decay into function pointers, too, and everything we've discussed regarding

1 type deduction and arrays applies to type deduction for functions and their decay
2 into function pointers. As a result:

```
3 void someFunc(int, double);      // someFunc is a function;
4                               // type is void(int, double)

5 template<typename T>
6 void f1(T param);             // in f1, param passed by value

7 template<typename T>
8 void f2(T& param);           // in f2, param passed by ref

9 f1(someFunc);                // param deduced as ptr-to-func;
10                          // type is void (*)(int, double)

11 f2(someFunc);                // param deduced as ref-to-func;
12                          // type is void (&)(int, double)
```

13 This rarely makes any difference in practice, but if you're going to know about ar-
14 ray-to-pointer decay, you might as well know about function-to-pointer decay, too.

15 So there you have it: the rules for template type deduction. I remarked at the out-
16 set that they're pretty straightforward, and for the most part, they are. The special
17 treatment accorded lvalues when deducing types for universal references muddies
18 the water a bit, however, and the decay-to-pointer rules for arrays and functions
19 stirs up even greater turbidity. Sometimes you simply want to grab your compilers
20 and demand, "Tell me what type you're deducing!" When that happens, turn to
21 Item 4, because it's devoted to coaxing compilers into doing just that.

22 **Things to Remember**

- 23 ♦ When deducing types for parameters that are pointers and non-universal ref-
24 erences, whether the initializing expression is a reference is ignored.
- 25 ♦ When deducing types for parameters that are universal references, lvalue ar-
26 guments yield lvalue references, and rvalue arguments yield rvalue references.
- 27 ♦ When deducing types for by-value parameters, whether the initializing expres-
28 sion is a reference or is `const` is ignored.
- 29 ♦ During type deduction, arguments that are array or function names decay to
30 pointers, unless they're used to initialize references.

1 Item 2: Understand auto type deduction.

If you've read Item 1 on template type deduction, you already know almost everything you need to know about `auto` type deduction, because, with only one curious exception, `auto` type deduction *is* template type deduction. But how can that be? Template type deduction involves templates and functions and parameters, but `auto` deals with none of those things.

That's true, but it doesn't matter. There's a direct mapping between template type deduction and `auto` type deduction. There is literally an algorithmic transformation from one to the other.

10 In Item 1, template type deduction is explained using this general function tem-
11 plate

```
12 template<typename T>
13 void f(ParamType param);
```

14 and this general call:

16 In the call to `f`, compilers use `expr` to deduce types for `T` and `ParamType`.

17 When a variable is declared using `auto`, `auto` plays the role of `T` in the template,
18 and the type specifier for the variable acts as *ParamType*. This is easier to show
19 than to describe, so consider this example:

20 **auto** x = 27;

21 Here, the type specifier for x is simply auto by itself. On the other hand, in this
22 declaration,

23 **const auto** cx = x;

24 the type specifier is `const auto`. And here,

25 **const auto& rx = x;**

26 the type specifier is `const auto&`. To deduce types for `x`, `cx`, and `rx` in these ex-
27 amples, compilers act as if there were a template for each declaration as well as a
28 call to that template with the corresponding initializing expression:

```

1 template<typename T>           // conceptual template for
2 void func_for_x(T param);      // deducing x's type
3 func_for_x(27);               // conceptual call: param's
4                                         // deduced type is x's type
5 template<typename T>           // conceptual template for
6 void func_for_cx(const T param); // deducing cx's type
7 func_for_cx(x);               // conceptual call: param's
8                                         // deduced type is cx's type
9 template<typename T>           // conceptual template for
10 void func_for_rx(const T& param); // deducing rx's type
11 func_for_rx(x);               // conceptual call: param's
12                                         // deduced type is rx's type

```

13 As I said, deducing types for `auto` is the same as deducing types for templates.

14 Item 1 divides template type deduction into three cases, based on the characteristics of *ParamType*, the type specifier for `param` in the general function template. In
 15 a variable declaration using `auto`, the type specifier takes the place of *ParamType*,
 16 so there are three cases for that, too:

- 18 • Case 1: The type specifier is a pointer or reference, but not a universal reference.
- 19
- 20 • Case 2: The type specifier is a universal reference.
- 21 • Case 3: The type specifier is neither a pointer nor a reference.

22 We've already seen examples of cases 1 and 3:

```

23 auto x = 27;           // case 3 (x is neither ptr nor reference)
24 const auto cx = x;    // case 3 (cx isn't either)
25 const auto& rx = x;  // case 1 (rx is a non-universal ref.)

```

26 Case 2 works as you'd expect:

```

27 auto&& uref1 = x;    // x is int and lvalue,
28                                         // so uref1's type is int&
29 auto&& uref2 = cx;   // cx is const int and lvalue,
30                                         // so uref2's type is const int&

```

```

1 auto&& uref3 = 27;      // 27 is int and rvalue,
2                                // so uref3's type is int&&
3 Item 1 concludes with a discussion of how array and function names decay into
4 pointers for non-reference type specifiers. That happens in auto type deduction,
5 too, of course:
6
7 const char name[] =           // name's type is const char[13]
8     "R. N. Briggs";
9
10 auto arr1 = name;           // arr1's type is const char*
11
12 auto& arr2 = name;          // arr2's type is
13                                // const char (&)[13]
14
15 void someFunc(int, double); // someFunc is a function;
16                                // type is void(int, double)
17
18 auto func1 = someFunc;      // func1's type is
19                                // void (*) (int, double)
20
21 auto& func2 = someFunc;      // func2's type is
22                                // void (&)(int, double)
23
24 int x1 = 27;
25 int x2(27);
26
27 int x3 = {27};
28 int x4{27};
29
30 All in all, four syntaxes, but only one result: a variable with value 27.
31 But as Item 5 explains, there are advantages to declaring variables using auto in-
32 stead of fixed types, so it'd be nice to replace int with auto in the above variable
33 declarations. Straightforward textual substitution yields this code:

```

```
1 auto x1 = 27;  
2 auto x2(27);  
3 auto x3 = {27};  
4 auto x4{27};
```

5 These declarations all compile, but they don't have the same meaning as the ones
6 they replace. The first two statements do, indeed, declare a variable of type `int`
7 with value 27. The second two, however, declare a variable of type
8 `std::initializer_list<int>` containing a single element with value 27!

```
9 auto x1 = 27;           // type is int, value is 27  
10 auto x2(27);          // ditto  
11 auto x3 = {27};         // type is std::initializer_list<int>,  
12                      // value is {27}  
13 auto x4{27};          // ditto
```

14 This is due to a special type deduction rule for `auto`. When the initializer for an
15 `auto`-declared variable is enclosed in braces, the deduced type is a
16 `std::initializer_list`. If such a type can't be deduced (e.g., because the val-
17 ues in the braced initializer are of different types), the code will be rejected:

```
18 auto x5 = {1, 2, 3.0};    // error! can't deduce T for  
19                           // std::initializer_list<T>
```

20 As the comment indicates, type deduction will fail in this case, but it's important to
21 recognize that there are actually two kinds of type deduction taking place. One
22 kind stems from the use of `auto`: `x5`'s type has to be deduced. Because `x5`'s initial-
23 izer is in braces, `x5` must be deduced to be a `std::initializer_list`. But
24 `std::initializer_list` is a template. Instantiations are
25 `std::initializer_list<T>` for some type `T`, and that means that `T`'s type
26 must be deduced. Such deduction falls under the purview of the second kind of
27 type deduction occurring here: template type deduction. In this example, that de-
28 duction fails, because the values in the braced initializer don't have a single type.

29 The treatment of braced initializers is the only way in which `auto` type deduction
30 and template type deduction differ. When an `auto` variable is initialized with a
31 braced initializer, the deduced type is some instantiation of
32 `std::initializer_list`. If a template is faced with deducing the type for a

1 braced initializer, however, the code is rejected. (This has consequences for per-
2 fect forwarding, as Item 32 explains.)

3 You might wonder why `auto` type deduction has a special rule for braced initializ-
4 ers, but template type deduction does not. I wonder this, myself. Unfortunately, I
5 have not been able to find a compelling explanation. But the rule is the rule, and
6 this means that you must bear in mind that if you declare a variable using `auto`
7 and you initialize it with a braced initializer, the deduced type will always be
8 `std::initializer_list`. It's especially important to bear this in mind if you
9 embrace the philosophy of uniform initialization—of enclosing initializing values
10 in braces as a matter of course. One of the most classic mistakes in C++11 pro-
11 gramming is accidentally declaring a `std::initializer_list` variable when you
12 mean to declare something else. To reiterate:

```
13 auto x1 = 27;           // x1 and x2 are ints
14 auto x2(27);
15 auto x3 = {27};         // x3 and x4 are
16 auto x4{27};           // std::initializer_list<int>s
```

17 This pitfall is one of the reasons some developers put braces around their initializ-
18 ers only when they have to. (When you have to is discussed in Item 7.)

19 For C++11, this is the full story, but for C++14, the tale continues. C++14 permits
20 `auto` to indicate that a function's return type should be deduced (see Item 3), and
21 C++14 lambda expressions may use `auto` in parameter declarations. However,
22 these uses of `auto` employ *template type deduction*, not `auto` type deduction. That
23 means that braced initializers in these contexts cause type deduction to fail. So a
24 function with an `auto` return type that returns a braced initializer won't compile:

```
25 auto createInitList()
26 {
27     return { 1, 2, 3 };           // error: can't deduce type
28 }                           // for { 1, 2, 3 }
```

29 The same is true when `auto` is used in a parameter type specification in a C++14
30 lambda (thus yielding a generic lambda):

```
31 std::vector v;
32 ...
```

```
1 auto resetV =
2 [&v](const auto& newValue) { v = newValue; }; // C++14 only
3 ...
4 resetV( { 1, 2, 3 } ); // error! can't deduce type
5 // for { 1, 2, 3 }
```

6 The net result is that `auto` type deduction is identical to template type deduction
7 unless (1) a variable is being declared and (2) its initializer is inside braces. In that
8 case only, `auto` deduces a `std::initializer_list`, but template type deduction
9 fails.

10 Things to Remember

- 11 • `auto` type deduction is normally identical to template type deduction.
- 12 • The sole exception is that in variable declarations using `auto` and braced ini-
13 tializers, `auto` deduces `std::initializer_lists`.
- 14 • Template type deduction fails for braced initializers.

15 Item 3: Understand `decltype`.

16 `decltype` is a funny beast. Given a name or an expression, `decltype` tells you the
17 name's or the expression's type. Typically, what it tells you is exactly what you'd
18 predict. Occasionally however, it provides results that leave you scratching your
19 head and turning to reference works or online Q&A sites for revelation.

20 We'll begin with the typical cases—the ones harboring no surprises. In contrast to
21 what happens during type deduction for templates and `auto` (see Items 1 and 2),
22 `decltype` almost always parrots back the type of the name or expression you give
23 it without any modification:

```
24 const int i = 0; // decltype(i) is const int
25 bool f(const Widget& w); // decltype(w) is const Widget&
26 // decltype(f) is bool(const Widget&)
27 struct Point {
28     int x, y; // decltype(Point::x) is int
29 }; // decltype(Point::y) is int
30 Widget w; // decltype(w) is Widget
```

```
1 if (f(w)) ... // decltype(f(w)) is bool
2 template<typename T> // simplified version of std::vector
3 class vector {
4 public:
5 ...
6 T& operator[](std::size_t index);
7 ...
8 };
9 vector<int> v; // decltype(v) is vector<int>
10 ...
11 if (v[0] == 0) ... // decltype(v[i]) is int&
```

12 See? No surprises.

13 In C++11, the primary use for `decltype` is declaring function templates where the
14 function's return type depends on its parameter types. For example, suppose we'd
15 like to write a function that takes a container that supports indexing via square
16 brackets (i.e., the use of "`[]`") plus an index, then authenticates the user before re-
17 turning the result of the indexing operation. The return type of the function should
18 be the same as the type returned by the indexing operation.

19 `operator[]` on a container of objects of type `T` typically returns a `T&`. This is the
20 case for `std::deque`, for example, and it's almost always the case for
21 `std::vector`. For `std::vector<bool>`, however, `operator[]` does not return a
22 `bool&`. Instead, it returns a brand new object. The whys and hows of this situation
23 are explored in Item 6, but what's important here is that the type returned by a
24 container's `operator[]` depends on the container.

25 `decltype` makes it easy to express that. Here's a first cut at the template we'd like
26 to write, showing the use of `decltype` to compute the return type. The template
27 needs a bit of refinement, but we'll defer that for now:

```
28 template<typename Container, typename Index> // works, but
29 auto authAndAccess(Container& c, Index i) // requires
30 -> decltype(c[i]) // refinement
31 {
32     authenticateUser();
33     return c[i];
34 }
```

1 The use of `auto` before the function name has nothing to do with type deduction.
2 Rather, it indicates that C++11's *trailing return type* syntax is being used, i.e., that
3 the function's return type will be declared following the parameter list (after the
4 "`->`"). A trailing return type has the advantage that the function's parameters can
5 be used in the specification of its return type. In `authAndAccess`, for example, we
6 specify the return type using `c` and `i`. If we were to have the return type precede
7 the function name in the conventional fashion, `c` and `i` would be unavailable, be-
8 cause they would not have been declared yet.

9 With this declaration, `authAndAccess` returns whatever type `operator[]` re-
10 turns when applied to the passed-in container, exactly as we desire.

11 C++11 permits return types for single-statement lambdas to be deduced, and
12 C++14 extends this to both all lambdas and all functions (including those with mul-
13 tiple statements). In the case of `authAndAccess`, that means that in C++14 we can
14 omit the trailing return type, leaving just the leading `auto`. With that form of dec-
15 laration, `auto` *does* mean that type deduction will take place. In particular, it
16 means that compilers will deduce the function's return type from the function's
17 implementation:

```
18 template<typename Container, typename Index> // C++14 only, and
19 auto authAndAccess(Container& c, Index i) // not quite
20 {                                         // correct
21     authenticateUser();
22     return c[i];                         // return type deduced from c[i]
23 }
```

24 But which of C++'s type deduction rules will be used to infer `authAndAccess`'s
25 return type: those for templates, those for `auto`, or those for `decltype`?

26 Perhaps surprisingly, functions with an `auto` return type perform type deduction
27 using the template type deduction rules. It might seem that use of the `auto` type
28 deduction rules would better correspond to the declaration syntax, but remember
29 that template type deduction and `auto` type deduction are nearly identical. The
30 only difference is that template type deduction deduces no type for a braced ini-
31 tializer.

1 In this case, deducing `authAndAccess`'s return type using template type deduction
2 is problematic, but `auto`'s type deduction rules would fare no better. The difficulty
3 stems from something these forms of type deduction have in common: their
4 treatment of expressions that are references.

5 As we've discussed, `operator[]` for most containers-of-T returns a `T&`, but Item 1
6 explains that during template type deduction, the reference-ness of an initializing
7 expression is ignored. Consider what that would mean for this client code using
8 the declaration for `authAndAccess` with an `auto` return type (i.e., using template
9 type deduction for its return type):

```
10 std::deque<int> d;
11 ...
12 authAndAccess(d, 5) = 10; // authenticate user, return d[5],
13 // then assign 10 to it;
14 // this won't compile!
```

15 Here, `d[5]` returns an `int&`, but `auto` return type deduction for `authAndAccess`
16 will strip off the reference, thus yielding a return type of `int`. That `int`, being the
17 return value of a function, is an rvalue, and the code above thus attempts to assign
18 10 to an rvalue `int`. That's forbidden in C++, so the code won't compile.

19 The problem is that we're using template type deduction, which discards reference
20 qualifiers from its initializing expression. What we want in this case is `decltype`
21 type deduction. Such type deduction would permit us to say that `authAndAccess`
22 should return exactly the same type that the expression `c[i]` returns.

23 The guardians of C++, anticipating the need to use `decltype` type deduction rules
24 in some cases where types are inferred, make this possible in C++14 through the
25 `decltype(auto)` specifier. What may initially seem contradictory (`decltype` and
26 `auto`?) actually makes perfect sense: `auto` specifies that the type is to be deduced,
27 and `decltype` says that `decltype` rules should be used during the deduction. We
28 can thus write `authAndAccess` like this:

```
29 template<typename Container, typename Index> // C++14 only;
30 decltype(auto) // works, but
31 authAndAccess(Container& c, Index i) // still requires
32 { // refinement
33     authenticateUser();
```

```
1     return c[i];
2 }
```

3 Now `authAndAccess` will truly return whatever `c[i]` returns. In particular, for
4 the common case where `c[i]` returns a `T&`, `authAndAccess` will also return a `T&`,
5 and in the uncommon case where `c[i]` returns an object, `authAndAccess` will
6 return an object, too.

7 The use of `decltype(auto)` is not limited to function return types. It can also be
8 convenient for declaring variables when you want to apply the `decltype` type de-
9 duction rules to the initializing expression:

```
10 Widget w;
11 const Widget& cw = w;
12 auto myWidget1 = cw;           // auto type deduction:
13                           // myWidget1's type is Widget
14 decltype(auto) myWidget2 = cw; // decltype type deduction:
15                           // myWidget2's type is
16                           // const Widget&
```

17 But two things are bothering you, I know. One is the refinement to `authAndAc-`
18 `cess` I mentioned, but have not yet described. Let's address that now.

19 Look again at the declaration for the C++14 version of `authAndAccess`:

```
20 template<typename Container, typename Index>
21 decltype(auto) authAndAccess(Container& c, Index i);
```

22 The container is passed by lvalue-reference-to-non-const, because returning a
23 reference to an element of the container permits clients to modify that container.
24 But this means it's not possible to pass rvalue containers to this function. Rvalues
25 can't bind to lvalue references (unless they're lvalue-references-to-const, which is
26 not the case here).

27 Admittedly, passing an rvalue container to `authAndAccess` is an edge case. An
28 rvalue container, being a temporary object, would typically be destroyed at the
29 end of the statement containing the call to `authAndAccess`, and that means that a
30 reference to an element in that container (which is typically what `authAndAccess`
31 would return) would dangle at the end of the statement that created it. Still, it

1 could make sense to pass a temporary object to `authAndAccess`. A client might
2 simply want to make a copy of an element in the temporary container, for exam-
3 ple:

```
4 std::deque<std::string> makeStringDeque(); // factory function
5 // make copy of 5th element of deque returned
6 // from makeStringDeque
7 auto s = authAndAccess(makeStringDeque(), 5);
```

8 Supporting such use means we need to revise the declaration for `c` to accept both
9 lvalues and rvalues, and that means that `c` needs to be a universal reference (see
10 Item 26):

```
11 template<typename Container, typename Index>
12 decltype(auto) authAndAccess(Container&& c, Index i);
```

13 In this template, we don't know what type of container we're operating on, and
14 that means we're equally ignorant of the type of index objects it uses. Employing
15 pass-by-value for objects of an unknown type generally risks the performance hit
16 of unnecessary copying, the behavioral problems of object slicing (see Item 17),
17 and the sting of our coworkers' derision, but in the case of container indices, fol-
18 lowing the example of the Standard Library (e.g., `std::string`, `std::vector`,
19 and `std::deque`) seems reasonable, so we'll stick with pass-by-value for them.

20 All that remains is to update the template's implementation to bring it into accord
21 with Item 27's admonition to apply `std::forward` to universal references:

```
22 template<typename Container, typename Index> // final
23 decltype(auto) // C++14
24 authAndAccess(Container&& c, Index i) // version
25 {
26     authenticateUser();
27     return std::forward<Container>(c)[i];
28 }
```

29 This should do everything we want, but it requires a C++14 compiler. If you don't
30 have one, you'll need to use the C++11 version of the template. It's the same as its
31 C++14 counterpart, except that you have to specify the return type yourself:

```
32 template<typename Container, typename Index> // final
33 auto // C++11
34 authAndAccess(Container&& c, Index i) // version
```

```
1 -> decltype(std::forward<Container>(c)[i])
2 {
3     authenticateUser();
4     return std::forward<Container>(c)[i];
5 }
```

6 The other issue that's likely to be nagging at you is my remark at the beginning of
7 this Item that `decltype` *almost* always produces the type you expect, that it *rarely*
8 surprises. Truth be told, you're unlikely to encounter these exceptions to the rule
9 unless you're a heavy-duty library implementer.

10 To *fully* understand `decltype`'s behavior, you'll have to familiarize yourself with a
11 few special cases. Most of these are too obscure to warrant discussion in a book
12 like this, but looking at one lends insight into `decltype` as well as its use.

13 Applying `decltype` to a name yields the declared type for that name, just as I said.
14 Names are lvalue expressions, but that doesn't affect `decltype`'s behavior. For
15 lvalue expressions more complicated than names, however, `decltype` ensures
16 that the type reported is always an lvalue reference. That is, if an lvalue expression
17 other than a name has type `T`, `decltype` reports that type as `T&`. This rarely has
18 any impact, because the type of most lvalue expressions inherently includes an
19 lvalue reference qualifier. Functions returning lvalues, for example, always return
20 lvalue references.

21 There is an implication of this behavior that is worth being aware of, however. In

```
22 int x = 0;
```

23 `x` is the name of a variable, so `decltype(x)` is `int`. But wrapping the name `x` in pa-
24 rentheses—“(x)”—yields an expression more complicated than a name. Being a
25 name, `x` is an lvalue, and C++ defines (x) to be an lvalue, too. `decltype((x))` is
26 therefore `int&`. Putting parentheses around a name can change the type that
27 `decltype` reports for it!

28 In C++11, this is little more than a curiosity, but in conjunction with C++14's sup-
29 port for `decltype(auto)`, it means that a seemingly trivial change in the way you
30 write a `return` statement can affect the deduced type for a function:

```
1 decltype(auto) f1()
2 {
3     int x = 0;
4     ...
5     return x;           // decltype(x) is int, so f1 returns int
6 }
7 decltype(auto) f2()
8 {
9     int x = 0;
10    ...
11    return (x);        // decltype((x)) is int&, so f2 returns int&
12 }
```

13 Note that not only does `f2` have a different return type from `f1`, but it's also returning a reference to a local variable! That's the kind of code that puts you on the express train to undefined behavior—a train you most assuredly don't want to be on.

17 The primary lesson here is to pay very close attention when using `decltype(auto)`. Seemingly insignificant details in the expression whose type is being deduced can affect the type that `decltype(auto)` reports. To ensure that the type being deduced is the type you expect, use the techniques described in Item 4.

22 At the same time, don't lose sight of the bigger picture. Sure, `decltype` (both alone and in conjunction with `auto`) may occasionally yield type-deduction surprises, but that's not the normal situation. Normally, `decltype` produces the type you expect. This is especially true when `decltype` is applied to names, because in that case, `decltype` does just what it sounds like: it reports that name's declared type.

27 **Things to Remember**

- 28 ♦ `decltype` almost always yields the type of a variable or expression without
29 any modifications.
- 30 ♦ For lvalue expressions of type T other than names, `decltype` always reports a
31 type of T&.
- 32 ♦ C++14 supports `decltype(auto)`, which, like `auto`, deduces a type from its
33 initializer, but it performs the type deduction using the `decltype` rules.

1 **Item 4: Know how to view deduced types.**

2 People who want to see the types that compilers deduce usually fall into one of
3 two camps. The first are the pragmatists. They're typically motivated by a behav-
4 ioral problem in their software (i.e., they're debugging), and they're looking for
5 insights into compilation that can help them identify the source of the problem.
6 The second are the experimentalists. They're exploring the type deduction rules
7 described in Items 1-3. Often, they want to confirm their predictions about the re-
8 sults of various type deduction scenarios ("For this code, I think compilers will de-
9 duce *this* type..."), but sometimes they simply want to answer "what if" questions.
10 "How," they might wonder, "do the results of template type deduction change if I
11 replace a universal reference (see Item 26) with an lvalue-reference-to-const pa-
12 rameter (i.e., replace `T&&` with `const T&` in a function template parameter list)?"

13 Regardless of the camp you fall into (both are legitimate), the tools you have at
14 your disposal depend on the phase of the software development process where
15 you'd like to see the types your compilers have inferred. We'll explore three possi-
16 bilities: getting type deduction information as you edit your code, getting it during
17 compilation, and getting it at runtime.

18 **IDE Editors**

19 Code editors in IDEs often show the types of program entities (e.g., variables,
20 parameters, functions, etc.) when you do something like hover your cursor over the
21 entity. For example, given this code,

22 `const int theAnswer = 42;`
23 `auto x = theAnswer;`
24 `auto y = &theAnswer;`

25 an IDE editor would likely show that `x`'s deduced type was `int` and `y`'s was `const`
26 `int*`.

27 For this to work, your code must be in a more or less compilable state, because
28 what makes it possible for the IDE to offer this kind of information is a C++ com-
29 piler running inside the IDE. If that compiler can't make enough sense of your code
30 to parse it and perform type deduction, it can't show you what types it deduced.

1 **Compiler Diagnostics**

2 An effective way to get a compiler to show a type it has deduced is to use that type
3 in a way that leads to compilation problems. The error message reporting the
4 problem is virtually sure to mention the type that's causing it.

5 Suppose, for example, we'd like to see the types that were deduced for `x` and `y` in
6 the previous example. We first declare a class template that we *don't define*. Some-
7 thing like this does nicely:

```
8    template<typename T>         // declaration only for TD;  
9    class TD;                     // TD == "Type Displayer"
```

10 Attempts to instantiate this template will elicit an error message, because there's
11 no template definition to instantiate. To see the types for `x` and `y`, just try to instant-
12 iate `TD` with their types:

```
13   TD<decltype(x)> xType;      // elicit errors containing  
14   TD<decltype(y)> yType;      // x's and y's types;  
15                                  // see Item 3 for decltype info
```

16 I use variable names of the form `variableNameType`, because they tend to yield
17 quite informative error messages. For the code above, one of my compilers issues
18 diagnostics reading, in part, as follows. (I've highlighted the type information
19 we're looking for.)

```
20   error: aggregate 'TD<int> xType' has incomplete type and  
21        cannot be defined  
22   error: aggregate 'TD<const int *> yType' has incomplete type  
23        and cannot be defined
```

24 A different compiler provides the same information, but in a different form:

```
25   error: 'xType' uses undefined class 'TD<int>'  
26   error: 'yType' uses undefined class 'TD<const int *>'
```

27 Formatting differences aside, all the compilers I've tested produce error messages
28 with useful type information when this technique is employed.

29 **Runtime Output**

30 The `printf` approach to displaying type information (not that I'm recommending
31 you use `printf`) can't be employed until runtime, but it offers full control over the

1 formatting of the output. The challenge is to create a textual representation of the
2 type you care about that is suitable for display. “No sweat,” you’re thinking, “it’s
3 `typeid` and `std::type_info::name` to the rescue.” In our continuing quest to
4 see the types deduced for `x` and `y`, you figure, we can write this:

```
5 std::cout << typeid(x).name() << '\n'; // display types for
6 std::cout << typeid(y).name() << '\n'; // x and y
```

7 This approach relies on the fact that invoking `typeid` on an object such as `x` or `y`
8 yields a `std::type_info` object, and `std::type_info` has a member function,
9 `name`, that produces a C-style string (i.e., a `const char*`) representation of the
10 name of the type.

11 Calls to `std::type_info::name` are not guaranteed to return anything sensible,
12 but implementations try to be helpful. The level of helpfulness varies. The GNU and
13 Clang compilers report that the type of `x` is “`i`”, and the type of `y` is “`PKi`”, for ex-
14 ample. These results make more sense once you learn that, in output from these
15 compilers, “`i`” means “`int`” and “`PK`” means “pointer to `konst const`.” (Both com-
16 pilers support a tool, `c++filt`, that decodes such “mangled” types.) Microsoft’s com-
17 piler produces less cryptic output: “`int`” for `x` and “`int const *`” for `y`.

18 Because these results are correct for the types of `x` and `y`, you might be tempted to
19 view the type-reporting problem as solved, but let’s not be hasty. Consider a more
20 complex example:

```
21 template<typename T> // template function to
22 void f(const T& param); // be called
23 std::vector<Widget> createVec(); // factory function
24 const auto vw = createVec(); // init vw w/factory return
25 if (!vw.empty()) {
26     f(&vw[0]); // call f
27     ...
28 }
```

29 This code, which involves a user-defined type (`Widget`), an STL container
30 (`std::vector`), and an `auto` variable (`vw`), is more representative of the situa-
31 tions where you might want some visibility into the types your compilers are de-

1 ducing. For example, it'd be nice to know what types are inferred for the template
2 type parameter `T` and the function parameter `param` in `f`.

3 Loosing `typeid` on the problem is straightforward. Just add some code to `f` to dis-
4 play the types you'd like to see:

```
5 template<typename T>
6 void f(const T& param)
7 {
8     using std::cout;
9
10    cout << "T = " << typeid(T).name() << '\n'; // show T
11    cout << "param = " << typeid(param).name() << '\n'; // show
12    ... // param's
13 } // type
```

13 Executables produced by the GNU and Clang compilers produce this output:

```
14 T = PK6Widget
15 param = PK6Widget
```

16 We already know that for these compilers, `PK` means “pointer to `const`,” so the
17 only mystery is the number `6`. That's simply the number of characters in the class
18 name that follows `(Widget)`. So these compilers tell us that both `T` and `param` are
19 of type `const Widget*`.

20 Microsoft's compiler concurs:

```
21 T = class Widget const *
22 param = class Widget const *
```

23 Three independent compilers producing the same information suggests that the
24 information is accurate. But look more closely. In the template `f`, `param`'s declared
25 type is `const T&`. That being the case, doesn't it seem odd that `T` and `param` have
26 the same type? If `T` were `int`, for example, `param`'s type should be `const int&`—
27 not the same type at all.

28 Sadly, the results of `std::type_info::name` are not reliable. In this case, for ex-
29 ample, the type that all three compilers report for `param` are incorrect. Further-
30 more, they're essentially *required* to be incorrect, because the specification for
31 `std::type_info::name` mandates that the type being processed be treated as if

1 it had been passed to a template function as a by-value parameter. As Item 1 explains, that means that if the type is a reference, its reference-ness is ignored, and
2 if the type after reference removal is `const`, its `constness` is also ignored. That's
3 why `param`'s type—which is `const Widget * const &`—is reported as `const`
4 `Widget*`. First the type's reference-ness is removed, and then the `constness` of
5 the result type is eliminated.

6
7 Equally sadly, the type information displayed by IDE editors is also not reliable—
8 or at least not reliably useful. For this same example, one IDE editor I know reports
9 `T`'s type as (I am not making this up):

```
10 const
11 std::Simple_types<std::_Wrap_alloc<std::Vec_base_types<Widget,
12 std::allocator<Widget> >::_Alloc>::value_type>::value_type *
```

13 The same IDE editor shows `param`'s type as:

```
14 const std::Simple_types<...>::value_type *const &
```

15 That's less intimidating than the type for `T`, but the “`...`” in the middle is disturbing
16 until you realize that it's the IDE editor's way of saying “I'm omitting all that
17 stuff that's part of `T`'s type.”

18 My understanding is that most of what's displayed here is `typedef` cruft and that
19 once you push through the `typedefs` to get to the underlying type information,
20 you get what you're looking for, but having to do that work pretty much eliminates
21 any utility the display of the types in the IDE originally promised. With any luck,
22 your IDE editor does a better job on code like this.

23 In my experience, compiler diagnostics are a more dependable source of information about the results of type deduction. Revising the template `f`'s implementation to instantiate the declared-but-not-defined template `TD` yields this:

```
26 template<typename T>
27 void f(const T& param)
28 {
29     TD<T> TType;                      // elicit errors containing
30     TD<decltype(param)> paramType;    // T's and param's types
31     ...
32 }
```

1 Each of GNU's, Clang's, and Microsoft's compilers produce error messages with the
2 correct types for `T` and `param`. The exact message contents and formats vary, but
3 as an example, this is what GNU's compiler issues (after minor reformatting):

```
4 error: 'TD<const Widget *> TType' has incomplete type
5 error: 'TD<const Widget * const &> paramType' has incomplete
6 type
```

7 **Beyond typeid**

8 If you want accurate runtime information about deduced types, we've seen that
9 `typeid` is not a reliable route to getting it. One way to work around that is to im-
10 plement your own mechanism for mapping from a type to its displayable repre-
11 sentation. In concept, it's not difficult: you just use type traits and template met-
12aprogramming (see Item 9) to break a type into its various components (using
13 type traits such as `std::is_const`, `std::is_pointer`,
14 `std::is_lvalue_reference`, etc.), and you create a string representation of the
15 type from textual representations of each of its parts. (You'd still be dependent on
16 `typeid` and `std::type_info::name` to generate string representations of the
17 names of user-defined classes, though.)

18 If you'd use such a facility often enough to justify the effort needed to write, debug,
19 document, and maintain it, that's a reasonable approach. But if you're willing to
20 live with a little platform-dependent code that's easy to implement and produces
21 better results than those based on `typeid`, it's worth noting that many compilers
22 support a language extension that yields a printable representation of the full sig-
23 nature for a function, including, for functions generated from templates, types for
24 both template and function parameters.

25 For example, the GNU and Clang compilers support a construct called
26 `__PRETTY_FUNCTION__`, and Microsoft's compiler offers `__FUNCSIG__`. These
27 constructs represent a variable (for GNU and Clang) or a macro (for Microsoft)
28 whose value is the signature of the containing function. If we reimplement our
29 template `f` like this,

```
30 template<typename T>
31 void f(const T& param)
32 {
```

```
1 #if defined(__GNUC__)
2     std::cout << __PRETTY_FUNCTION__ << '\n';      // For GNU and
3 #elif defined(_MSC_VER)
4     std::cout << __FUNCSIG__ << '\n';              // For Microsoft
5 #endif
6 ...
7 }
```

8 and call `f` as we did before,

```
9 std::vector<Widget> createVec();           // factory function
10 const auto vw = createVec();                // init vw w/factory return
11 if (!vw.empty()) {
12     f(&vw[0]);                            // call f
13 ...
14 }
```

15 we get the following result from GNU:

```
16 void f(const T&) [with T = const Widget*]
```

17 This tells us that `T` has been deduced to be `const Widget*` (the same thing we got
18 via `typeid`, but without the “PK” encoding and the “6” in front of the class name),
19 but it also tells us that `f`’s parameter has type `const T&`. If we expand `T` in that
20 formulation, we get `const Widget * const &`. That’s different from what `typeid`
21 told us, though it’s the same as the type in the error message provoked by use of
22 the declared-but-not-defined `TD` template. It’s also correct.

23 Use of Microsoft’s `__FUNCSIG__` produces this output:

```
24 void __cdecl f<const classWidget*>(const class Widget *const &)
```

25 The type inside the angle brackets is the type deduced for `T`: `const Widget*`. This,
26 too, is what we got via `typeid`. But the type inside parentheses is the type de-
27 duced for `param`: `const Widget * const&`. That’s not what `typeid` told us,
28 though, again, it’s the same as the (correct) information we’d get during compila-
29 tion from use of the `TD` template.

1 Clang's function-signature-reporting facility, despite using the same name as
2 GNU's (`__PRETTY_FUNCTION__`), is not as forthcoming as GNU's or Microsoft's. It
3 yields simply:

4 `void f(const Widget *const &)`

5 This shows `param`'s type directly, but it leaves it up to you to deduce that `T`'s type
6 must have been `const Widget*` (or to rely on the information provided via
7 `typeid`).

8 IDE editors, compiler error messages, `typeid`, and language extensions like
9 `__PRETTY_FUNCTION__` and `__FUNCSIG__` are merely tools you can use to help
10 you figure out what types your compilers are deducing for you. All can be helpful,
11 but at the end of the day, there's no substitute for understanding the type deduc-
12 tion information in Items 1-3.

13 **Things to Remember**

- 14 ♦ Deduced types can often be seen using IDE editors, compiler error messages,
15 `typeid`, and language extensions such as `__PRETTY_FUNCTION__` and
16 `__FUNCSIG__`.
- 17 ♦ The results of such tools may be neither helpful nor accurate, so an under-
18 standing of C++'s type deduction rules remains essential.

1 **Chapter 2 auto**

2 It's been a wild ride for `auto`. In 1983, Bjarne Stroustrup took this lonely and
3 largely superfluous keyword and gave it new purpose in his C++ compiler: to de-
4 clare a variable whose type was deduced from its initializing expression. The move
5 proved too bold for the C developers who were C++'s target audience, so `auto`'s
6 revised semantics were rolled back, and the keyword retired to the home for trivia
7 topics. ("What is `auto` used for?")

8 Nearly three decades later, it debuted a second time (with the same semantics
9 Bjarne had instilled in it so many years before) and thereby became C++11's oldest
10 new feature. It's also probably C++11's most frequently used one, because few acts
11 are as common as declaring variables. In C++14, its utility gains new ground:
12 lambda parameters may incorporate `auto`, and function return types may be au-
13 to-deduced. It took a long time for `auto` to make it into the limelight, but its tenure
14 there now seems secure.

15 In concept, `auto` is as simple as simple can be, but it's more subtle than it looks.
16 Using it saves typing, sure, but it also prevents correctness and performance issues
17 that can bedevil manual type declarations. Furthermore, some of `auto`'s type de-
18 duction results, while dutifully conforming to the prescribed algorithm, are, from
19 the perspective of a programmer, just plain wrong. When that's the case, it's im-
20 portant to know how to guide `auto` to the right answer, because falling back on
21 manual type declarations is an alternative that's often best avoided.

22 This brief chapter covers all of `auto`'s ins and outs.

23 **Item 5: Prefer `auto` to explicit type declarations.**

24 Ah, the simple joy of

25 `int x;`

26 Wait. Damn. I forgot to initialize `x`, so its value is indeterminate. Maybe. It might
27 actually be initialized to zero. Depends on the context. Sigh.

1 Never mind. Let's move on to the simple joy of declaring a local variable to be ini-
2 tialized with the value of an iterator:

```
3 template<typename It>      // algorithm to dwim ("do what I mean")
4 void dwim(It b, It e)        // for all elements in range from
5 {                           // b to e
6     while (b != e) {
7         typename std::iterator_traits<It>::value_type
8         currValue = *b;
9         ...
10    }
11 }
```

12 Ugh. “`typename std::iterator_traits<It>::value_type`” to express the
13 type of the value pointed to by an iterator? Really? I must have blocked out the
14 memory of how much fun that is. Damn. Wait—didn't I already say that?

15 Okay, simple joy (take three): the delight of declaring a local variable whose type is
16 the same as that resulting from a lambda expression. Oh, right. The type of a lamb-
17 da expression is known only to the compiler, hence can't be written out. Sigh.
18 Damn.

19 Damn, damn, damn! Programming in C++ is not the joyous experience it should be!

20 Well, it didn't used to be. But as of C++11, all these issues go away, courtesy of au-
21 to. `auto` variables have their type deduced from their initializer, so they must be
22 initialized. That means you can wave goodbye to a host of uninitialized variable
23 problems as you speed by on the C++11 superhighway:

```
24 int x;                  // valid in C++ 98/11/14, potentially uninitialized
25 auto x;                 // error! initializer required
26 auto x = 0;              // fine, x's value is well-defined
```

27 Said highway also paves over the potholes associated with declaring a local varia-
28 ble whose value is determined by an iterator:

```
29 template<typename It>      // as before
30 void dwim(It b, It e)
31 {
32     while (b != e) {
33         auto currValue = *b;
34         ...
```

```
1      }
2 }
```

3 And because `auto` is based on type deduction, it can represent types known only
4 to compilers:

```
5 auto derefUPLess =                      // comparison func.
6   [](const std::unique_ptr<Widget>& p1,    // for Widgets
7     const std::unique_ptr<Widget>& p2)    // pointed to by
8   { return *p1 < *p2; };                  // std::unique_ptrs
```

9 Very cool. In C++14, the coolness chills further, because in C++14, parameters to
10 lambda expressions may involve `auto`:

```
11 auto derefLess =                      // comparison func.
12   [](const auto& p1,                 // for values pointed
13     const auto& p2)                 // to by anything
14   { return *p1 < *p2; };            // pointer-like
```

15 Coolness notwithstanding, perhaps you're thinking that we don't really need `auto`
16 to declare a variable that holds a closure, because we can use a `std::function`
17 object:

```
18 std::function<bool(const std::unique_ptr<Widget>&,
19   const std::unique_ptr<Widget>&)>
20 derefUPLess = [](const std::unique_ptr<Widget>& p1,
21   const std::unique_ptr<Widget>& p2)
22   { return *p1 < *p2; };
```

23 You're right, but it's important to recognize that—even setting aside the syntactic
24 verbosity and the need to repeat the parameter types—using `std::function` is
25 not the same thing as using `auto`. An `auto`-declared variable holding the result of a
26 lambda expression has the same type as the lambda expression. The type of a
27 `std::function`-declared variable holding the result of a lambda expression is
28 some instantiation of the `std::function` template. The constructor for such in-
29 stantiations may allocate heap memory. (Whether it does depends on how much
30 data is captured by the lambda and on how `std::function` is implemented). `au-`
31 `to` never allocates heap memory. The `std::function` object uses more memory
32 than the `auto`-declared object. And, thanks to implementation details that restrict
33 inlining and yield indirect function calls, invoking a lambda's closure via a
34 `std::function` object is almost certain to be slower than calling it via an `auto`-
35 declared object. In other words, the `std::function` approach is generally bigger

1 and slower than the `auto` approach, and it may yield out-of-memory exceptions,
2 too. Plus, as you can see in the examples above, writing “`auto`” is a whole lot less
3 work than writing the type of the `std::function` instantiation. In the competi-
4 tion between `auto` and `std::function` for holding the result of a lambda expres-
5 sion, it’s pretty much game, set, and match for `auto`. (A similar argument can be
6 made for `auto` over `std::function` for holding the result of calls to `std::bind`,
7 though in Item 36, I do my best to convince you to use lambdas instead of
8 `std::bind`, anyway.)

9 The advantages of `auto` extend beyond the avoidance of uninitialized variables,
10 verbose variable declarations, and the ability to directly hold the result of lambda
11 expressions. One is the ability to avoid what I call problems related to “type
12 shortcuts.” Here’s something you’ve probably seen—possibly even written:

```
13 std::vector<int> v;  
14 ...  
15 unsigned sz = v.size();
```

16 The official return type of `v.size()` is `std::vector<int>::size_type`, but few
17 developers are aware of that. `std::vector<int>::size_type` is specified to be
18 an unsigned integral type, so a lot of programmers figure that `unsigned` is good
19 enough and write code such as the above. This can have some interesting conse-
20 quences. On 32-bit Windows, for example, both `unsigned` and
21 `std::vector<int>::size_type` are the same size, but on 64-bit Windows, un-
22 signed is 32 bits, while `std::vector<int>::size_type` is 64 bits. This means
23 that code that works under 32-bit Windows may behave incorrectly under 64-bit
24 Windows, and when porting your application from 32 to 64 bits, who wants to
25 spend time on issues like that?

26 Using `auto` ensures that you don’t have to:

```
27 auto sz = v.size(); // sz's type is std::vector<int>::size_type
```

28 Still unsure about the wisdom of using `auto`? Then consider this code:

```
29 std::unordered_map<std::string, int> m;  
30 ...
```

```
1 for (const std::pair<std::string, int>& p : m)
2 {
3     ...
4 }
```

5 This looks perfectly reasonable, but there's a problem. Do you see it?

6 Recognizing what's amiss requires remembering that the key part of a
7 `std::unordered_map` is `const`, so the type of `std::pair` in the hash table
8 (which is what a `std::unordered_map` is) isn't `std::pair<std::string,`
9 `int>`, it's `std::pair<const std::string, int>`. But that's not the type de-
10 clared for the variable `p` in the loop above. As a result, compilers will strive to find
11 a way to convert `std::pair<const std::string, int>` objects (i.e., what's in
12 the hash table) to `std::pair<std::string, int>` objects (the declared type for
13 `p`). They'll succeed by creating a temporary object of the type that `p` wants to bind
14 to by copying each object in `m`, then binding the reference `p` to that temporary ob-
15 ject. At the end of each loop iteration, the temporary object will be destroyed. If
16 you wrote this loop, you'd likely be surprised by this behavior, because you'd al-
17 most certainly intend to simply bind the reference `p` to each element in `m`.

18 Such unintentional type mismatches can be `autoed` away:

```
19 for (const auto& p : m)
20 {
21     ...
22 }
```

23 This is not only more efficient, it's also easier to type. Furthermore, this code has
24 the very attractive characteristic that if you take `p`'s address, you're sure to get a
25 pointer to an element within `m`. In the code not using `auto`, you'd get a pointer to a
26 temporary object—an object that would be destroyed at the end of the loop itera-
27 tion.

28 The last two examples—writing `unsigned` when you should have written
29 `std::vector<int>::size_type` and writing `std::pair<std::string, int>`
30 when you should have written `std::pair<const std::string, int>`—
31 demonstrate how explicitly specifying types can lead to implicit conversions that
32 you neither want nor expect. Such conversions arise whenever the type of the ini-
33 tializing expression is different from the type you specify for the target variable,

1 but an implicit conversion exists from one to the other. It generally arises because
2 your expectations about the type of the initializing expression are incorrect, e.g.,
3 you believe that `std::vector<int>::size_type` is the same as `unsigned`, or
4 you forget that the first component of the `std::pairs` in a `std::unordered_map`
5 is `const`. If you use `auto` as the type of the target variable, you need not worry
6 about mismatches between the type of variable you're declaring and the type of
7 the expression used to initialize it.

8 There are thus several reasons to prefer `auto` over explicit type declarations. Yet
9 `auto` isn't perfect. The type for each `auto` variable is deduced from its initializing
10 expression, and some initializing expressions have types that are neither antici-
11 pated nor desired. The conditions under which such cases arise, and what you can
12 do about them, are discussed in Items 2 and 6, so I won't address them here. In-
13 stead, I'll turn my attention to a different concern you may have about using `auto`
14 in place of traditional type declarations: the readability of the resulting source
15 code.

16 First, take a deep breath and relax. `auto` is an option, not a mandate. If, in your
17 professional judgment, your code will be clearer or more maintainable or in some
18 other way better by using explicit type declarations, you're free to continue using
19 them. But bear in mind that C++ is far from breaking new ground in adopting what
20 is generally known in the programming languages world as *type inference*. Other
21 statically typed procedural languages (e.g., C#, D, Scala, Visual Basic) have a more
22 or less equivalent feature, to say nothing of a variety of statically typed functional
23 languages (e.g., ML, Haskell, OCaml, F#, etc.). In part, this is due to the success of
24 dynamically typed languages such as Perl, Python, and Ruby, where variables are
25 essentially never explicitly typed. The software development community has ex-
26 tensive experience with type inference, and it has demonstrated that there is noth-
27 ing contradictory about such technology and the creation and maintenance of in-
28 dustrial-strength, industrial-sized code bases.

29 Some developers are disturbed by the fact that using `auto` eliminates the ability to
30 determine an object's type by a quick glance at the source code. However, IDEs'
31 ability to show object types often mitigates this problem (even taking into account
32 the IDE type-display issues mentioned in Item 4), and, in many cases, a somewhat

1 abstract view of an object's type is just as useful as the exact type. It often suffices,
2 for example, to know that an object is a container or a counter or a smart pointer,
3 without knowing exactly what kind of container, counter, or smart pointer it is.
4 Assuming well-chosen variable names, such abstract type information should al-
5 most always be at hand.

6 The fact of the matter is that writing out types explicitly often does little more than
7 introduce opportunities for subtle errors, either in correctness or efficiency or
8 both. Furthermore, `auto` types automatically change if the type of their initializing
9 expression changes, and that means that some refactorings are facilitated by the
10 use of `auto`. For example, if a function is declared to return an `int`, but you later
11 decide that a `long` would be better, the calling code automatically updates itself
12 the next time you compile if the results of calling the function are stored in an `au-`
13 `to` variable. If the results are stored in a variable explicitly declared to be an `int`,
14 you'll need to find all the call sites so that you can revise them.

15 By all means, consider the impact on your source code's readability when choosing
16 between declaring `auto` types and explicit types. But before you reject `auto`, be
17 sure you're doing it for a solid technical reason, not simply because you're used to
18 writing out types by hand.

19 **Things to Remember:**

- 20 • `auto` variables must be initialized, are generally immune to type mismatches
21 that can lead to portability or efficiency problems, can ease the process of re-
22 factoring, and typically require less typing than variables with explicitly speci-
23 fied types.
- 24 • `auto`-typed variables are subject to the pitfalls described in Items 2 and 6.

25 **Item 6: Be aware of the typed initializer idiom.**

26 Item 5 explains that using `auto` to declare variables offers a number of technical
27 advantages over explicitly specifying types, but sometimes `auto`'s type deduction
28 zigs when you want it to zag. For example, suppose I have a function that takes a
29 `Widget` and returns a `std::vector<bool>`, where each `bool` indicates whether
30 the `Widget` offers a particular feature:

1 `std::vector<bool> features(const Widget& w);`
2 Further suppose that bit 5 indicates whether the `Widget` has high priority. We can
3 thus write code like this:

4 `Widget w;`
5 `...`
6 `bool highPriority = features(w)[5]; // is w high priority?`
7 `...`
8 `processWidget(w, highPriority); // process w in accord`
9 `// with its priority`

10 There's nothing wrong with this code. It'll work fine. But if we make the seemingly
11 innocuous change of replacing the explicit type for `highPriority` with `auto`,

12 `auto highPriority = features(w)[5]; // is w high priority?`
13 the situation changes. All the code will continue to compile, but its behavior is no
14 longer predictable:

15 `processWidget(w, highPriority); // undefined behavior!`

16 As the comment indicates, the call to `processWidget` now has undefined behav-
17 ior. But why? The answer is likely to be surprising. In the code using `auto`, the type
18 of `highPriority` is no longer `bool`. Though `std::vector<bool>` conceptually
19 holds `bools`, `operator[]` for `std::vector<bool>` doesn't return a reference to
20 an element of the container (which is what `std::vector::operator[]` returns
21 for every type *except* `bool`). Instead, it returns an object of type
22 `std::vector<bool>::reference` (a class nested inside `std::vector<bool>`).

23 `std::vector<bool>::reference` exists because `std::vector<bool>` is speci-
24 fied to represent its `bools` in packed form, one bit per `bool`. That creates a prob-
25 lem for `std::vector<bool>`'s `operator[]`, because `operator[]` for
26 `std::vector<T>` is supposed to return a `T&`, but C++ forbids references to bits.
27 Not being able to return a `bool&`, `operator[]` for `std::vector<bool>` returns
28 an object that *acts like* a `bool&`. For this act to succeed,
29 `std::vector<bool>::reference` objects must be usable in essentially all con-
30 texts where `bool&s` can be. Among the features in
31 `std::vector<bool>::reference` that make this work is an implicit conversion

1 to `bool`. (Not to `bool&`, to `bool`. To explain the full set of techniques used by
2 `std::vector<bool>::reference` to emulate the behavior of a `bool&` would
3 take us too far afield, so I'll simply remark that this implicit conversion is only one
4 stone in a larger mosaic.)

5 With this information in mind, look again at this part of the original code:

6 `bool highPriority = features(w)[5]; // declare highPriority's`
7 `// type explicitly`

8 Here, `features` returns a `std::vector<bool>` object, on which `operator[]` is
9 invoked. `operator[]` returns a `std::vector<bool>::reference` object, which
10 is then implicitly converted to the `bool` that is needed to initialize `highPriority`.
11 `highPriority` thus ends up with the value of bit 5 in the `std::vector<bool>`
12 returned by `features`, just like it's supposed to.

13 Contrast that with what happens in the `auto`-ized declaration for `highPriority`:

14 `auto highPriority = features(w)[5]; // deduce highPriority's`
15 `// type`

16 Again, `features` returns a `std::vector<bool>` object, and, again, `operator[]`
17 is invoked on it. `operator[]` continues to return a
18 `std::vector<bool>::reference` object, but now there's a change, because `au-`
19 `to` deduces that as the type of `highPriority`. `highPriority` doesn't have the
20 value of bit 5 of the `std::vector<bool>` returned by `features` at all.

21 The value it does have depends on how `std::vector<bool>::reference` is im-
22 plemented. One implementation is for such objects to contain a pointer to the ma-
23 chine word holding the referenced bit, plus the offset into that word for that bit.
24 Consider what that means for the initialization of `highPriority`, assuming that
25 such a `std::vector<bool>::reference` implementation is in place.

26 The call to `features` returns a temporary `std::vector<bool>` object. This ob-
27 ject has no name, but for purposes of this discussion, I'll call it `temp`. `operator[]`
28 is invoked on `temp`, and the `std::vector<bool>::reference` it returns con-
29 tains a pointer to a word in the data structure holding the bits that are managed by
30 `temp`, plus the offset into that word corresponding to bit 5. `highPriority` is a

1 copy of this `std::vector<bool>::reference` object, so `highPriority`, too,
2 contains a pointer to a word in `temp`, plus the offset corresponding to bit 5. At the
3 end of the statement, `temp` is destroyed, because it's a temporary object. There-
4 fore, `highPriority` contains a dangling pointer, and that's the cause of the unde-
5 fined behavior in the call to `processWidget`:

```
6 processWidget(w, highPriority);      // undefined behavior!
7                                     // highPriority may contain
8                                     // dangling pointer!
```

9 `std::vector<bool>::reference` is an example of a *proxy class*: a class that ex-
10 ists for the purpose of emulating and augmenting the behavior of some other type.
11 Proxy classes are employed for a variety of purposes.
12 `std::vector<bool>::reference` exists to offer the illusion that `operator[]`
13 for `std::vector<bool>` returns a reference to a bit, for example, and the Stand-
14 ard Library's smart pointer types (see Chapter 4) are proxy classes that graft re-
15 source management onto raw pointers. The utility of proxy classes is well-
16 established. In fact, the design pattern "Proxy" is one of the most longstanding
17 members of the software design patterns Pantheon.

18 Some proxy classes are designed to be apparent to clients. That's the case for
19 `std::shared_ptr` and `std::unique_ptr`, for example. Other proxy classes are
20 designed to act more or less invisibly. `std::vector<bool>::reference` is an
21 example of such "invisible" proxies, as is its C++14 cousin,
22 `std::bitset::reference`.

23 Also in that camp are some classes in C++ libraries employing a technique known
24 as *expression templates*. Such libraries were originally developed to improve the
25 efficiency of numeric code. Given a class `Matrix` and `Matrix` objects `m1`, `m2`, `m3`,
26 and `m4`, for example, the expression

```
27 Matrix sum = m1 + m2 + m3 + m4;
```

28 can be computed much more efficiently if `operator+` for `Matrix` objects returns a
29 proxy for the result instead of the result itself. That is, `operator+` for two `Matrix`
30 objects would return an object of a proxy class such as `Sum<Matrix, Matrix>` in-
31 stead of a `Matrix` object. As was the case with `std::vector<bool>::reference`

1 and `bool`, there'd be an implicit conversion from the proxy class to `Matrix`, which
2 would permit the initialization of `sum` from the proxy object produced by the ex-
3 pression on the right side of the “`=`”. (The type of that object would traditionally
4 encode the entire initialization expression, i.e., be something like
5 `Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>`. That's definitely a type
6 from which clients should be shielded.)

7 As a general rule, “invisible” proxy classes don't play well with `auto`. Objects of
8 such classes are often not designed to live longer than a single statement, so creat-
9 ing variables of those types tends to violate fundamental library design assump-
10 tions. That's the case with `std::vector<bool>::reference`, and we've seen
11 that violating that assumption can lead to undefined behavior.

12 You therefore want to avoid code of this form:

13 `auto someVar = expression of "invisible" proxy class type;`

14 But how can you recognize when proxy objects are in use? The software employ-
15 ing them is unlikely to advertise their existence. They're supposed to be *invisible*,
16 at least conceptually! And once you've found them, do you really have to abandon
17 `auto` and the many advantages Item 5 demonstrates for it?

18 Let's take the how-do-you-find-them question first. Although “invisible” proxy
19 classes are designed to fly beneath programmer radar in day-to-day use, libraries
20 using them often document that they do so. The more you've familiarized yourself
21 with the basic design decisions of the libraries you use, the less likely you are to be
22 blindsided by proxy usage within those libraries.

23 Where documentation comes up short, header files fill the gap. It's rarely possible
24 for source code to fully cloak proxy objects. They're typically returned from func-
25 tions that clients are expected to call, so function signatures usually reflect their
26 existence. Here's the spec for `std::vector<bool>::operator[]`, for example:

27 `namespace std {` // from C++98/11/14 Standards
28 `template <class Allocator>`
29 `class vector<bool, Allocator> {`
30 `public:`

```
1     ...
2     class reference { ... };
3 
4     reference operator[](size_type n);
5     ...
6 }
```

7 Assuming you know that `operator[]` for `std::vector<T>` normally returns a
8 `T&`, the unconventional return type for `operator[]` in this case is a tip-off that a
9 proxy class is in use. Paying careful attention to the interfaces you're using can of-
10 ten reveal the existence of proxy classes.

11 In practice, many developers discover the use of proxy classes only when they try
12 to track down mystifying compilation problems or debug incorrect unit test re-
13 sults. Regardless of how you find them, once `auto` has been determined to be de-
14 ducing the type of a proxy class instead of the type being proxied, the solution
15 need not involve abandoning `auto`. `auto` itself isn't the problem. The problem is
16 that `auto` isn't deducing the type you want it to deduce. The solution is to force a
17 different type deduction. The way you do that is what I call *the typed initializer id-
18 om*.

19 The typed initializer idiom involves declaring a variable with `auto`, but casting the
20 initialization expression to the type you want `auto` to deduce. Here's how it can be
21 used to force `highPriority` to be a `bool`, for example:

```
22 auto highPriority = static_cast<bool>(features(w)[5]);
```

23 Here, `features(w)[5]` continues to return a `std::vector<bool>::reference`
24 object, just as it always has, but the cast changes the type of the expression to
25 `bool`, which `auto` then deduces as the type for `highPriority`. At runtime, the
26 `std::vector<bool>::reference` object returned from
27 `std::vector<bool>::operator[]` executes the conversion to `bool` that it sup-
28 ports, and as part of that conversion, the still-valid pointer to the
29 `std::vector<bool>` returned from `features` is dereferenced. That avoids the
30 undefined behavior we ran into earlier. The index 5 is then applied to the bits
31 pointed to by the pointer, and the `bool` value that emerges is used to initialize
32 `highPriority`.

1 For the `Matrix` example, the typed initializer idiom would look like this:

2 `auto sum = static_cast<Matrix>(m1 + m2 + m3 + m4);`

3 Applications of the idiom aren't limited to initializers yielding proxy class types. It
4 can also be useful to emphasize that you are deliberately creating a variable of a
5 type that is different from that generated by the initializing expression. For exam-
6 ple, suppose you have a function to calculate some tolerance value:

7 `double calcEpsilon(); // return tolerance value`

8 `calcEpsilon` clearly returns a `double`, but suppose you know that for your appli-
9 cation, the precision of a `float` is adequate, and you care about the difference in
10 size between `floats` and `doubles`. You could declare a `float` variable to store the
11 result of `calcEpsilon`,

12 `float ep = calcEpsilon(); // implicitly convert`
13 `// double→float`

14 but this hardly announces "I'm deliberately reducing the precision of the value re-
15 turned by the function!" A declaration using the typed initializer idiom, however,
16 does:

17 `auto ep = static_cast<float>(calcEpsilon());`

18 Similar reasoning applies if you have a floating point expression that you are de-
19 liberately storing as an integral value. Suppose you need to calculate the index of
20 an element in a container with random access iterators (e.g., a `std::vector`,
21 `std::deque`, or `std::array`), and you're given a `double` between 0.0 and 1.0
22 indicating how far from the beginning of the container the desired element is lo-
23 cated. (0.5 would indicate the middle of the container.) Further suppose that
24 you're confident that the resulting index will fit in an `int`. If the container is `c` and
25 the double is `d`, you could calculate the index this way,

26 `int index = d * c.size();`

27 but this obscures the fact that you're intentionally converting the `double` on the
28 right to an `int`. The typed initializer idiom eliminates makes things transparent:

29 `auto index = static_cast<int>(d * c.size());`

1 **Things to Remember**

- 2 ♦ “Invisible” proxy types can cause `auto` to deduce the “wrong” type for an ini-
3 tializing expression.
- 4 ♦ The typed initializer idiom forces `auto` to deduce the type you want it to have.

1 **Chapter 3 From C++98 to C++11 and C++14**

2 If you're an experienced C++ developer, you understand and are comfortable with
3 C++98 and its common programming idioms. Enums, classes, member functions,
4 inheritance, templates, `const`, using objects to manage resources—you've been
5 there, you've done that. As such, you're likely to view C++11 as C++98 plus a bunch
6 of new features, with C++14 comprising essentially more of the same.

7 For the most part, that's fine. C++'s legendary commitment to backwards-
8 compatibility means that not only does virtually all legacy source code remain val-
9 id, the same is true for most C++ software development techniques. Good practices
10 in C++98 typically remain good practices in C++11. But there are exceptions. In
11 some cases, C++98 language features should generally be eschewed in favor of
12 newer, similar C++11 features that take advantage of the experience the C++ pro-
13 gramming community garnered during the 13 year reign of C++98. Examples in-
14 clude using `nullptr` instead of `NULL` or `0` (Item 8), preferring alias declarations
15 over `typedefs` (Item 9), choosing scoped instead of unscoped `enums` (Item 10),
16 and prohibiting the use of unwanted member functions by `deleteing` them in-
17 stead of declaring them `private` (Item 11). In other cases, taking advantage of
18 new language features requires modification of some well-entrenched habits. For
19 example, virtual function overrides should now be declared as such (Item 12),
20 some parameters are now better passed by value than by reference-to-`const`
21 (Item 17), calls to `push_back` or `push_front` or `insert` should sometimes be re-
22 placed with their emplacement counterparts (Item 18), functions that can poten-
23 tially run during compilation should be declared `constexpr` (Item 14), and excep-
24 tion specifications should be more widely employed (Item 16). In still other cases,
25 C++11 simply changes the rules of the C++ programming game. Ways this is ap-
26 parent include that there's now an important syntactic choice to be made when
27 declaring objects (Item 7), `const_iterators` are substantially more useful than
28 they used to be (Item 13), declaring member functions `const` implies that they're
29 thread-safe (Item 15), and compiler-generated member functions are governed by
30 a set of rules that have been significantly revised (Item 19).

1 This chapter covers guidelines that will help you update C++98 habits to bring
2 them into accord with C++11 and C++14, but not all guidelines of that nature are in
3 this chapter. Item 5, for example (“Prefer `auto` to explicit type declarations”), be-
4 longs here, too, but it fits more naturally into the chapter on `auto`, and the over-
5 arching guidance to prefer smart pointers to raw pointers has so many facets, it
6 gets its own chapter (Chapter 4). Still, the Items that follow do a good job of sum-
7 marizing how effective programming in C++11 and C++14 distinguishes itself from
8 C++98 approaches to accomplishing common development tasks. It demonstrates
9 that being an effective contemporary C++ programmer consists of not just adopt-
10 ing new language features, but also reevaluating techniques and practices proven
11 in C++98.

12 **Item 7: Distinguish () and {} when creating objects.**

13 Depending on your perspective, syntax choices for object initialization in C++11
14 embody either an embarrassment of riches or a confusing mess. As a general rule,
15 initialization values may be specified with parentheses, an equals sign, or braces:

```
16 int x(0);           // initializer is in parentheses
17 int y = 0;          // initializer follows "="
18 int z{0};           // initializer is in braces
```

19 In many cases, it's also possible to use an equals sign and braces together:

```
20 int z = {0};        // initializer uses "=" and braces
```

21 For the remainder of this Item, I'll generally ignore the equals-sign-plus-braces
22 syntax, because C++ usually treats it the same as the braces-only version.

23 The “confusing mess” lobby points out that the use of an equals sign for initializa-
24 tion often misleads C++ newbies into thinking that an assignment is taking place,
25 even though it's not. For built-in types like `int`, the difference is academic, but for
26 user-defined types, it's important to distinguish initialization from assignment,
27 because different function calls are involved:

```
28 Widget w1;          // call default constructor
29 Widget w2 = w1;      // not an assignment; calls copy ctor
```

```
1 w1 = w2;           // an assignment; calls copy operator=
2 Even with several initialization syntaxes, there were some situations where C++98
3 had no way to express a desired initialization. For example, it wasn't possible to
4 directly indicate that an STL container (e.g., std::vector<int>) should be created
5 holding a particular set of values (e.g., 1, 3, and 5).
6 To address the confusion of multiple initialization syntaxes, as well as the fact that
7 they don't cover all initialization scenarios, C++11 introduces uniform initialization
8: a single initialization syntax that can be used anywhere and can express everything. It's based on braces, and for that reason I prefer the term braced initialization.
9 "Uniform initialization" is a concept. "Braced initialization" is a syntactic construct.
```

```
12 Braced initialization lets you express the formerly inexpressible. Using braces,
13 specifying the initial contents of a container is easy:
```

```
14 std::vector<int> v{1, 3, 5};    // v's initial content is 1, 3, 5
15 Braces can also be used to specify default initialization values for non-static data
16 members. This capability—new to C++11—is shared with the “=” initialization
17 syntax, but not with parentheses:
```

```
18 class Widget {
19 ...
20 private:
21     int x{0};          // fine, x's default value is 0
22     int y = 0;         // also fine
23     int z(0);          // error!
24 };
```

```
25 On the other hand, uncopyable objects (e.g., std::atomics) may be initialized using braces or parentheses, but not using “=”:
```

```
27 std::atomic<int> ai1{0};      // fine
28 std::atomic<int> ai2(0);      // fine
29 std::atomic<int> ai3 = 0;      // error!
```

1 Perhaps now you see why braced initialization is called “uniform.” Of C++’s three
2 ways to designate an initializing expression (braces, parentheses, and “=”), only
3 braces can be used everywhere.

4 A novel feature of braced initialization is that it prohibits implicit *narrowing conversions* among built-in types. If the value of an expression in a braced initializer
5 isn’t guaranteed to be expressible in the type of the object being initialized, the
6 code won’t compile:

```
8 double x, y, z;  
9 ...  
10 int sum1{x + y + z};           // error! sum of doubles may  
11                                // not be expressible as int
```

12 Initialization using parentheses and “=” doesn’t check for narrowing conversions,
13 because that could break too much legacy code:

```
14 int sum2(x + y + z);          // okay (value of expression  
15                                // truncated to an int)  
16 int sum3 = x + y + z;         // ditto
```

17 Another noteworthy characteristic of braced initialization is its immunity to C++’s
18 *most vexing parse*. A side-effect of C++’s rule that anything that can be parsed as a
19 declaration must be interpreted as one, the most vexing parse most frequently af-
20 flicts developers when they want to default-construct an object, but inadvertently
21 end up declaring a function, instead. The root of the problem is that if you want to
22 call a constructor with an argument, you can do it like this,

```
23 Widget w1(10);    // call Widget ctor with argument 10  
24 but if you try to call a Widget constructor with zero arguments using the analo-  
25 gous syntax, you declare a function instead of an object:
```

```
26 Widget w2();        // most vexing parse! declares a function  
27                                // named w2 that returns a Widget!
```

28 This trap is particularly odious, because an empty set of parentheses sometimes
29 *does* call a constructor with zero arguments:

```
1 void f(const Widget& w = Widget()); // w's default value is a
2 // default-constructed
3 // Widget
4 Braced initialization eliminates the most vexing parse, yet has no effect on the
5 meaning of initializations that already do what's desired:
```

```
6 Widget w1{10}; // as before, calls Widget ctor with arg 10
7 Widget w2{}; // now calls Widget ctor with no args
8 void f(const Widget& w = Widget{}); // as before, w's default
9 // value is a default-
10 // constructed Widget
```

```
11 There's thus a lot to be said for braced initialization. It's the syntax that can be
12 used in the widest variety of contexts, it prevents implicit narrowing conversions,
13 and it's immune to C++'s most vexing parse. A trifecta of goodness, right? So why
14 isn't this Item entitled something like "Use braced initialization syntax"?
```

```
15 The drawback to braced initialization is the sometimes-surprising behavior that
16 accompanies it. Such behavior grows out of the unusually tangled relationship
17 among braced initializers, std::initializer_lists, and constructor overload
18 resolution. Their interactions can lead to code that seems like it should do one
19 thing, but actually does another. For example, Item 2 explains that when an auto-
20 declared variable has a braced initializer, the type deduced is
21 std::initializer_list, even though other ways of declaring a variable with
22 the same initializer would cause auto to deduce the type of the initializer:
```

```
23 auto v1 = -1; // -1's type is int, and so is v1's
24 auto v2(-1); // -1's type is int, and so is v2's
25 auto v3{-1}; // -1's type is still int, but
26 // v3's type is std::initializer_list<int>
27 auto v4 = {-1}; // -1's type remains int, but
28 // v4's type is std::initializer_list<int>
```

```
29 In constructor calls, parentheses and braces have the same meaning as long as
30 std::initializer_list parameters are not involved:
```

```
31 class Widget {
32 public:
```

```
1   Widget(int i, bool b);      // ctors not declaring
2   Widget(int i, double d);    // std::initializer_list params
3 ...
4 }

5 Widget w1(10, true);          // calls first ctor
6 Widget w2{10, true};          // also calls first ctor
7 Widget w3(10, 5.0);           // calls second ctor
8 Widget w4{10, 5.0};           // also calls second ctor

9 If, however, one or more constructors declares a parameter of type
10 std::initializer_list, calls using the braced initialization syntax strongly
11 prefer the overloads taking std::initializer_lists. Strongly. If there's any
12 way for compilers to construe a call using a braced initializer to be to a constructor
13 taking a std::initializer_list, compilers will employ that interpretation. If
14 the Widget class above is augmented with a constructor taking a
15 std::initializer_list<long double>, for example,
```

```
16 class Widget {
17 public:
18   Widget(int i, bool b);                  // as before
19   Widget(int i, double d);                // as before
20   Widget(std::initializer_list<long double> il); // added
```

```
21 ...
22 }
```

23 Widgets w2 and w4 will be constructed using the new constructor, even though the
24 type of the std::initializer_list elements (long double) is, compared to
25 the non-std::initializer_list constructors, a worse match for both argu-
26 ments!

```
27 Widget w1(10, true);      // uses parens and, as before,
28                           // calls first ctor
29 Widget w2{10, true};       // uses braces, but now calls
30                           // std::init_List ctor (10 and
31                           // true convert to long double)
32 Widget w3(10, 5.0);        // uses parens and, as before,
33                           // calls second ctor
```

```
1 Widget w4{10, 5.0};           // uses braces, but now calls
2                                     // std::init_list ctor (10 and
3                                     // 5.0 convert to long double)
4 Compilers' determination to match braced initializers with constructors taking
5 std::initializer_lists is so strong, it prevails even if the best-match
6 std::initializer_list constructor can't be called. For example, consider this
7 slightly-revised example:
```

```
8 class Widget {
9 public:
10    Widget(int i, bool b);          // as before
11    Widget(int i, double d);        // as before
12    Widget(std::initializer_list<bool> il); // std::init_list
13    ...
14 };
15 Widget w{10, 5.0};      // error! requires narrowing conversions
```

16 Here, compilers will ignore the first two constructors (the second of which offers
17 an exact match on both argument types) and try to call the constructor taking a
18 std::initializer_list<bool>. Calling that constructor would require con-
19 verting an int (10) and a double (5.0) to bools. Both conversions would be nar-
20 rowing (bool can't exactly represent either value), and narrowing conversions are
21 prohibited inside braced initializers, so the call is invalid, and the code is rejected.

22 If there's no way to convert the types of the arguments in a braced initializer to the
23 type taken by a std::initializer_list, compilers fall back on normal overload
24 resolution. For example, if we replace the std::initializer_list<bool> con-
25 structor with one taking a std::initializer_list<std::string>, the non-
26 std::initializer_list constructors become candidates again, because there
27 is no way to convert ints and bools to std::strings:

```
28 class Widget {
29 public:
30    Widget(int i, bool b);          // as before
31    Widget(int i, double d);        // as before
32    // std::init_list element type is now std::string
33    Widget(std::initializer_list<std::string> il);
34    ...
35 };
```

```
1 Widget w1(10, true);      // uses parens, still calls first ctor
2 Widget w2{10, true};      // uses braces, now calls first ctor
3 Widget w3(10, 5.0);       // uses parens, still calls second ctor
4 Widget w4{10, 5.0};       // uses braces, now calls second ctor
5 There are two additional twists to the tale of constructor overload resolution and
6 braced initializers that are worth knowing about:
```

- ```
7 • Empty braces mean no arguments, not an empty std::initializer_list. Speci-
8 fying constructor arguments with an empty pair of braces is a request to call
9 the default constructor, not a request to call a constructor with an empty
10 std::initializer_list:
```

```
11 class Widget {
12 public:
13 Widget(); // default ctor
14 Widget(std::initializer_list<int> il); // std::init_list
15 ...
16 };
17 Widget w1; // calls default ctor
18 Widget w2{}; // also calls default ctor
19 // (doesn't create empty std::init_list)
20 Widget w3(); // most vexing parse! declares a function!
21 If you want to call a std::initializer_list constructor with an empty
22 std::initializer_list, you do it by making the empty braces a construc-
23 tor argument—by putting the empty braces inside the parentheses or braces
24 demarcating what you're passing!
```

```
25 Widget w4({}); // calls std::init_list ctor
26 // with empty list
27 Widget w5{{}}; // ditto
```

- ```
28 • Copy and move constructors are called as usual. Creating an object from
29 another object of the same type always invokes the conventional copying and
30 moving functions:
```

```
31 class Widget {
32 public:
```

```
1     Widget(const Widget& rhs);           // copy ctor
2     Widget(Widget&& rhs);               // move ctor
3
4     Widget(std::initializer_list<int> il); // std::init_list
5                               // ctor
6
7     operator int() const;                // convert to int
8     ...
9
10    };
11
12    auto w6{w5};                         // calls copy ctor, not
13                               // std::init_list <int> ctor, even
14                               // though Widget converts to int
15
16    auto w7{std::move(w5)};               // ditto, but for move ctor
17                               // (Item 25 has info on std::move)
```

13 At this point, with seemingly arcane rules about braced initializers,
14 `std::initializer_lists`, and constructor overloading burbling about in your
15 brain, you may be wondering how much of this information matters in day-to-day
16 programming. More than you might think. That's because one of the classes direct-
17 ly affected is `std::vector`. `std::vector` has a non-`std::initializer_list`
18 constructor that allows you to specify the initial size of the container and a value
19 each of the initial elements should have, but it also has a constructor taking a
20 `std::initializer_list` that permits you to specify the initial values in the con-
21 tainer. If you create a `std::vector` of a numeric type (e.g., a `std::vector<int>`)
22 and you pass two arguments to the constructor, whether you enclose those argu-
23 ments in parentheses or braces makes a tremendous difference:

```
24 std::vector<int> v1(10, 20); // use non-std::init_list ctor:
25                               // create 10-element std::vector,
26                               // all elements have value of 20
27 std::vector<int> v2{10, 20}; // use std::init_list ctor:
28                               // create 2-element std::vector,
29                               // element values are 10 and 20
```

30 But let's step back from `std::vector` and also from the details of parentheses,
31 braces, and constructor overloading resolution rules. There are two primary take-
32 aways from this discussion. First, as a class author, you need to be aware that if
33 your constructor overloads include one or more functions taking a
34 `std::initializer_list`, client code using braced initialization may see only the
35 `std::initializer_list` overloads. As a result, it's best to design your construc-

1 tors so that the overload called isn't affected by whether clients use parentheses or
2 braces. In other words, learn from what is now considered to be an error in the
3 design of the `std::vector` interface, and design your classes to avoid it.

4 An implication is that if you have a class with no `std::initializer_list` con-
5 structor and you add one, client code using braced initialization may find that calls
6 that used to resolve to non-`std::initializer_list` constructors now resolve
7 to the new function. Of course, this kind of thing can happen any time you add a
8 new function to a set of overloads: calls that used to resolve to one of the old over-
9 loads might start calling the new one. The difference with
10 `std::initializer_list` constructor overloads is that a
11 `std::initializer_list` overload doesn't just compete with other overloads, it
12 overshadows them to the point that the other overloads may not even be consid-
13 ered. So add such overloads only with great deliberation.

14 The second lesson is that as a class client, you must choose carefully between pa-
15 rentheses and braces when creating objects. Most developers end up choosing one
16 kind of delimiter as a default, using the other only when they have to. Braces-by-
17 default folks are attracted by their unrivaled breadth of applicability, their preven-
18 tion of narrowing conversions, and their immunity to C++'s most vexing parse.
19 Such folks understand that in some cases (e.g., creation of a `std::vector` with a
20 given size and initial element value), parentheses are required. In contrast, the go-
21 parentheses-go crowd embraces parentheses as their default argument delimiter.
22 They're attracted to its consistency with the C++98 syntactic tradition, its avoid-
23 ance of the auto-deduced-a-`std::initializer_list` problem, and the
24 knowledge that their object creation calls won't be inadvertently waylaid by
25 `std::initializer_list` constructors. They concede that sometimes only brac-
26 es will do (e.g., when creating a container with particular values). Neither ap-
27 proach is rigorously better than the other. My advice is to pick one and apply it
28 consistently.[†]

[†] The examples in this book reveal that I'm a parentheses-by-default person.

1 If you're a template author, the parentheses-braces duality for object creation can
2 be especially frustrating, because, in general, it's not possible to know which form
3 should be used. For example, suppose you'd like to create an object of an arbitrary
4 type from an arbitrary number of arguments. A variadic template makes this con-
5 ceptually straightforward:

```
6 template<typename T,           // type of object to create
7         typename... Args>      // types of arguments to use
8 void doSomeWork(const T& obj, Args&&... args)
9 {
10     create Local T object from args...
11     ...
12 }
```

13 There are two ways to turn the line of pseudocode into real code (see Item 27 for
14 information about `std::forward`):

```
15 T localObject(std::forward<Args>(args)...);    // using parens
16 T localObject{std::forward<Args>(args)...};    // using braces
```

17 So consider this calling code:

```
18 std::vector<int> v;
19 ...
20 doSomeWork(v, 10, 20);
```

21 If `doSomeWork` uses parentheses when creating `localObject`, the result is a
22 `std::vector` with 10 elements. If `doSomeWork` uses braces, the result is a
23 `std::vector` with 2 elements. Which is correct? The author of `doSomeWork` can't
24 know. Only the caller can.

25 This is precisely the problem faced by the Standard Library functions
26 `std::make_unique` and `std::make_shared` (see Item 23). These functions re-
27 solve the problem by internally using parentheses and documenting this decision
28 as part of their interfaces. This is not the only way of dealing with the issue, how-
29 ever. Alternative designs permit callers to determine whether parentheses or

1 braces should be used in functions generated from a template. A common compo-
2 nent of such designs is tag dispatch, which is described in Item 29.[†]

3 **Things to Remember**

- 4 ♦ Braced initialization is the most widely applicable initialization syntax, it pre-
5 vents narrowing conversions, and it's immune to C++'s most vexing parse.
- 6 ♦ As detailed in Item 2, braced initializers yield `std::initializer_lists` for
7 auto-declared objects.
- 8 ♦ During constructor overload resolution, braced initializers are matched to
9 `std::initializer_list` parameters, even if other constructors offer seem-
10 ingly better matches.
- 11 ♦ An example of where the choice between parentheses and braces can make a
12 significant difference is creating a `std::vector` with two arguments.
- 13 ♦ Choosing between parentheses and braces for object creation inside templates
14 can be challenging.

15 **Item 8: Prefer `nullptr` to `0` and `NULL`.**

16 So here's the deal: `0` is an `int`, not a pointer. If C++ finds itself looking at `0` in a con-
17 text where only a pointer can be used, it'll grudgingly interpret `0` as a null pointer,
18 but that's a fallback position. C++'s primary policy is that `0` is an `int`, not a pointer.

19 Practically speaking, the same is true of `NULL`. There is some uncertainty in the de-
20 tails in `NULL`'s case, because implementations are allowed to give `NULL` an integral
21 type other than `int` (e.g., `long`).^{*} That's not common, but it doesn't really matter,
22 because the issue here isn't the exact type of `NULL`, it's that neither `0` nor `NULL` has
23 a pointer type.

[†] The treatment in Item 29 is general. For an example of how it can be specifically applied to functions like `doSomeWork`, see the 5 June 2013 entry of *Andrzej's C++ blog*, “[Intuitive interface — Part I](#).”

* In C, `NULL`'s type can be `void*`, but this has never been legal in C++.

1 In C++98, the primary implication of this was that overloading on pointer and in-
2 tegral types could lead to surprises. Passing `0` or `NULL` to such overloads never
3 called a pointer overload:

```
4 void f(int);           // three overloads of f
5 void f(bool);
6 void f(void*);
```



```
7 f(0);                 // calls f(int) overload, not f(void*)
8 f(NULL);              // might not compile, but typically calls
9                  // f(int) overload. Never calls f(void*)
```

10 The uncertainty regarding the behavior of `f(NULL)` is a reflection of the leeway
11 granted to implementations regarding the type of `NULL`. If `NULL` is defined to be,
12 say, `0L` (i.e., `0` as a `long`), the call is ambiguous, because conversion from `long` to
13 `int`, `long` to `bool`, and `0L` to `void*` are considered equally good. The interesting
14 thing about that call is the contradiction between the *apparent* meaning of the
15 source code (“I’m calling `f` with `NULL`—the null pointer”) and its *actual* meaning
16 (“I’m calling `f` with some kind of integer—not the null pointer”). This counterintu-
17 itive behavior is what led to the guideline for C++98 programmers to avoid over-
18 loading on pointer and integral types. That guideline remains valid in C++11, be-
19 cause, the advice of this Item notwithstanding, it’s likely that some developers will
20 continue to use `0` and `NULL`, even though `nullptr` is a better choice.

21 `nullptr`’s advantage is that it doesn’t have an integral type. To be honest, it
22 doesn’t have a pointer type, either, but you can think of it as a pointer of *all* types.
23 `nullptr`’s actual type is `std::nullptr_t`, and, in a wonderfully circular defini-
24 tion, `std::nullptr_t` is defined to be the type of `nullptr`. The type
25 `std::nullptr_t` implicitly converts to all raw pointer types, and that’s what
26 makes `nullptr` act as if it were a pointer of all types.

27 Calling the overloaded function `f` with `nullptr` calls the `void*` overload (i.e., the
28 pointer overload), because `nullptr` can’t be viewed as anything integral:

```
29 f(nullptr);           // calls f(void*) overload
```

1 Using `nullptr` instead of `0` or `NULL` thus avoids overload resolution surprises, but
2 that's not its only advantage. It can also improve code clarity, especially when `auto`-
3 `to` variables are involved. For example, suppose you encounter this in a code base:

```
4 auto result = findRecord( /* arguments */ );  
5 if (result == 0) {  
6     ...  
7 }
```

8 If you don't happen to know (or be able to easily find out) what `findRecord` re-
9 turns, it may not be clear whether `result` is a pointer type or an integral type. Af-
10 ter all, `0` (what `result` is tested against) could go either way. If you see the follow-
11 ing, on the other hand,

```
12 auto result = findRecord( /* arguments */ );  
13 if (result == nullptr) {  
14     ...  
15 }
```

16 there's no ambiguity: `result` must be a pointer type.

17 Where `nullptr` truly shines is when templates enter the picture. Suppose you
18 have some functions that should be called only when the appropriate mutex has
19 been locked. Each function takes a different kind of pointer:

```
20 int    f1(std::shared_ptr<Widget> spw); // call these only when  
21 double f2(std::unique_ptr<Widget> upw); // the appropriate  
22 bool   f3(Widget* pw);                // mutex is locked
```

23 Calling code that wants to pass null pointers could look like this:

```
24 std::mutex f1m, f2m, f3m;           // mutexes for f1, f2, and f3  
25 using MuxGuard =                  // C++11 typedef; see Item 9  
26     std::lock_guard<std::mutex>;  
27 ...  
28 {  
29     MuxGuard g(f1m);             // lock mutex for f1  
30     auto result = f1(0);         // pass 0 as null ptr to f1  
31 }                                // unlock mutex  
32 ...
```

```

1  {
2      MuxGuard g(f2m);           // lock mutex for f2
3      auto result = f2(NULL);   // pass NULL as null ptr to f2
4  }                           // unlock mutex
5
6  ...
7  {
8      MuxGuard g(f3m);           // lock mutex for f3
9      auto result = f3(nullptr); // pass nullptr as null ptr to f3
10     }                          // unlock mutex

```

10 The failure to use `nullptr` in the first two calls in this code is sad, but the code
11 works, and that counts for something. However, the repeated pattern in the calling
12 code—lock mutex, call function, unlock mutex—is more than sad. It's disturbing.
13 This kind of source code duplication is one of the things that templates are de-
14 signed to avoid, so let's templatize the pattern:

```

15 template<typename FuncType,
16          typename MuxType,
17          typename PtrType>
18 auto lockAndCall(FuncType func,
19                  MuxType& mutex,
20                  PtrType ptr) -> decltype(func(ptr))
21 {
22     MuxGuard g(mutex);
23     return func(ptr);
24 }

```

25 If the return type of this function (`auto ... -> decltype(func(ptr))`) has you
26 scratching your head, do your head a favor and navigate to Item 3, which explains
27 what's going on. There you'll see that in C++14, the return type could be reduced
28 to a simple `decltype(auto)`:

```

29 template<typename FuncType,
30          typename MuxType,
31          typename PtrType>
32 decltype(auto) lockAndCall(FuncType func,           // C++14 only
33                           MuxType& mutex,
34                           PtrType ptr)
35 {
36     MuxGuard g(mutex);
37     return func(ptr);
38 }

```

39 Given the `lockAndCall` template (either version), callers can write code like this:

```
1 auto result1 = lockAndCall(f1, f1m, 0);           // error!
2 ...
3 auto result2 = lockAndCall(f2, f2m, NULL);        // error!
4 ...
5 auto result3 = lockAndCall(f3, f3m, nullptr);     // fine
```

6 Well, they can write it, but, as the comments indicate, in two of the three cases, the
7 code won't compile. The problem in the first call is that when `0` is passed to
8 `lockAndCall`, template type deduction kicks in to figure out its type. The type of `0`
9 is, was, and always will be `int`, so that's the type of the parameter `ptr` inside the
10 instantiation of this call to `lockAndCall`. Unfortunately, this means that in the call
11 to `func` inside `lockAndCall`, an `int` is being passed, and that's not compatible
12 with the `std::shared_ptr<Widget>` parameter that `f1` expects. The `0` passed in
13 the call to `lockAndCall` was intended to represent a null pointer, but what actual-
14 ly got passed was a run-of-the-mill `int`. Trying to pass this `int` to `f1` as a
15 `std::shared_ptr<Widget>` is a type error. The call to `lockAndCall` with `0`,
16 then, fails, because inside the template, an `int` is being passed to a function that
17 requires a `std::shared_ptr<Widget>`.

18 The analysis for the call involving `NULL` is essentially the same. When `NULL` is
19 passed to `lockAndCall`, an integral type is deduced for the parameter `ptr`, and a
20 type error occurs when `ptr`—an `int` or `int`-like type—is passed to `f2`, which ex-
21 pects to get a `std::unique_ptr<Widget>`.

22 In contrast, the call involving `nullptr` has no trouble. When `nullptr` is passed to
23 `lockAndCall`, the type for `ptr` is deduced to be `std::nullptr_t` (i.e., `nullptr`'s
24 type). When `ptr` is passed to `f3`, there's an implicit conversion from
25 `std::nullptr_t` to `void*`, because `std::nullptr_t` objects implicitly convert
26 to all pointer types. (If the idea of there being objects—*plural*—of type
27 `std::nullptr_t` seems odd, I agree, it is. But don't worry about it. All objects of
28 type `std::nullptr_t` represent the null pointer, so they all behave the same way.
29 You can create your own such objects if you want to, but given the universal avail-
30 ability of `nullptr`, why would you?)

1 The fact that template type deduction deduces the “wrong” types for `0` and `NULL`
2 (i.e., their true types, rather than their fallback meaning as a representation for a
3 null pointer) is the most compelling reason to use `nullptr` instead of `0` or `NULL`
4 when you want to refer to a null pointer. With `nullptr`, templates pose no special
5 challenge. With `0` and `NULL`, the challenge is essentially insurmountable: templates
6 always deduce the wrong type. Combined with the fact that `nullptr` doesn’t suffer
7 from the overload resolution surprises that `0` and `NULL` are susceptible to, the case
8 is iron-clad. When you want to refer to a null pointer, use `nullptr`, not `0` or `NULL`.

9 As a coda, it’s worth noting that `lockAndCall` is a simplified version of a perfect
10 forwarding template, and the inability to forward `0` and `NULL` as null pointers is an
11 example of a situation where perfect forwarding doesn’t work. For details on per-
12 fect forwarding and the conditions under which it breaks down, consult Item 32.

13 **Things to Remember**

- 14 • Prefer `nullptr` to `0` and `NULL`.
15 • Avoid overloading on integral and pointer types.

16 **Item 9: Prefer alias declarations to `typedefs`.**

17 I’m confident we can agree that using STL containers is a good idea, and I hope that
18 Item 20 convinces you that using `std::unique_ptr` is a good idea, but my guess
19 is that neither of us is fond of writing types like
20 “`std::unique_ptr<std::unordered_map<std::string, std::string>>`”
21 more than once. Just thinking about it probably increases the risk of carpal tunnel
22 syndrome.

23 Avoiding such medical tragedies is easy. Just introduce a `typedef`:

```
24 typedef  
25   std::unique_ptr<std::unordered_map<std::string, std::string>>  
26   UPtrMapSS;
```

27 But `typedefs` are so...C++98. They work in C++11, sure, but C++11 also offers *ali-*
28 *as declarations*:

```
1 using UPtrMapSS =  
2     std::unique_ptr<std::unordered_map<std::string, std::string>>;  
3 Given that the typedef and the alias declaration do exactly the same thing (in the  
4 Standard, an alias declaration is actually defined to be an alternative way to create  
5 a typedef), it's reasonable to wonder whether there is a solid technical reason for  
6 preferring one over the other.
```

7 There is, but before I get to it, I want to mention that many people find the alias
8 declaration easier to swallow when dealing with types involving function pointers:

```
9 // FP is a synonym for a pointer to a function taking an int and  
10 // a const std::string& and returning nothing  
11 typedef void (*FP)(int, const std::string&);           // typedef  
12 // same meaning as above  
13 using FP = void (*)(int, const std::string&);           // alias  
14                                         // declaration
```

15 Of course, neither form is particularly easy to choke down, and few people spend
16 much time dealing with synonyms for function pointer types, anyway, so this is
17 hardly a compelling reason to choose alias declarations over **typedefs**.

18 But a compelling reason does exist: templates. In particular, alias declarations may
19 be templatized (in which case they're called *alias templates*), while **typedefs** can-
20 not. This gives C++11 programmers a clear, straightforward mechanism for ex-
21 pressing things that in C++98 had to be hacked together with **typedefs** nested
22 inside templatized **structs**. For example, consider defining a synonym for a linked
23 list that uses a custom allocator, **MyAlloc**. With an alias template, it's a piece of
24 cake:

```
25 template<typename T>                                // MyAllocList<T>  
26 using MyAllocList = std::list<T, MyAlloc<T>>;    // is synonym for  
27                                         // std::list<T,  
28                                         //   MyAlloc<T>>  
29 MyAllocList<Widget> lw;                           // client code
```

30 With a **typedef**, you pretty much have to create the cake from scratch:

```
31 template<typename T>                                // MyAllocList<T>::type  
32 struct MyAllocList {                                // is synonym for
```

```
1     typedef std::list<T, MyAlloc<T>> type; // std::list<T,  
2 }; // MyAlloc<T>>  
3 MyAllocList<Widget>::type lw; // client code  
4 It gets worse. If you want to use the typedef inside a template for the purpose of  
5 creating a linked list holding objects of a type specified by a template parameter,  
6 you have to precede the typedef name with typename:
```

```
7 template<typename T>  
8 class Widget { // Widget<T> contains  
9 private: // a MyAllocList<T>  
10 typename MyAllocList<T>::type list; // as a data member  
11 ...  
12 };
```

13 Here, `MyAllocList<T>::type` refers to a type that's dependent on a template
14 type parameter (`T`). `MyAllocList<T>::type` is thus a *dependent type*, and one of
15 C++'s many endearing rules is that the names of dependent types must be preceded
16 by `typename`.

17 If `MyAllocList` is defined as an alias template, this need for `typename` vanishes
18 (as does the cumbersome “`::type`” suffix):

```
19 template<typename T>  
20 using MyAllocList = std::list<T, MyAlloc<T>>; // as before  
21 template<typename T>  
22 class Widget {  
23 private:  
24     MyAllocList<T> list; // no "typename",  
25     ... // no "::type"  
26 };
```

27 To you, `MyAllocList<T>` (i.e., use of the alias template) may look just as dependent
28 on the template parameter `T` as `MyAllocList<T>::type` (i.e., use of the
29 nested `typedef`), but you're not a compiler. When compilers process the `Widget`
30 template and encounter the use of `MyAllocList<T>` (i.e. use of the alias tem-
31 plate), they know that `MyAllocList<T>` is the name of a type, because `MyAlloc-`
32 `list` is an alias template: it *must* name a type. `MyAllocList<T>` is thus a *non-*
33 *dependent type*, and a `typename` specifier is neither required nor permitted.

1 When compilers see `MyAllocList<T>::type` (i.e., use of the nested `typedef`) in
2 the `Widget` template, on the other hand, they can't know for sure that it names a
3 type, because there might be a specialization of `MyAllocList` that they haven't yet
4 seen where `MyAllocList<T>::type` refers to something other than a type. That
5 sounds crazy, but don't blame compilers for this possibility. It's the humans who
6 have been known to produce such code.

7 For example, some misguided soul may have concocted something like this:

```
8 class Wine { ... };

9 template<>
10 class MyAllocList<Wine> {           // MyAllocList specialization
11 private:
12     enum class WineType             // see Item 10 for info on
13     { White, Red, Rose };          // "enum class"

14     WineType type;                // in this class, type is
15     ...
16 };
```

17 As you can see, `MyAllocList<Wine>::type` doesn't refer to a type. If `Widget`
18 were to be instantiated with `Wine`, `MyAllocList<T>::type` inside the `Widget`
19 template would refer to a data member, not a type. Inside the `Widget` template,
20 then, whether `MyAllocList<T>::type` refers to a type is honestly dependent on
21 what `T` is, and that's why compilers insist on your asserting that it is a type by pre-
22 ceding it with `typename`.

23 If you've done any template metaprogramming (TMP), you've almost certainly
24 bumped up against the need to take template type parameters and create revised
25 types from them. For example, given some type `T`, you might want to strip off any
26 `const`-or reference-qualifiers that `T` contains, i.e., you might want to turn `const`
27 `std::string&` into `std::string`. Or you might want to take a type and add
28 `const` to it or turn it into an lvalue reference, i.e., turn `Widget` into `const Widget`
29 or into `Widget&`. (If you haven't done any TMP, that's too bad, because if you want
30 to be a truly effective C++ programmer, you need to be familiar with at least the
31 basics of this facet of C++. You can see examples of TMP in action, including the
32 kinds of type transformations I just mentioned, in Items 25 and 29.)

1 C++11 gives you the tools to perform these kinds of transformations in the form of
2 *type traits*, an assortment of templates inside the header `<type_traits>`. There
3 are dozens of type traits in that header, and not all of them perform type transfor-
4 mations, but the ones that do offer a predictable interface. Given a type `T` to which
5 you'd like to apply a transformation, the resulting type is
6 `std::transformation<T>::type`. For example:

```
7 std::remove_const<T>::type           // yields T from const T
8 std::remove_reference<T>::type        // yields T from T& and T&&
9 std::add_lvalue_reference<T>::type    // yields T& from T
```

10 The comments merely summarize what these transformations do, so don't take
11 them too literally. Before using them on a project, you'd look up the precise speci-
12 fications, I know.

13 My motivation here isn't to give you a tutorial on type traits, anyway. Rather, note
14 that application of these transformations entails writing “`::type`” at the end of
15 each use. If you apply them to a type parameter inside a template (which is virtual-
16 ly always how you employ them in real code), you'd also have to precede each use
17 with `typename`. The reason for both of these syntactic speed bumps is that the
18 C++11 type traits are implemented as nested `typedefs` inside templated
19 `structs`. That's right, they're implemented using the type synonym technology
20 I've been trying to convince you is inferior to alias templates!

21 There's a historical reason for that, but we'll skip over it (it's dull, I promise), be-
22 cause the Standardization Committee belatedly recognized that alias templates are
23 the better way to go, and they included such templates in C++14 for all the C++11
24 type transformations. The aliases have a common form: for each C++11 transfor-
25 mation `std::transformation<T>::type`, there's a corresponding C++14 alias
26 template named `std::transformation_t`. Examples will clarify what I mean:

```
27 std::remove_const<T>::type           // C++11: const T → T
28 std::remove_const_t<T>               // C++14 equivalent
29 std::remove_reference<T>::type        // C++11: T&/T&& → T
30 std::remove_reference_t<T>            // C++14 equivalent
```

```
1 std::add_lvalue_reference<T>::type // C++11: T → T&
2 std::add_lvalue_reference_t<T> // C++14 equivalent
3 The C++11 constructs remain valid in C++14, but I don't know why you'd want to
4 use them. Even if you don't have access to C++14, writing the alias templates your-
5 self is child's play. Only C++11 language features are required, and even children
6 can mimic a pattern, right? If you happen to have access to an electronic copy of
7 the C++14 Standard, it's easier still, because all that's required is some copying and
8 pasting. Here, I'll get you started (via copy-and-paste technology):
```

```
9 template <class T>
10 using remove_const_t = typename remove_const<T>::type;
11 template <class T>
12 using remove_reference_t = typename remove_reference<T>::type;
13 template <class T>
14 using add_lvalue_reference_t =
15     typename add_lvalue_reference<T>::type;
```

16 See? Couldn't be easier.

17 Things to Remember

- 18 • `typedefs` don't support templatization, but alias declarations do.
- 19 • Alias templates avoid the “`::type`” suffix and, in templates, the “`typename`”
- 20 prefix often required to refer to `typedefs`.
- 21 • C++14 offers alias templates for all the C++11 type traits transformations.

22 Item 10: Prefer scoped `enums` to unscoped `enums`.

23 As a general rule, declaring a name inside curly braces limits the visibility of that
24 name to the scope defined by the braces. Not so for the enumerators declared in C-
25 and C++98-style `enums`. The names of such enumerators belong to the scope con-
26 taining the `enum`, and that means that nothing else in that scope may have the
27 same name:

```
28 enum Color { black, white, red }; // black, white, red are
29 // in same scope as Color
30 bool white = false; // error! white already
31 // declared in this scope
```

1 The fact that these enumerator names leak into the scope containing their `enum`
2 definition gives rise to the official name for this kind of `enum`: *unscoped enums*.
3 Their C++11 scoped counterparts, *scoped enums*, don't leak names in this way:

```
4 enum class Color { black, white, red }; // black, white, red  
5 // are scoped to Color  
6 bool white = false; // fine, no other  
7 // "white" in scope  
8 Color c = white; // error! no enumerator named  
9 // "white" is in this scope  
10 Color c = Color::white; // fine  
11 auto c = Color::white; // also fine (and in accord  
12 // with Item 5's advice)
```

13 Because scoped `enums` are declared via “`enum class`”, scoped `enums` are sometimes referred to as *enum classes*.

15 The reduction in namespace pollution offered by scoped `enums` is reason enough to
16 prefer them over their unscoped siblings, but scoped `enums` have a second compelling advantage: their enumerators are much more strongly typed. Enumerators for
18 unscoped `enums` implicitly convert to integral types (and, from there, to floating
19 point types). Semantic travesties such as the following are therefore perfectly valid:
20

```
21 enum Color { black, white, red }; // unscoped enum  
22 std::vector<std::size_t> primeFactors(std::size_t x); // func. returning  
23 // prime factors of x  
24 Color c = red;  
25 ...  
26 if (c < 14.5) { // compare Color to double (!)  
27     auto factors = primeFactors(c); // compute prime factors  
28     // of a color (!)  
29     ...  
30 }
```

31 Throw a simple “`class`” after “`enum`”, however, thus transforming an unscoped
32 `enum` into a scoped one, and it’s a very different story, because there are no implicit
33 conversions from enumerators in a scoped `enum` to any other type:

```
1 enum class Color { black, white, red }; // enum is now scoped
2 Color c = Color::red; // as before, but
3 ... // with scope qualifier
4 if (c < 14.5) { // error! can't compare
5 // Color and double
6 auto factors = // error! can't pass Color to
7 primeFactors(c); // function expecting std::size_t
8 ...
9 }
```

10 If you honestly want to perform a conversion from `Color` to a different type, do
11 what you always do to twist the type system to your wanton desires: use a cast:

```
12 if (static_cast<double>(c) < 14.5) { // odd code, but
13 // it's valid
14 auto factors = // suspect, but
15 primeFactors(static_cast<std::size_t>(c)); // it compiles
16 ...
17 }
```

18 It may seem that scoped `enums` have a third advantage over unscoped `enums`, be-
19 cause scoped `enums` may be forward-declared, i.e., their names may be declared
20 without specifying their enumerators:

```
21 enum Color; // error!
22 enum class Color; // fine
```

23 This is misleading. In C++11, unscoped `enums` may also be forward-declared, but
24 only after a bit of additional work.

25 Every `enum` in C++ has an integral *underlying type* that is determined by compilers.
26 For an unscoped `enum` like `Color`,

```
27 enum Color { black, white, red };
28 compilers might choose char as the underlying type, because there are only three
29 values to represent. On the other hand, some enums may have a range of values
30 that is much larger, e.g.:
```

```
31 enum Status { good = 0,
32 failed = 1,
33 incomplete = 100,
```

```
1         corrupt = 200,
2         indeterminate = 0xFFFFFFFF
3     };
4 Here the values to be represented range from 0 to 0xFFFFFFFF. Except on very
5 unusual machines (where a char consists of at least 32 bits), compilers will have
6 to select an integral type larger than char for the underlying representation of
7 Status values.
```

```
8 To make efficient use of memory, compiler writers want to choose the smallest
9 underlying type for each enum that's sufficient to represent the enum's range of
10 enumerator values. (In some cases, compilers will optimize for speed instead of
11 size, and in that case, they may not choose the smallest permissible underlying
12 type, but they certainly want to be able to optimize for minimal size.) To make that
13 possible, C++98 requires that enums be defined (i.e., list all their enumerators)
14 where they're declared. Having seen the range of enumerator values that must be
15 representable, compilers are then in a position to choose a suitable underlying
16 type for that enum type.
```

```
17 But the inability to forward-declare enums has drawbacks, too. Perhaps the most
18 notable is the increase in compilation dependencies. Consider again the Status
19 enum:
```

```
20 enum Status { good = 0,
21             failed = 1,
22             incomplete = 100,
23             corrupt = 200,
24             indeterminate = 0xFFFFFFFF
25         };
```

```
26 This is the kind of enum that's likely to be used throughout a system, hence includ-
27 ed in a header file that every part of the system is dependent on. If a new status
28 value is then introduced,
```

```
29 enum Status { good = 0,
30             failed = 1,
31             incomplete = 100,
32             corrupt = 200,
33             audited = 500,
34             indeterminate = 0xFFFFFFFF
35         };
```

1 it's likely that the entire system will have to be recompiled, even if only a single
2 subsystem—possibly only a single function!—uses the new enumerator. This is
3 the kind of thing that people *hate*. And it's the kind of thing that the ability to for-
4 ward-declare `enums` would eliminate. With forward-declared `enums`, declarations
5 for `enum`-manipulating functions would be freed of their dependencies on the `enum`
6 definitions. For example, here's a perfectly valid C++11 declaration of a scoped
7 `enum` and a function that takes one as a parameter:

```
8 enum class Status;           // forward declaration
9 void continueProcessing(Status s); // use of fwd-declared enum
```

10 With this design, the header containing these two declarations requires no
11 recompilation if the enumerators in `Status` are revised. Furthermore, if the
12 enumerators in `Status` are revised (e.g., to add the new enumerator `audited`),
13 but `continueProcessing`'s behavior is unaffected (e.g., because `continuePro-`
14 `cessing` doesn't care about the `audited` value), `continueProcessing`'s
15 implementation need not be recompiled, either.

16 But if code generation requires knowing the size of the enumerators (and it does),
17 how can C++11's scoped `enums` get away with forward declarations when C++98's
18 unscoped `enums` can't? That's easy: compilers always know the underlying type for
19 scoped `enums`.

20 All scoped `enums` have a default underlying type of `int`:

```
21 enum class Status;           // underlying type is int
```

22 If you don't like the default, you may specify one explicitly:

```
23 enum class Status: std::uint32_t; // underlying type for
24                                     // Status is std::uint32_t
25                                     // (from <cstdint>)
```

26 Either way, compilers know the size of the enumerators in a scoped `enum`.

27 But I said that this isn't really an advantage of scoped `enums` over unscoped `enums`.
28 That's because C++11 allows you to specify the underlying type for unscoped
29 `enums`, too. When you do, those `enums` can be forward-declared, just like their
30 scoped compatriots:

```
1 enum Color: std::uint8_t;           // fwd decl for unscoped enum;
2                                     // underlying type is
3                                     // std::uint8_t
4 Underlying type specifications aren't restricted to enum declarations. You can put
5 them on the definitions, too:
```

```
6 enum class Status: std::uint32_t { good = 0,
7                                     failed = 1,
8                                     incomplete = 100,
9                                     corrupt = 200,
10                                    audited = 500,
11                                    indeterminate = 0xFFFFFFFF
12 };
```

13 It should go without saying that the underlying type you specify when you declare
14 an enum must match the underlying type you specify when you define the enum. In
15 general, mismatches will be diagnosed during compilation, although there are sit-
16 uations where compilers are unable to do this. As long as you put all your enum
17 declarations in header files and dutifully #include those header files everywhere
18 you use the enums, you'll be fine. If you try to cut corners by skipping an #include
19 and manually declaring your enum instead, you may not be fine, and the almost
20 invariably resulting overnight debugging sessions should serve to remind you not
21 to commit such crimes against software engineering.

22 In view of the fact that scoped enums avoid namespace pollution and aren't suscep-
23 tible to nonsensical implicit type conversions, it may surprise you to hear that
24 there's at least one situation in C++11 where unscoped enums are sometimes con-
25 sidered useful. That's when referring to fields within std::tuples. For example,
26 suppose we have a tuple holding values for the name, email address, and reputa-
27 tion value for a user at a social networking web site:

```
28 using UserInfo =           // type alias; see Item 9
29   std::tuple<std::string,      // name
30             std::string,      // email
31             std::size_t>;     // reputation
```

32 Though the comments indicate what each field of the tuple represents, that's
33 probably not very helpful when you encounter code like this in a separate source
34 file:

```
1 UserInfo uInfo; // object of tuple type
2 ...
3 auto val = std::get<1>(uInfo); // get value of field 1
4 As a programmer, you have a lot of stuff to keep track of. Should you really be
5 expected to remember that field 1 corresponds to the user's email address? I think
6 not. Use of an unscoped enum avoids the need to:
```

```
7 enum UserInfoFields { uiName, uiEmail, uiReputation };
8 UserInfo uInfo; // as before
9 ...
10 auto val = std::get<uiEmail>(uInfo); // ah, get value of
11 // email field
12 You need to use an unscoped enum here, because std::get requires a
13 std::size_t argument, and that means you need to take advantage of the implicit
14 conversion from UserInfoFields to std::size_t that unscoped enums support.
```

16 The corresponding code with scoped enums is substantially more verbose:

```
17 enum class UserInfoFields { uiName, uiEmail, uiReputation };
18 UserInfo uInfo; // as before
19 ...
20 auto val =
21     std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
22     (uInfo);
```

23 The verbosity can be reduced by writing a function that takes an enumerator and
24 returns its corresponding std::size_t value, but it's a bit tricky. std::get is a
25 template, and the value you provide is a template argument (notice the use of an-
26 gle brackets, not parentheses), so the function that transforms an enumerator into
27 a std::size_t has to produce its result *during compilation*. As Item 14 explains,
28 that means it must be a constexpr function. In fact, it should really be a con-
29 stexpr function template, because it should work with any kind of enum:

```
30 template<typename E>
31 constexpr std::size_t toIndex(E enumerator)
32 {
```

```
1     return static_cast<std::size_t>(enumerator);  
2 }
```

3 This `toIndex` template permits us to access a field of the tuple this way:

```
4 auto val = std::get<toIndex(UserInfoFields::uiEmail)>(uInfo);
```

5 It's still more to write than use of the unscoped `enum`, of course, but it also avoids
6 namespace pollution and inadvertent conversions involving enumerators. In many
7 cases, you may decide that typing a few extra characters is a reasonable price to
8 pay for the ability to avoid an enum technology that dates to a time when the state
9 of the art in digital telecommunications was the 2400-baud modem.

10 **Things to Remember**

- 11 • C++98-style `enums` are now known as unscoped `enums`.
- 12 • Enumerators of scoped `enums` are visible only within the `enum`. They convert to
13 other types only with a cast.
- 14 • Both scoped and unscoped `enums` support specification of the underlying type.
15 The default underlying type for scoped `enums` is `int`. Unscoped `enums` have no
16 default underlying type.
- 17 • Scoped `enums` may always be forward-declared. Unscoped `enums` may be for-
18 ward declared only if their declaration specifies an underlying type.

19 **Item 11: Prefer deleted functions to private undefined ones.**

20 If you're providing code to other developers, and you want to prevent them from
21 calling a particular function, you generally just don't declare the function. No func-
22 tion declaration, no function to call. Easy, peasy. But sometimes C++ declares func-
23 tions for you, and if you want to prevent clients from calling those functions, the
24 peasy isn't quite so easy any more.

25 This situation arises only for the *special member functions*, i.e., the member func-
26 tions that C++ automatically generates when they're needed. Item 19 discusses
27 these functions in detail, but for now, we'll worry only about the copy constructor
28 and the copy assignment operator. This chapter is devoted to common practices in
29 C++98 that have been superseded by better practices in C++11, and in C++98, if

1 you want to suppress use of a member function, it's almost always the copy constructor, the assignment operator, or both.

3 The C++98 approach to preventing use of these functions is to declare them **private** and to avoid defining them. For example, near the base of the iostreams hierarchy in the C++ Standard Library is the class template **basic_ios**. All **istream** and **ostream** classes inherit (possibly indirectly) from this class. Copying **istreams** and **ostreams** is undesirable, because it's not really clear what such operations should do. An **istream** object, for example, represents a stream of input values, some of which may have already been read, and some of which will potentially be read later. If an **istream** were to be copied, would that entail copying all the values that had already been read as well as all the values that would be read in the future? The easiest way to deal with such questions is to define them out of existence. Prohibiting the copying of streams does just that.

14 To render **istream** and **ostream** classes uncopyable, **basic_ios** is specified in C++98 as follows (including the comments):

```
16 template <class charT, class traits = char_traits<charT> >
17 class basic_ios : public ios_base {
18 public:
19 ...
20 private:
21     basic_ios(const basic_ios& );           // not defined
22     basic_ios& operator=(const basic_ios&); // not defined
23 };
```

24 Declaring these functions **private** prevents clients from calling them. Deliberately failing to define them means that if code that still has access to them (i.e., member functions or friends of the class) uses them, linking will fail due to missing function definitions.

28 In C++11, there's a better way to achieve essentially the same end: use “**=delete**” to mark the copy constructor and the copy assignment operator as **deleted functions**. Here's the same part of **basic_ios** as it's specified in C++11:

```
31 template <class charT, class traits = char_traits<charT> >
32 class basic_ios : public ios_base {
33 public:
```

```
1 ...
2     basic_ios(const basic_ios& ) = delete;
3     basic_ios& operator=(const basic_ios&) = delete;
4 ...
5 };

6 The difference between deleting these functions and declaring them private may
7 seem more a matter of fashion than anything else, but there's greater substance
8 here than you might think. Deleted functions may not be used in any way, so even
9 code that's in member and friend functions will fail to compile if it tries to copy
10 basic_ios objects. That's an improvement over the C++98 behavior, where such
11 improper usage wouldn't be diagnosed until link-time.

12 By convention, deleted functions are declared public, not private. There's a rea-
13 son for that. When client code tries to use a member function, C++ checks accessi-
14 bility before =delete status. When client code tries to use a deleted private
15 function, some compilers complain only about the function being private, even
16 though the function's accessibility doesn't really affect whether it can be used. It's
17 worth bearing this in mind when revising legacy code to replace private-and-
18 not-defined member functions with deleted ones, because making the new func-
19 tions public will generally result in superior error messages.

20 An important advantage of deleted functions is that any function may be deleted,
21 while only member functions may be private. For example, suppose we have a
22 non-member function that takes an integer and returns whether it's a lucky num-
23 ber:

24 bool isLucky(int number);

25 C++'s C heritage means that pretty much any type that can be viewed as vaguely
26 numerical will implicitly convert to int, but some calls that would compile might
27 not make sense:

28 if (isLucky('a')) ...           // is 'a' a lucky number?
29 if (isLucky(true)) ...          // is "true"?
30 if (isLucky(3.5)) ...           // should we truncate to 3
31                                         // before checking for luckiness?
```

1 If lucky numbers must really be integers, we'd like to prevent calls such as these
2 from compiling.

3 One way to accomplish that is to create deleted overloads for the types we want to
4 filter out:

```
5 bool isLucky(int number);           // original function
6 bool isLucky(char) = delete;        // reject chars
7 bool isLucky(bool) = delete;        // reject bools
8 bool isLucky(double) = delete;      // reject doubles and
9                           // floats
```

10 (The comment on the `double` overload that says that both `doubles` and `floats`
11 will be rejected may initially surprise you, but your surprise will dissipate once
12 you recall that, given a choice between converting a `float` to an `int` or to a dou-
13 ble, C++ prefers the conversion to `double`. Calling `isLucky` with a `float` will
14 therefore call the `double` overload, not the `int` one. Well, it'll try to. The fact that
15 that overload is deleted will prevent the call from compiling.)

16 Although deleted functions can't be used, they are part of your program. As such,
17 they are taken into account during overload resolution. That's why, with the delet-
18 ed function declarations above, the undesirable calls to `isLucky` will be rejected:

```
19 if (isLucky('a')) ...           // error! call to deleted function
20 if (isLucky(true)) ...          // error!
21 if (isLucky(3.5f)) ...          // error!
```

22 Another trick that deleted functions can perform (and that `private` member func-
23 tions can't) is to prevent use of template instantiations that should be disabled. For
24 example, suppose you need a template that works with built-in pointers
25 (Chapter 4's advice to prefer smart pointers to raw pointers notwithstanding):

```
26 template<typename T>
27 void processPointer(T* ptr);
```

28 There are two special cases in the world of pointers. One is `void*` pointers, be-
29 cause there is no way to dereference them, to increment or decrement them, etc..
30 The other is `char*` pointers, because they typically represent pointers to C-style

1 strings, not pointers to individual characters. These special cases often call for spe-
2 cial handling, and, in the case of the `processPointer` template, let's assume the
3 proper handling is to reject calls using those types. That is, it should not be possi-
4 ble to call `processPointer` with `void*` or `char*` pointers.

5 That's easily enforced. Just delete those instantiations:

```
6 template<>
7 void processPointer<void>(void*) = delete;
8 template<>
9 void processPointer<char>(char*) = delete;
```

10 Now, if calling `processPointer` with a `char*` is invalid, it's probably also invalid
11 to call it with a `const char*`, so that instantiation will typically need to be deleted,
12 too:

```
13 template<>
14 void processPointer<const char>(const char*) = delete;
```

15 If you really want to be thorough, you'll also delete the `const volatile char*`
16 overload, and then you'll get to work on the overloads for pointers to the other
17 standard character types: `std::wchar_t`, `std::char16_t`, and `std::char32_t`.

18 Interestingly, if you have a function template inside a class, and you'd like to dis-
19 able some instantiations by declaring them `private` (à la classic C++98 conven-
20 tion), you can't, because it's not possible to give a member function template spe-
21 cialization a different access level from that of the main template. If `pro-`
22 `cessPointer` were a member function template inside `Widget`, for example, and
23 we wanted to disable calls for `void*` pointers, this would be the C++98 approach,
24 though it would not compile:

```
25 class Widget {
26 public:
27 ...
28     template<typename T>
29     void processPointer(T* ptr)
30     { ... }
31 private:
32     template<> // error!
33     void processPointer<void>(void*);
```

1 };
2 The problem is that template specializations must be written at namespace scope,
3 not class scope. This issue doesn't arise for deleted functions, because they don't
4 need a different access level. They can be deleted outside the class (hence at
5 namespace scope):

```
6  class Widget {  
7  public:  
8  ...  
9  template<typename T>  
10 void processPointer(T* ptr)  
11 { ... }  
12 ...  
13 };  
14 template<>  
15 void Widget::processPointer<void>(void*) = delete; // still  
16 // public,  
17 // but  
18 // deleted
```

18 The truth is that the C++98 practice of declaring functions `private` and not defining them was really an attempt to achieve what C++11's deleted functions actually do. As an emulation, the C++98 approach is not as good as the real thing. It doesn't work outside classes, it doesn't always work inside classes, and when it does work, it may not work until link-time. So stick to deleted functions.

23 **Things to Remember**

- 24 ♦ Prefer deleted functions to private undefined ones.
- 25 ♦ Any function may be deleted, including non-member functions and template instantiations.

27 **Item 12: Declare overriding functions `override`.**

28 The world of object-oriented programming in C++ revolves around classes, inheritance, and virtual functions. Among the most fundamental ideas in this world is 29 that virtual function implementations in derived classes *override* the implementations of their base class counterparts. It's disheartening, then, to realize how easily 30 31 virtual function overriding can go wrong. It's almost as if this part of the language 32 virtual function overriding can go wrong. It's almost as if this part of the language

1 were designed with the idea that Murphy's Law wasn't just to be obeyed, it was to
2 be honored.

3 Because "overriding" sounds a lot like "overloading," yet is completely unrelated,
4 let me make clear that virtual function overriding is what makes it possible to in-
5voke a derived class function through a base class interface:

```
6 class Base {  
7 public:  
8     virtual void doWork();           // base class virtual function  
9     ...  
10};  
  
11 class Derived: public Base {  
12 public:  
13     virtual void doWork();         // overrides Base::doWork  
14     ...                           // ("virtual" is optional  
15};                                // here")  
  
16 std::unique_ptr<Base> upb =      // create base class pointer  
17     std::make_unique<Derived>();    // to derived class object;  
18                                     // see Item 23 for info on  
19                                     // std::make_unique  
  
20 upb->doWork();                  // call doWork through base  
21                                     // class ptr; derived class  
22                                     // function is invoked
```

23 For overriding to occur, several requirements must be met:

- 24 • The base class function must be virtual.
- 25 • The base and derived function names must be identical (except in the case of
26 destructors).
- 27 • The parameter types of the base and derived functions must be identical.
- 28 • The **constness** of the base and derived functions must be identical.

29 To these constraints, which have been part of C++ since the beginning, C++11 adds
30 one more:

- 31 • The functions' **reference qualifiers** must be identical. Member function refer-
32 ence qualifiers are one of C++11's less-publicized features, so don't be sur-
33 prised if you've never heard of them. They make it possible to limit use of a

1 member function to lvalues only or to rvalues only. Member functions need not
2 be virtual to use them:

```
3 class Widget {  
4 public:  
5     ...  
6     void doWork() &;      // this version of doWork applies only  
7                         // when *this is an lvalue  
8     void doWork() &&;    // this version of doWork applies only  
9                         // when *this is an rvalue  
10    ...  
11    Widget makeWidget(); // factory function (returns rvalue)  
12    Widget w;           // normal object (an lvalue)  
13    ...  
14    w.doWork();         // calls Widget::doWork for lvalues  
15                         // (i.e., Widget::doWork &)  
16    makeWidget().doWork(); // calls Widget::doWork for rvalues  
17                         // (i.e., Widget::doWork &&)
```

18 I'll say more about member functions with reference qualifiers later, but for
19 now, simply note that if a virtual function in a base class has a reference qual-
20 fier, derived class overrides of that function must have exactly the same refer-
21 ence qualifier. If they don't, the declared functions will still exist in the derived
22 class, but they won't override anything in the base class.

23 All these requirements for overriding mean that small mistakes can make a big
24 difference. Code with overriding errors is typically valid, but its meaning isn't what
25 you intended. As a result, you can't rely on compilers to notify you if you do some-
26 thing wrong. For example, the following code is completely legal and, at first sight,
27 looks reasonable, but it contains no virtual function overrides—not a single de-
28 rived class function that is tied to a base class function. Can you identify the prob-
29 lem in each case, i.e., why the derived class function doesn't override the base class
30 function with the same name?

```
31 class Base {  
32 public:  
33     virtual void mf1() const;  
34     virtual void mf2(int x);
```

```
1   virtual void mf3() &;
2   void mf4() const;
3 };
4 class Derived: public Base {
5 public:
6   virtual void mf1();
7   virtual void mf2(unsigned int x);
8   virtual void mf3() &&;
9   void mf4() const;
10};
11 Need some help?
12 • mf1 is declared const in Base, but not in Derived.
13 • mf2 takes an int in Base, but an unsigned int in Derived.
14 • mf3 is lvalue-qualified in Base, but rvalue-qualified in Derived.
15 • mf4 isn't declared virtual in Base.
16 You may think, "Hey, in practice, these things will elicit compiler warnings, so I
17 don't need to worry." Maybe that's true. But maybe it's not. With two of the com-
18 pilers I checked, the code was accepted without complaint, and that was with all
19 warnings enabled. (Other compilers provided warnings about some—but not all—
20 of the issues.)
21 Because declaring derived class overrides is important to get right, but easy to get
22 wrong, C++11 gives you a way to make explicit that a derived class function is sup-
23 posed to override a base class version. Just declare it override. Applying this to
24 the example above would yield this derived class:
25 class Derived: public Base {
26 public:
27   virtual void mf1() override;
28   virtual void mf2(unsigned int x) override;
29   virtual void mf3() && override;
30   virtual void mf4() const override;
31 };
32 This won't compile, of course, because when written this way, compilers will
33 kvetch about all the overriding-related problems. That's exactly what you want,
34 and it's why you should declare all your overriding functions override.
```

1 The code using `override` that does compile looks as follows (assuming that the
2 goal is for all functions in `Derived` to override `virtuals` in `Base`):

```
3 class Base {
4 public:
5     virtual void mf1() const;
6     virtual void mf2(int x);
7     virtual void mf3() &;
8     virtual void mf4() const;
9 };
10 class Derived: public Base {
11 public:
12     virtual void mf1() const override;
13     virtual void mf2(int x) override;    // note revised param type
14     virtual void mf3() & override;
15     void mf4() const override;          // adding "virtual" is OK,
16 };                                // but not necessary
```

17 Note that in this example, part of getting things to work involves declaring `mf4` vir-
18 tual in `Base`. Most overriding-related errors occur in derived classes, but it's pos-
19 sible for things to be incorrect in base classes, too.

20 A policy of using `override` on all your derived class overrides can do more than
21 just enable compilers to tell you when would-be overrides aren't overriding any-
22 thing. It can also help you gauge the ramifications if you're contemplating changing
23 the signature of a virtual function in a base class. If derived classes use `override`
24 everywhere, you can just change the signature, recompile your system, see how
25 much damage you've caused (e.g., how many derived classes fail to compile), then
26 decide whether the signature change is worth the trouble. Without `override`,
27 you'd have to hope you have comprehensive unit tests in place, because, as we've
28 seen, derived class `virtuals` that are supposed to override base class functions, but
29 don't, need not elicit compiler diagnostics.

30 C++ has always had keywords, but C++11 introduces two *contextual keywords*,
31 `override` and `final`. These keywords have the characteristic that they are re-
32 served, but only in certain contexts. In the case of `override`, it has a reserved
33 meaning only when it occurs at the end of a member function declaration. That
34 means that if you have legacy code that already uses the name `override`, you
35 don't need to change it for C++11:

```
1 class Warning {           // potential legacy class from C++98
2 public:
3 ...
4     void override();      // legal in both C++98 and C++11
5     ...                   // (with the same meaning)
6 };
```

7 That's all there is to say about `override`, but it's not all there is to say about
8 member function reference qualifiers. I promised I'd provide more information on
9 them later, and it's now later.

10 If we want to write a function that accepts only lvalue arguments, we declare an
11 lvalue reference parameter:

```
12 void doSomething(Widget& w);      // accepts only lvalue Widgets
```

13 If we want to write a function that accepts only rvalue arguments, we declare an
14 rvalue reference parameter:

```
15 void doSomething(Widget&& w);      // accepts only rvalue Widgets
```

16 Member function reference qualifiers simply make it possible to draw the same
17 distinction for the object on which a member function is invoked, i.e., `*this`. It's
18 precisely analogous to the `const` at the end of a member function declaration,
19 which indicates that the object on which the member function is invoked (i.e.,
20 `*this`) is `const`.

21 The need for reference-qualified member functions is not common, but it does
22 arise. For example, suppose our `Widget` class has a `std::vector` data member,
23 and we offer an accessor function that gives clients direct access to it:

```
24 class Widget {
25 public:
26     using DataType = std::vector<double>;          // see Item 9 for
27     ...                                              // info on "using"
28     DataType& data() { return values; }
29     ...
30 private:
31     DataType values;
32 };
```

1 This is hardly the most encapsulated design that's seen the light of day, but set
2 that aside and consider what happens in this client code:

```
3 Widget w;  
4 ...  
5 auto vals1 = w.data(); // copy w.values into vals1  
6 The return type of Widget::data is an lvalue reference (a  
7 std::vector<double>&, to be precise), and because lvalue references are de-  
8 fined to be lvalues, we're initializing vals1 from an lvalue. vals1 is thus copy-  
9 constructed from w.values, just as the comment says.
```

10 Now suppose we have a factory function that creates Widgets,

```
11 Widget makeWidget();  
12 and we want to initialize a variable with the std::vector inside the Widget re-  
13 turned from makeWidget:
```

```
14 auto vals2 = makeWidget().data(); // copy values inside the  
15 // Widget into vals2  
16 Again, Widgets::data returns an lvalue reference, and, again, the lvalue refer-  
17 ence is an lvalue, so, again, our new object (vals2) is copy-constructed from val-  
18 ues inside the Widget. This time, though, the Widget is the temporary object re-  
19 turned from makeWidget (i.e., an rvalue), so copying the std::vector inside it is  
20 a waste of time. It'd be preferable to move it, but, because data is returning an  
21 lvalue reference, the rules of C++ require that compilers generate code for a copy.  
22 (There's some wiggle room for optimization through what is known as the "as if  
23 rule," but you'd be foolish to rely on your compilers finding a way to take ad-  
24 vantage of it.)
```

25 What's needed is a way to specify that when data is invoked on an rvalue Widget,
26 the result should also be an rvalue. Using reference qualifiers to overload data for
27 lvalue and rvalue Widgets makes that possible:

```
28 class Widget {  
29 public:  
30     using DataType = std::vector<double>;  
31     ...
```

```

1   DataType& data() &           // for lvalue Widgets,
2   { return values; }           // return lvalue
3
4   DataType data() &&          // for rvalue Widgets,
5   { return std::move(values); } // return rvalue
6   ...
7
8 private:
9   DataType values;
10 }

```

Notice the differing return types from the `data` overloads. The lvalue reference overload returns an lvalue reference (i.e., an lvalue), and the rvalue reference overload returns a temporary object (i.e., an rvalue). This means that client code now behaves as we'd like:

```

13 auto vals1 = w.data();           // calls lvalue overload for
14                                         // Widget::data, copy-
15                                         // constructs vals1
16 auto vals2 = makeWidget().data(); // calls rvalue overload for
17                                         // Widget::data, move-
18                                         // constructs vals2

```

This is certainly nice, but don't let the warm glow of this happy ending distract you from the true point of this Item. That point is that whenever you declare a function in a derived class that's meant to override a virtual function in a base class, be sure to declare that function `override`.

Things to Remember

- ♦ Declare overriding functions `override`.
- ♦ Member function reference qualifiers make it possible to treat lvalue and rvalue objects (`*this`) differently.

Item 13: Prefer `const_iterators` to `iterators`.

`const_iterators` are the STL equivalent of pointers-to-`const`. They point to values that may not be modified. The standard practice of using `const` whenever possible dictates that you should use `const_iterators` any time you need an iterator, yet have no need to modify what the iterator points to.

1 That's as true for C++98 as for C++11 and C++14, but in C++98, `const_iterators`
2 had only halfhearted support. It wasn't that easy to create them, and once you had
3 one, the ways you could use it were limited. For example, suppose you want to
4 search a `std::vector<int>` for the first occurrence of 1983 (the year "C++" re-
5 placed "C with Classes" as the name of the programming language), then insert the
6 value 1998 (the year the first C++ Standard was adopted) at that location. If there's
7 no 1983 in the vector, the insertion should go at the end of the vector. Using `iter-`
8 `ators` in C++98, that was easy:

```
9 std::vector<int> values;  
10 ...  
11 std::vector<int>::iterator it =  
12     std::find(values.begin(),values.end(), 1983);  
13 values.insert(it, 1998);
```

14 But `iterators` aren't really the proper choice here, because this code never modi-
15 fies what an `iterator` points to. Revising the code to use `const_iterators`
16 should be trivial, but in C++98, it was anything but. Here's one approach that's
17 conceptually sound, though still not correct:

```
18 typedef std::vector<int>::iterator IterT;           // type-  
19 typedef std::vector<int>::const_iterator ConstIterT; // defs  
20 std::vector<int> values;  
21 ...  
22 ConstIterT ci =  
23     std::find(static_cast<ConstIterT>(values.begin()), // cast  
24                 static_cast<ConstIterT>(values.end()), // cast  
25                 1983);  
26 values.insert(static_cast<IterT>(ci), 1998);      // may not  
27                                         // compile; see  
28                                         // below
```

29 The `typedefs` aren't required, of course, but they make the casts in the code easier
30 to write. (If you're wondering why I'm showing `typedefs` instead of following the
31 advice of Item 9 to use alias declarations, it's because this example shows C++98
32 code, and alias declarations are a feature new to C++11.)

1 The casts in the call to `std::find` are present because `values` is a non-`const`
2 container and, in C++98, there was no simple way to get a `const_iterator` from
3 a non-`const` container. The casts aren't strictly necessary, because it was possible
4 to get `const_iterators` in other ways,[†] but regardless of how you did it, the
5 process of getting `const_iterators` to elements of a non-`const` container
6 involved some amount of contorting.

7 Once you had the `const_iterators`, matters often got worse, because in C++98,
8 locations for insertions (and erasures) could be specified only by `iterators`.
9 `const_iterators` weren't acceptable. That's why, in the code above, I cast the
10 `const_iterator` (that I was so careful to get from `std::find`) into an
11 `iterator`: passing a `const_iterator` to `insert` wouldn't compile.

12 To be honest, the code I've shown might not compile, either, because there's no
13 portable conversion from a `const_iterator` to an `iterator`, not even with a
14 `static_cast`. Even the semantic sledgehammer known as `reinterpret_cast`
15 can't do the job. (That's not a C++98 restriction. It's true in C++11 and C++14, too.
16 `const_iterators` simply don't convert to `iterators`, no matter how much it
17 might seem like they should.) There are some portable ways to generate `itera-`
18 `tors` that point where `const_iterators` do, but they're not obvious, not
19 universally applicable, and not worth discussing in this book. Besides, I hope that
20 by now my point is clear: `const_iterators` were so much trouble in C++98, they
21 were rarely worth the bother. At the end of the day, developers don't use `const`
22 whenever *possible*, they use it whenever *practical*, and in C++98,
23 `const_iterators` just weren't that practical.

24 All that changed in C++11. Now `const_iterators` are both easy to get and easy to
25 use. The container member functions `cbegin` and `cend` produce
26 `const_iterators`, even for non-`const` containers, and STL member functions
27 that use `iterators` to identify positions (e.g., `insert` and `erase`) actually use

[†] For example, you could bind `values` to a reference-to-`const` variable, then use that variable in place of `values` in your code.

1 `const_iterators`. Revising the original C++98 code that uses `iterators` to use
2 `const_iterators` in C++11 is truly trivial:

```
3 std::vector<int> values; // as before
4 ...
5 auto it = // use cbegin
6     std::find(values.cbegin(), values.cend(), 1983); // and cend
7 values.insert(it, 1998);
```

8 Now that's code using `const_iterators` that's practical! If we decide we'd prefer
9 to do a reverse iteration—inserting 1998 at the location of the *last* occurrence of
10 1983 (or, if there are no such occurrences, at the beginning of the container)—the
11 problem is no more challenging, provided we remember that the way to convert a
12 reverse iterator to a “normal” iterator is to call its `base` member function:

```
13 auto it = // use
14     std::find(values.crbegin(), values.crend(), 1983); // crbegin
15                                             // and crend
16 values.insert(it.base(), 1998);
```

17 About the only situation in which C++11's support for `const_iterators` comes
18 up a bit short is when you want to write maximally generic library code. Such code
19 takes into account that some containers and container-like data structures offer
20 `begin` and `end` (plus `cbegin`, `cend`, `rbegin`, etc.) as *non-member* functions, rather
21 than members. This is the case for built-in arrays, for example, and it's also the
22 case for some third-party libraries with interfaces consisting only of free functions.
23 Maximally generic code thus uses non-member functions rather than assuming the
24 existence of member versions.

25 For example, we could generalize the code we've been working with into a
26 `findAndInsert` template as follows:

```
27 template<typename C, typename V>
28 void findAndInsert(C& container, // in container,
29                   const V& targetVal, // find 1st occurrence
30                   const V& insertVal) // of targetVal, then
31 { // insert insertVal
32 // there
```

```
1 auto it = std::find(std::cbegin(container), // non-mbr cbegin
2                     std::cend(container), // non-mbr cend
3                     targetVal);
4
5 }  
6
```

6 This works fine in C++14, but, sadly, not in C++11. Through an oversight during
7 standardization, C++11 added the non-member functions `begin` and `end`, but it
8 failed to add `cbegin`, `cend`, `rbegin`, `rend`, `crbegin`, and `crend`. C++14 rectifies
9 that oversight.

10 If you're using C++11, you want to write maximally generic code, and none of the
11 libraries you're using provides the missing templates for non-member `cbegin` and
12 friends, you can throw your own implementations together with ease. For exam-
13 ple, here's an implementation of non-member `cbegin`:

```
14 template <class C>
15 auto cbegin(const C& container)->decltype(std::begin(container))
16 {
17     return std::begin(container);           // see explanation below
18 }
```

19 You're surprised to see that non-member `cbegin` doesn't call member `cbegin`,
20 aren't you? I was, too. But follow the logic. This `cbegin` template accepts any type
21 of argument representing a container-like data structure, `C`, and it accesses this
22 argument through its reference-to-const parameter, `container`. If `C` is a conven-
23 tional container type (e.g., a `std::vector<int>`), `container` will be a reference
24 to a `const` version of that container (e.g., a `const std::vector<int>&`). Invoking
25 the non-member `begin` function (provided by C++11) on a `const` container yields
26 a `const_iterator`, and that iterator is what this template returns. The advantage
27 of implementing things this way is that it works even for containers that offer a
28 `begin` member function (which, for containers, is what C++11's non-member
29 `begin` calls), but fail to offer a `cbegin` member. You can thus use this non-member
30 `cbegin` on containers that directly support only `begin`.

31 This template also works if `C` is a built-in array type. In that case, `container` be-
32 comes a reference to a `const` array. C++11 provides a specialized version of non-
33 member `begin` for arrays that returns a pointer to the array's first element. (Itera-

1 tors into arrays are pointers.) The elements of a `const` array are `const`, so the
2 pointer that non-member `begin` returns for a `const` array is a pointer-to-`const`,
3 and a pointer-to-`const` is, in fact, a `const_iterator` for an array. (For insight in-
4 to how a template can be specialized for built-in arrays, consult Item 1's discussion
5 of type deduction in templates that take reference parameters to arrays.)

6 But back to basics. The point of this Item is to encourage you to use
7 `const_iterators` whenever you can. The fundamental motivation—using `const`
8 whenever it's meaningful—predates C++11, but in C++98, it simply wasn't practi-
9 cal when working with iterators. In C++11, it's eminently practical, and C++14 ti-
10 dies up the few bits of unfinished business that C++11 left behind.

11 **Things to Remember**

- 12 ♦ Prefer `const_iterators` to `iterators`.
- 13 ♦ In maximally generic code, prefer non-member versions of `begin`, `end`,
14 `rbegin`, etc., over their member function counterparts.

15 **Item 14: Use `constexpr` whenever possible.**

16 If there were an award for the most confusing new word in C++11, `constexpr`
17 would probably win it. When applied to objects, it's essentially a beefed-up form of
18 `const`, but when applied to functions, it has a quite different meaning. Cutting
19 through the confusion is worth the trouble, because when `constexpr` corresponds
20 to what you want to express, you definitely want to use it.

21 Conceptually, `constexpr` indicates a value that's not only constant, its value is
22 known during compilation. The concept is only part of the story, though, because
23 when `constexpr` is applied to functions, things are more nuanced than this sug-
24 gests. Lest I ruin the surprise ending, for now I'll just say that you can't assume
25 that the results of `constexpr` functions are `const`, nor can you take for granted
26 that their values are known during compilation. Perhaps most intriguingly, these
27 things are *features*. It's *good* that `constexpr` functions need not produce results
28 that are `const` or known during compilation!

1 But let's begin with `constexpr` objects. Such objects are, in fact, `const`, and they
2 do, in fact, have values that are known at compile-time. (Technically, their values
3 are determined during *translation*, and translation consists not just of compilation
4 but also of linking. Unless you write compilers or linkers for C++, however, this has
5 no effect on you, so you can blithely program as if the values of `constexpr` objects
6 were determined during compilation.)

7 Values known during compilation are privileged. They may be placed in read-only
8 memory, for example, and, especially for developers of embedded systems, this
9 can be a feature of considerable importance. Of broader applicability is that integral
10 values that are constant and known during compilation can be used in contexts
11 where C++ requires an *integral constant expression*. Such contexts include
12 specification of array sizes, integral template arguments (including lengths of
13 `std::array` objects), enumerator values, alignment specifiers, and more. If you
14 want to use a variable for these kinds of things, you certainly want to declare it
15 `constexpr`, because then compilers will ensure that it has a compile-time value:

```
16 int sz;                                // non-constexpr variable
17 ...
18 constexpr auto arraySize1 = sz;          // error! sz's value not
19                               // known at compilation
20 std::array<int, sz> data1;              // error! same problem
21 constexpr auto arraySize2 = 10;           // fine, 10 is a
22                               // compile-time constant
23 std::array<int, arraySize2> data2;       // fine, arraySize
24                               // is constexpr
```

25 Note that `const` doesn't offer the same guarantee as `constexpr`, because `const`
26 objects need not be initialized with values known during compilation:

```
27 int sz;                                // as before
28 ...
29 const auto arraySize = sz;               // fine, arraySize is
30                               // const copy of sz
31 std::array<int, arraySize> data;        // error! arraySize's value
32                               // not known at compilation
```

1 Simply put, all `constexpr` objects are `const`, but not all `const` objects are `constexpr`. If you want compilers to guarantee that a variable has a value that can be
2 used in contexts requiring compile-time constants, the tool to reach for is `constexpr`, not `const`.

5 Usage scenarios for `constexpr` objects become more interesting when `constexpr`
6 functions are involved. Such functions produce compile-time constants
7 *when they are called with compile-time constants*. If they're called with values not
8 known until runtime, they produce runtime values. This may sound as if you don't
9 know what they'll do, but that's the wrong way to think about it. The right way to
10 view it is this:

11 • `constexpr` functions can be used in contexts that demand compile-time con-
12 stants. If the values of the arguments you pass to a `constexpr` function in such
13 a context are known during compilation, the result will be computed during
14 compilation. If any of the arguments' values is not known during compilation,
15 your code will be rejected.

16 • When a `constexpr` function is called with one or more values that are not
17 known during compilation, it acts like a normal function, computing its result
18 at runtime. This means you don't need two functions to perform the same op-
19 eration, one for compile-time constants and one for all other values. The `con-`
20 `constexpr` function does it all.

21 Suppose we need a data structure to hold the results of an experiment that can be
22 run in a variety of ways. For example, the lights can be on or off as the experiment
23 runs, as can the heater or the wind machine, etc. If there are n binary conditions
24 applicable to the running of the experiment, the number of combinations is 2^n , so
25 we need a data structure with enough room for 2^n values. Assuming each result is
26 an `int` and that n is known (or can be computed) during compilation, a
27 `std::array` could be a reasonable data structure choice. But we'd need a way to
28 compute 2^n during compilation. The C++ Standard Library provides `std::pow`,
29 which is the mathematical functionality we need, but, for our purposes, there are
30 two problems with it. First, `std::pow` works on floating point types, and we need
31 an integral result. Second, `std::pow` isn't `constexpr` (i.e., isn't guaranteed to re-

1 turn a compile-time result when called with compile-time values), so we can't use
2 it to specify the size of a `std::array`.

3 Fortunately, we can write the `pow` we need. I'll show how to do that in a moment,
4 but first let's look at how it'd be declared and used:

```
5 constexpr int pow(int base, int exp) // pow's a constexpr func
6 {
7     ...
8 } // impl is below
9 constexpr auto numConds = 5; // # of conditions
10 std::array<int, pow(2, numConds)> results; // results has
11 //  $2^{\text{numConds}}$  elements
```

12 Recall that the `constexpr` in front of `pow` doesn't say that `pow` returns a `const`
13 value, it says that if `base` and `exp` are compile-time constants, `pow`'s result may be
14 used as a compile-time constant. If `base` and/or `exp` are not compile-time con-
15 stants, `pow`'s result will be computed at runtime. That means that `pow` can not only
16 be called to do things like compile-time-compute the size of a `std::array`, it can
17 also be called in runtime contexts such as this:

```
18 auto base = readFromDB("base"); // get these values
19 auto exp = readFromDB("exponent"); // at runtime
20 auto baseToExp = pow(base, exp); // call pow function
21 // at runtime
```

22 Because `constexpr` functions must be able to return compile-time results when
23 called with compile-time values, restrictions are imposed on their implementation.
24 Among these is that, in C++11, there may be only a single executable statement: a
25 `return`. That sounds more limiting than it is, because two tricks can be used to
26 extend the expressiveness of `constexpr` functions beyond what you might think.
27 First, the conditional “`? :`” operator can be used in place of `if-else` statements,
28 and second, recursion can be used instead of loops. `pow` can therefore be imple-
29 mented like this:

```
30 constexpr int pow(int base, int exp)
31 {
32     return (exp == 0 ? 1 // if (exp==0) return 1
```

```
1      : base * pow(base, exp - 1)); // else return  
2 } // base*baseexp-1
```

3 This works, but it's hard to imagine that anybody except a hard-core functional
4 programmer would consider it pretty. In C++14, the restrictions on `constexpr`
5 functions are substantially looser, so the following implementation is valid in
6 C++14:

```
7 constexpr int pow(int base, int exp) // C++14 only  
8 {  
9     if (exp == 0) return 1;  
10    auto result = base;  
11    for (int i = 1; i < exp; ++i) result *= base;  
12    return result;  
13 }
```

14 `constexpr` functions are limited to taking and returning *literal types*, which essen-
15 tially means types that can have values determined during compilation. All built-in
16 types qualify, but user-defined types may be literal, too, because constructors and
17 other member functions may be `constexpr`:

```
18 class Point {  
19 public:  
20     constexpr Point(double xVal, double yVal)  
21     : x(xVal), y(yVal)  
22     {}  
23     constexpr double xVal() const { return x; }  
24     constexpr double yVal() const { return y; }  
25     void setX(double newX) { x = newX; }  
26     void setY(double newY) { y = newY; }  
27 private:  
28     double x, y;  
29 };
```

30 Here, the `Point` constructor can be declared `constexpr`, because if the arguments
31 passed to it are known during compilation, the value of the data members of the
32 constructed `Point` can also be known during compilation. Points so initialized
33 could thus be `constexpr`:

```
34 constexpr Point p1 { 1, 1 }; // fine, "runs" constexpr  
35 // ctor during compilation
```

1 **constexpr** Point p2 { 4, 2 }; // also fine
2 Similarly, the getters **xVal** and **yVal** can be **constexpr**, because if they're invoked
3 on a **Point** object with a value known during compilation (e.g., a **constexpr**
4 **Point** object), the values of the data members **x** and **y** can be known during com-
5 pilation. That makes it possible to write **constexpr** functions that call **Point**'s
6 getters and to initialize **constexpr** objects with the results of such functions:

```
7   constexpr Point midpoint(const Point& p1, const Point& p2)  
8  {  
9    return { (p1.xVal() + p2.xVal()) / 2,     // call constexpr  
10               (p1.yVal() + p2.yVal()) / 2 }; // member functions  
11 }  
  
12 constexpr auto mid = midpoint(p1, p2);        // init constexpr  
13                                                      // object w/result of  
14                                                      // constexpr function
```

15 This is very exciting. It means that the object **mid**, though its initialization involves
16 calls to constructors, getters, and a non-member function, can be created in read-
17 only memory! It means you could use an expression like **mid.xVal()** * 10 as an
18 argument to a template or to specify the value of an enumerator! It means that the
19 traditionally fairly strict line between work done during compilation and work
20 done at runtime begins to blur, and in turn some computations traditionally done
21 at runtime can migrate to compile-time. The more code taking part in the migra-
22 tion, the faster your software will run. (Compilation may take longer, however.)

23 The setter functions **setX** and **setY** can't be declared **constexpr**, because, among
24 other things, they contain an assignment, and assignments are not permitted in
25 **constexpr** functions, not even in C++14. (For a detailed treatment of the re-
26 strictions on **constexpr** function implementations, consult your favorite refer-
27 ences for C++11 and C++14, bearing in mind that C++11 imposes more constraints
28 than C++14.)

29 The advice of this Item is to use **constexpr** whenever possible, and by now I hope
30 it's clear why: both **constexpr** objects and **constexpr** functions can be employed
31 in a wider range of contexts than non-**constexpr** objects and functions. By using
32 **constexpr** whenever possible, you maximize the range of situations in which
33 your objects and functions may be used.

1 It's important to note that `constexpr` is part of an object's or function's interface.
2 `constexpr` proclaims "I can be used in a context where C++ requires a constant
3 expression." If you declare an object or function `constexpr`, clients may use it in
4 such contexts. If you later decide that your use of `constexpr` was a mistake and
5 you remove it, you may cause arbitrarily large amounts of client code to stop com-
6 piling. (The simple act of adding I/O to a function for debugging or performance
7 tuning could lead to such a problem, because I/O statements are generally not
8 permitted in `constexpr` functions.) Part of "whenever possible" in "Use `con-
9 stexpr whenever possible" is your willingness to make a long-term commitment
10 to the constraints it imposes on the objects and functions you apply it to.`

11 Things to Remember

- 12 ♦ `constexpr` objects are `const` and are initialized with values known during
13 compilation.
- 14 ♦ `constexpr` functions produce compile-time results when called with argu-
15 ments whose values are known during compilation.
- 16 ♦ `constexpr` objects and functions may be used in a wider range of contexts
17 than non-`constexpr` objects and functions.
- 18 ♦ `constexpr` is part of an object's or function's interface.

19 Item 15: Make `const` member functions thread-safe.

20 If we're working in a mathematical domain, we might find it convenient to have a
21 class representing polynomials. Within this class, it would probably be useful to
22 have a function to compute the root(s) of the polynomial, i.e., values where the
23 polynomial evaluates to zero. Such a function would not modify the polynomial, so
24 it'd be natural to declare it `const`:

```
25 class Polynomial {  
26 public:  
27     using RootsType =           // data structure holding values  
28         std::vector<double>;   // where polynomial evals to zero  
29     ...                      // (see Item 9 for info on "using")  
30     RootsType roots() const;  
31     ...
```

1 };
2 Computing the roots of a polynomial can be expensive, so we don't want to do it if
3 we don't have to. And if we do have to do it, we certainly don't want to do it more
4 than once. We'll thus cache the root(s) of the polynomial if we have to compute
5 them, and we'll implement `roots` to return the cached value. Here's the basic ap-
6 proach:

```
7 class Polynomial {  
8 public:  
9     using RootsType = std::vector<double>;  
10    RootsType roots() const  
11    {  
12        if (!rootsAreValid) {           // if cache not valid  
13            ...                         // compute roots,  
14            // store them in rootVals  
15            rootsAreValid = true;  
16        }  
17        return rootVals;  
18    }  
19 private:  
20     mutable bool rootsAreValid { false };   // see Item 7 for info  
21     mutable RootsType rootVals {};          // on initializers  
22};
```

23 Conceptually, `roots` doesn't change the `Polynomial` object on which it operates,
24 but, as part of its caching activity, it may need to modify `rootVals` and `rootsAre-`
25 `Valid`. That's a classic use case for `mutable`, and that's why it's part of the declara-
26 tions for these data members.

27 Imagine now that two threads simultaneously call `roots` on a `Polynomial` object:

```
28 Polynomial p;  
29  
30 /*----- Thread 1 ----- */      /*----- Thread 2 ----- */  
31 auto rootsOfP = p.roots();       auto valsGivingZero = p.roots();
```

32 This client code is perfectly reasonable. `roots` is a `const` member function, and
33 that means it represents a read operation. Having multiple threads perform a read
34 operation without synchronization is safe. At least it's supposed to be. In this case,

1 it's not, because inside `roots`, one or both of these threads might try to modify the
2 data members `rootsAreValid` and `rootVals`. That means that this code could
3 have different threads reading and writing the same memory without synchroni-
4 zation, and that's the definition of a data race. This code has undefined behavior.

5 But there's still nothing wrong with the client code. Clients are supposed to be able
6 to rely on simultaneous reads being safe, and declaring a member function `const`
7 is tantamount to proclaiming it a read operation. The notion that `const` member
8 functions may be safely invoked without synchronization is so deeply engrained in
9 C++11, the Standard Library takes it for granted. If you use any part of the Stand-
10 ard Library on a type where a `const` member function can't safely be called with-
11 out synchronization, your program has undefined behavior.

12 The problem here is that `roots` is declared `const`, but it's not thread-safe. The
13 `const` declaration is as correct in C++11 as it would be in C++98 (retrieving the
14 roots of a polynomial doesn't change the value of the polynomial), so what re-
15 quires rectification is the lack of thread safety.

16 The easiest way to address the issue is the usual one: employ a `mutex`:

```
17 class Polynomial {
18 public:
19     using RootsType = std::vector<double>;
20
21     RootsType roots() const
22     {
23         std::lock_guard<std::mutex> g(m);           // Lock mutex
24
25         if (!rootsAreValid) {                         // if cache not valid
26             ...
27             // compute/store roots
28
29             rootsAreValid = true;
30         }
31
32         return rootVals;                           // release mutex
33     }
34
35 private:
36     mutable std::mutex m;
37     mutable bool rootsAreValid { false };
38     mutable RootsType rootVals {};
39 };
```

1 The `std::mutex`, `m`, is declared `mutable`, because locking and unlocking it are
2 non-`const` member functions, and within `roots` (a `const` member function), `m`
3 would otherwise be considered a `const` object.

4 It's hard to go wrong using a mutex in this kind of context, but sometimes a mutex
5 is overkill. For example, if all you're doing is counting how many times a member
6 function is called, an atomic variable will often be a less expensive way to go.
7 (Whether it actually is less expensive depends on the hardware you're running on
8 and the implementation of mutexes in your Standard Library.) Here's how you can
9 count calls using an atomic variable:

```
10 class Point {                                     // 2D point
11 public:
12 ...
13     double distanceFromOrigin() const
14     {
15         ++callCount;                           // atomic increment
16         return std::sqrt((x * x) + (y * y));
17     }
18 private:
19     mutable std::atomic<unsigned> callCount { 0 };
20     double x, y;
21 };
```

22 All operations on atomic types in the Standard Library are guaranteed to be atomic
23 (i.e., indivisible as observed by other threads), so even though the increment of
24 `callCount` is a read-modify-write operation, you can rely on it proceeding atomi-
25 cally.

26 In view of the fact that operations on atomic variables are often less expensive
27 than mutex acquisition and release, you may be tempted to lean on atomic varia-
28 bles more heavily than you should. For example, in a class caching an expensive-
29 to-compute `int`, you might try to use a pair of atomic variables instead of a mutex:

```
30 class Widget {
31 public:
32 ...
33     int magicValue() const
34     {
```

```

1     if (cacheValid) return cachedValue;
2     else {
3         auto val1 = expensiveComputation1();
4         auto val2 = expensiveComputation2();
5         cachedValue = val1 + val2;           // uh oh, part 1
6         cacheValid = true;                  // uh oh, part 2
7         return cachedValue;
8     }
9 }

10 private:
11     mutable std::atomic<bool> cacheValid { false };
12     mutable std::atomic<int> cachedValue;
13 
```

14 This will work, but sometimes it will work a lot harder than it should. Consider:

- 15 • A thread calls `Widget::magicValue`, sees `cacheValid` as `false`, performs
16 the two expensive computations and assigns their sum to `cachedValue`.
- 17 • At that point, a second thread calls `Widget::magicValue`, also sees `ca-`
18 `cheValid` as `false` and thus carries out the same expensive computations
19 that the first thread has just finished performing. (This “second thread” may in
20 fact be *several* other threads.)

21 Such behavior is contrary to the goal of caching. Reversing the order of the as-
22 signments to `cachedValue` and `CacheValid` eliminates that problem, but the re-
23 sult is even worse:

```

24 class Widget {
25     public:
26     ...
27     int magicValue() const
28     {
29         if (cacheValid) return cachedValue;
30         else {
31             auto val1 = expensiveComputation1();
32             auto val2 = expensiveComputation2();
33             cacheValid = true;           // uh oh, part 1
34             return cachedValue = val1 + val2; // uh oh, part 2
35         }
36     }
37     ...
38 
```

1 Imagine that `cacheValid` is `false`, and then:

- 2 • One thread calls `Widget::magicValue` and executes through the point where
3 `cacheValid` is set to true.
- 4 • At that moment, a second thread calls `Widget::magicValue` and checks `ca-`
5 `cheValid`. Seeing it `true`, the thread returns `cachedValue`, even though the
6 first thread has not yet made an assignment to it. The returned value is there-
7 fore incorrect.

8 There's a lesson here. For a single variable or memory location requiring synchro-
9 nization, use of a `std::atomic` is often adequate, but once you get to two or more
10 variables or memory locations (e.g., distinct elements of a data structure) that re-
11 quire manipulation as a unit, you should reach for a mutex. For `Widg-`
12 `et::magicValue`, that would look like this:

```
13 class Widget {
14 public:
15 ...
16     int magicValue() const
17     {
18         std::lock_guard<std::mutex> guard(m); // lock m
19         if (cacheValid) return cachedValue;
20         else {
21             auto val1 = expensiveComputation1();
22             auto val2 = expensiveComputation2();
23             cachedValue = val1 + val2;
24             cacheValid = true;
25             return cachedValue;
26         }
27     } // unlock m
28 ...
29 private:
30     mutable std::mutex m;
31     mutable int cachedValue; // no longer atomic
32     mutable bool cacheValid { false }; // no longer atomic
33 };
```

34 Regardless of how you make your `const` member functions thread-safe, it's essen-
35 tial that you do. Callers of `const` member functions have a right to assume that
36 their calls will succeed without their performing any synchronization. Implemen-

1 tations of `const` member functions achieve thread safety either by being bitwise
2 `const` (i.e., not modifying any bits making up the value of the object) or by using
3 some kind of internal synchronization, e.g., a mutex or an atomic variable.

4 **Things to Remember**

- 5 ♦ Make `const` member functions thread safe.
6 ♦ Use of atomic variables may offer better performance than a mutex, but they're
7 generally only suited for manipulation of a single variable or memory location.

8 **Item 16: Declare functions `noexcept` whenever possible.**

9 In C++98, exception specifications were rather temperamental creatures. You had
10 to summarize the exception types a function might emit, so if the function's im-
11 plementation was modified, the exception specification might need revision, too.
12 Changing an exception specification could break client code, because callers might
13 be dependent on the original exception specification. Compilers typically offered
14 no help in maintaining consistency among function implementations, exception
15 specifications, and client code. Most programmers ultimately decided that C++98
16 exception specifications weren't worth the trouble.

17 Interest in the idea of exception specifications remained strong, however, and as
18 work on C++ progressed, a consensus emerged that the truly meaningful infor-
19 mation about a function's exception-emitting behavior was whether it had any.
20 Black or white, either a function might emit an exception or it guaranteed that it
21 wouldn't. This maybe-or-never dichotomy forms the basis of C++11's exception
22 specifications, which essentially replace C++98's. (C++98-style exception specifica-
23 tions remain valid, but they're deprecated.) In C++11, `noexcept` is for functions
24 that guarantee they won't emit an exception.

25 Whether a function should be so declared is fundamentally a matter of interface
26 design. The exception-emitting behavior of a function is of key interest to clients.
27 Callers can query a function's `noexcept` status, and the results of such a query can
28 affect the exception safety or efficiency of the calling code. As such, whether a func-
29 tion is `noexcept` is as important a piece of information as whether a member func-

1 tion is `const`. Failure to declare a function `noexcept` when you know that it will
2 never emit an exception is simply poor interface specification.

3 But there's an additional incentive to apply `noexcept` to functions that won't pro-
4 duce exceptions: it permits compilers to generate better object code. To under-
5 stand why, it helps to examine the difference between the C++98 and C++11 ways
6 of saying that a function won't emit exceptions. Consider a function `f` that promis-
7 es callers they'll never receive an exception. The two ways of expressing that are:

8

```
int f(int x) throw();      // no exceptions from f: C++98 style
```


9

```
int f(int x) noexcept;    // no exceptions from f: C++11 style
```

10 If, at runtime, an exception leaves `f`, `f`'s exception specification is violated. With
11 the C++98 approach, the call stack is unwound to `f`'s caller, and, after some actions
12 not relevant here, program execution is terminated. With the C++11 approach,
13 runtime behavior is a bit different: the stack is only *possibly* unwound before pro-
14 gram execution is terminated.

15 The difference between unwinding the call stack and *possibly* unwinding it has a
16 surprisingly large impact on code generation. In a `noexcept` function, optimizers
17 need not keep the runtime stack in an unwindable state if an exception would
18 propagate out of the function, nor must they ensure that objects in a `noexcept`
19 function are destroyed in the inverse order of construction should an exception
20 leave the function. The result is more opportunities for optimization, not only
21 within the body of a `noexcept` function, but also at sites where the function is
22 called. Such flexibility is present only for `noexcept` functions. Functions with
23 “`throw()`” exception specifications lack it, as do functions with no exception speci-
24 fication at all. The situation can be summarized this way:

25

```
ReturnType function(params) noexcept;      // most optimizable
```


26

```
ReturnType function(params) throw();        // less optimizable
```


27

```
ReturnType function(params);                // less optimizable
```

28 This alone should provide sufficient motivation to declare functions `noexcept`
29 whenever you can.

1 For some functions, the case is even stronger. The move operations are the
2 preeminent example. Suppose you have a C++98 code base making use of
3 `std::vectors` of `Widgets`. `Widgets` are added to the `std::vectors` from time to
4 time, perhaps via `push_back`:

```
5 std::vector<Widget> vw;  
6 ...  
7 Widget w;  
8 ... // work with w  
9 vw.push_back(w); // add w to vw  
10 ...
```

11 Assume this code works fine, and you have no interest in modifying it for C++11.
12 However, you do want to take advantage of the fact that C++11's move semantics
13 can improve the performance of legacy code when move-enabled types are in-
14 volved. You therefore ensure that `Widget` has move operations, either by writing
15 them yourself or by seeing to it that the conditions for their automatic generation
16 are fulfilled (see Item 19).

17 When a new element is added to a `std::vector` via `push_back`, it's possible that
18 the `std::vector` lacks space for it, i.e., that the `std::vector`'s size is equal to its
19 capacity. When that happens, the `std::vector` allocates a new, larger, chunk of
20 memory to hold its elements, and it transfers the elements from the existing chunk
21 of memory to the new one. In C++98, the transfer was accomplished by copying
22 each element from the old memory to the new memory, then destroying the origi-
23 nals in the old memory. This approach enabled `push_back` to offer the strong ex-
24 ception safety guarantee: if an exception was thrown during the copying of the el-
25 ements, the state of the `std::vector` remained unchanged, because none of the
26 elements in the original memory was destroyed until all elements had been suc-
27 cessfully copied into the new memory.

28 In C++11, a natural optimization would be to replace the copying of `std::vector`
29 elements with moves. Unfortunately, doing this runs the risk of violating
30 `push_back`'s exception safety guarantee. If n elements have been moved from the
31 old memory and an exception is thrown moving element $n+1$, the `push_back` op-

1 eration can't run to completion. But the original `std::vector` has been modified:
2 n of its elements have been moved from. Restoring their original state may not be
3 possible, because attempting to move each object back into the original memory
4 may itself yield an exception.

5 This is a serious problem, because the behavior of legacy code could depend on
6 `push_back`'s strong exception safety guarantee. Therefore, C++11 implementa-
7 tions can't silently replace copy operations inside `push_back` with moves. They
8 must continue to employ copy operations. *Unless*, that is, it's known that the move
9 operations are guaranteed not to emit exceptions. In that case, replacing element
10 copy operations inside `push_back` with move operations would be safe, and the
11 only side effect would be improved performance.

12 `std::vector::push_back` takes advantage of this "move if you can, but copy if
13 you must" strategy, and it's not the only function in the Standard Library that does.
14 Other functions sporting the strong exception safety guarantee in C++98 (e.g.,
15 `std::vector::reserve`, `std::deque::insert`, etc.) behave the same way. All
16 these functions replace calls to copy operations in C++98 with calls to move opera-
17 tions in C++11 if (and only if) the move operations are known to not emit excep-
18 tions. But how can a function know if a move operation won't produce an excep-
19 tion? The answer is obvious: it checks to see if the operation is declared `noex-
20 cept`.*

21 `swap` functions comprise another case where `noexcept` is particularly desirable.
22 `swap` is a key component of many STL algorithm implementations, and it's com-
23 monly employed in copy assignment operators, too. Its widespread use renders
24 the optimizations that `noexcept` affords especially worthwhile. Furthermore,

* The checking is typically rather roundabout. Functions like `std::vector::push_back` call `std::move_if_noexcept`, a variation of `std::move` that conditionally casts to an rvalue (see Item 25), depending on whether the type's move constructor is `noexcept`. In turn, `std::move_if_noexcept` calls `std::is_nothrow_move_constructible`, and the value of this type trait is set by compilers, based on whether the move constructor has a `noexcept` (or `throw()`) designation.

1 whether `swaps` in the Standard Library are `noexcept` is sometimes dependent on
2 whether user-defined `swaps` are `noexcept`. For example, the declarations for the
3 Standard Library's `swaps` for arrays and for `std::pair` are:

```
4 template <class T, size_t N>
5 void swap(T (&a)[N],
6           T (&b)[N]) noexcept(noexcept(swap(*a, *b)));
7
8 template <class T1, class T2>
9 struct pair {
10 ...
11     void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
12                               noexcept(swap(second, p.second)));
13 }
```

14 These functions are *conditionally noexcept*: whether they are `noexcept` depends
15 on whether the expressions inside the `noexcepts` are `noexcept`. Given two arrays
16 of `Widget`, for example, swapping them is `noexcept` only if swapping individual
17 elements from the arrays is `noexcept`, i.e., if `swap` for `Widget` is `noexcept`. The
18 author of `Widget`'s `swap` thus determines whether swapping arrays of `Widget` is
19 `noexcept`. That, in turn, determines whether other `swaps`, such as the one for ar-
20 rays of arrays of `Widget`, are `noexcept`. Similarly, whether swapping two
21 `std::pair` objects containing `Widgets` is `noexcept` depends on whether `swap` for
22 `Widgets` is `noexcept`. The fact that swapping higher-level data structures can
23 generally be `noexcept` only if swapping their lower-level constituents is `noex-`
24 `cept` is the reason why you should strive to offer `noexcept swap` functions.

25 By now, I hope you're excited about the optimization opportunities that `noexcept`
26 affords. Alas, I must temper your enthusiasm. Optimization is important, but cor-
27 rectness is more important. I noted at the beginning of this Item that `noexcept` is
28 part of a function's interface, so you should declare a function `noexcept` only if
29 you are willing to commit to a `noexcept` implementation over the long term. If
30 you declare a function `noexcept` and later regret that decision, your options are
31 bleak. You can remove `noexcept` from the function's declaration (i.e., change its
32 interface), thus running the risk of breaking client code. You can change the im-
33 plementation such that an exception could escape, but keep the original (now in-
34 correct) exception specification. If you do that, your program will be terminated if

1 an exception tries to leave the function. Or you can resign yourself to your existing
2 implementation, abandoning whatever motivated your desire to change the im-
3 plementation in the first place. None of these options is appealing.

4 The fact of the matter is that most functions are *exception-neutral*. Such functions
5 throw no exceptions themselves, but functions they call might emit one. When that
6 happens, the calling function allows the emitted exception to pass through on its
7 way to a handler further up the call chain. Exception-neutral functions are never
8 `noexcept`, because they may emit such “just passing through” exceptions. Most
9 functions, therefore, quite properly lack the `noexcept` designation.

10 Some functions, however, have natural implementations that emit no exceptions,
11 and for a few more—notably the move operations and `swap`—being `noexcept` has
12 such a significant payoff, it’s worth implementing them in a `noexcept` manner if at
13 all possible.[†] When you can honestly say that a function should never emit excep-
14 tions, you should definitely declare it `noexcept`.

15 Please note that I said some functions have *natural* `noexcept` implementations.
16 Twisting a function’s implementation to permit a `noexcept` declaration is the tail
17 wagging the dog. Is putting the cart before the horse. Is not seeing the forest for
18 the trees. Is...choose your favorite metaphor. If a straightforward function imple-
19 mentation might yield exceptions (e.g., by invoking a function that might throw),
20 the hoops you’ll jump through to hide that from callers (e.g., catching all excep-
21 tions and replacing them with status codes or special return values) will not only
22 complicate your function’s implementation, it will typically complicate code at call
23 sites, too (e.g., code there may have to check for status codes or special return val-

[†] The prescribed declarations for move operations on containers in the Standard Library lack `noexcept`. However, implementers are permitted to strengthen exception specifications for Standard Library functions, and, in practice, it is common for at least some container move operations to be declared `noexcept`. That practice exemplifies this Item’s advice. Having found that it’s possible to write container move operations such that exceptions never need to be emitted, implementers often declare the operations `noexcept`, even though the Standard does not require them to do so.

1 ues). The runtime cost of those complications (e.g., extra branches, larger functions
2 that put more pressure on instruction caches, etc.) could exceed any speedup
3 you'd hope to achieve via `noexcept`, plus you'd be saddled with source code that's
4 more difficult to comprehend and maintain. That'd hardly be exemplary software
5 engineering. As a general rule, the only time it makes sense to actively search for a
6 `noexcept` algorithm is when you're implementing the move functions or `swap`.

7 Two more points about `noexcept` functions are worth mentioning. First, in C++98,
8 it was considered bad style to permit the memory deallocation functions (i.e., op-
9 erator `delete` and operator `delete[]`) and destructors to emit exceptions, and
10 in C++11, this style rule has been all but upgraded to a language rule. By default, all
11 memory deallocation functions and all destructors—both user-defined and com-
12 piler-generated—are implicitly `noexcept`. There's thus no need to declare them
13 `noexcept`. (Doing so doesn't hurt anything, it's just unconventional.) The only
14 time a destructor is not implicitly `noexcept` is when a data member of the class
15 (including inherited members and those contained inside other data members) is
16 of a type that expressly states that its destructor may emit exceptions (e.g., de-
17 clares it "`noexcept(false)`"). Such destructors are uncommon. There are none in
18 the Standard Library.

19 Second, let me elaborate on my earlier observation that compilers typically offer
20 no help in identifying inconsistencies between function implementations and their
21 exception specifications. Consider this code, which is perfectly legal:

```
22 void setup();           // functions defined elsewhere
23 void cleanup();
24 void doWork() noexcept
25 {
26     setup();           // set up work to be done
27     ...                // do the actual work
28     cleanup();         // perform cleanup actions
29 }
```

30 Here, `doWork` is declared `noexcept`, even though it calls the non-`noexcept` func-
31 tions `setup` and `cleanup`. This seems contradictory, but it could be that `setup`
32 and `cleanup` document that they never emit exceptions, even though they're not

1 declared that way. There could be good reasons for their non-noexcept declarations.
2 For example, they might be part of a library written in C. (Even functions
3 from the C Standard Library that have been moved into the `std` namespace lack
4 exception specifications, e.g., `std::strlen` isn't declared `noexcept`.) Or they
5 could be part of a C++98 library that decided not to use C++98 exception specifications
6 and hasn't yet been revised for C++11.

7 Because there are legitimate reasons for `noexcept` functions to rely on code lacking
8 the `noexcept` guarantee, C++ permits such code, and compilers generally don't
9 issue warnings about it.

10 **Things to Remember**

- 11 ♦ `noexcept` is part of a function's interface, so callers may depend on it.
- 12 ♦ `noexcept` functions are more optimizable than non-`noexcept` functions.
- 13 ♦ `noexcept` is particularly valuable for the move operations and for `swap`.
- 14 ♦ Most functions are exception-neutral rather than `noexcept`.

15 **Item 17: Consider pass by value for cheap-to-move parameters that are always copied.**

17 Some functions take parameters that are, at least under normal circumstances, always copied. Setters are a good example. Setter functions typically take a value and store it in a data member of an object. For such functions to operate with maximal efficiency, they should copy lvalue arguments and move rvalue arguments:

```
21 class Widget {  
22 public:  
23     void setName(const std::string& newName)      // take lvalue;  
24     { name = newName; }                            // copy it  
25     void setName(std::string&& newName)           // take rvalue;  
26     { name = std::move(newName); }                  // move it  
27     ...  
28 private:  
29     std::string name;  
30 };
```

1 This works, but it requires writing two functions that do essentially the same
2 thing. That chafes a bit: two functions to declare, two functions to implement, two
3 functions to document, two functions to maintain. Ugh.

4 Furthermore, there will be two functions in the object code—something you might
5 care about if you're concerned about your program's footprint. In this case, both
6 functions will probably be inlined, and that's likely to eliminate any bloat issues
7 related to the existence of two functions, but if these functions aren't inlined eve-
8 rywhere, you really will get two functions in your object code. (Inlining might not
9 occur due to the creation of pointers to these functions, calls to these functions oc-
10 curring inside complicated calling contexts, or building the system with inlining
11 disabled.)

12 An alternative approach is to make `setName` a template function taking a universal
13 reference (see Item 26):

```
14 class Widget {  
15 public:  
16     template<typename T>  
17     void setName(T&& newName)           // take lvalues and  
18     { name = std::forward<T>(newName); } // rvalues; copy  
19                                         // lvalues, move rvalues  
20 ...  
21 };
```

22 This reduces the source code you have to deal with, though, assuming this function
23 is called with both lvalues and rvalues, there will still be two functions in the ob-
24 ject code. (The template will be instantiated differently for lvalues and rvalues.)
25 Furthermore, the use of universal references has drawbacks. As Item 32 explains,
26 not all argument types can be passed via universal reference, and, as noted in
27 Item 29, if clients pass improper argument types to functions taking universal ref-
28 erences, compiler error messages can be, er, challenging.

29 Wouldn't it be nice if there were a way to write functions like `setName` such that
30 lvalues were copied, rvalues were moved, there was only one function to deal with
31 (in both source and object code), and the idiosyncrasies of universal references
32 were avoided? As it happens, there is. All you have to do is abandon one of the very
33 first rules you probably learned as a C++ programmer. That rule was to avoid pass-

1 ing objects by value. For parameters like `newName` in functions like `setName`, pass
2 by value may be exactly what you want.

3 Before we discuss *why* pass-by-value is probably a good fit for `newName` and `set-`
4 `Name`, let's see how it would be implemented.

```
5 class Widget {  
6 public:  
7     void setName(std::string newName)      // take lvalue or  
8     { name = std::move(newName); }          // rvalue; move it  
9  
10    ...  
11};
```

11 The only non-obvious part of this code is the application of `std::move` to the pa-
12 rameter, `newName`. Typically, `std::move` is used with rvalue references (see
13 Item 27), but in this case, we know that (1) `newName` is a completely independent
14 object from whatever the caller passed in, so changing `newName` won't affect call-
15 ers and (2) this is the final use of `newName`, so moving from it won't have any im-
16 pact on the rest of the function.

17 The fact that there's only one `setName` function explains how we avoid code dupli-
18 cation—both in the source file as well as the corresponding object file. We're not
19 using a universal reference, so this approach doesn't lead to odd failure cases or
20 confounding error messages. The question remaining regards the efficiency of this
21 design. We're passing *by value*. Isn't that expensive?

22 In C++98, it was a reasonable bet that it was, because no matter what callers
23 passed in, the parameter `newName` would be created by *copy construction*. In
24 C++11, `newName` will be copy-constructed only for lvalues. For rvalues, it will be
25 *move-constructed*. Here, look:

```
26 Widget w;  
27 ...  
28 std::string widgetID("Bart");  
29 w.setName(widgetID);           // call setName with lvalue  
30 ...
```

```
1 w.setName(widgetID + "Jenne");           // call setName with rvalue
2                                         // (see below)

3 In the first call to setName (when widgetID is passed), the parameter newName is
4 initialized with an lvalue. newName is thus copy-constructed, just like it would be in
5 C++98. In the second call, newName is initialized with the std::string object re-
6 sulting from a call to operator+ for std::string (i.e., the append operation).
7 That object is an rvalue, and newName is therefore move-constructed.

8 In sum, when callers pass lvalues, they're copied into newName, and when callers
9 pass rvalues, they're moved into newName, just like we want. Neat, huh?
```

10 “It can’t be that simple,” I hear you thinking. “There’s got to be a catch.” Generally,
11 there’s not, but there are some conditions you need to keep in mind. Doing that
12 will be easier if we recap the three versions of setName we’ve considered:

```
13 class Widget {                                // Approach 1:
14 public:                                         // overload for
15     void setName(const std::string& newName)    // lvalues and
16     { name = newName; }                         // rvalues
17
18     void setName(std::string&& newName)
19     { name = std::move(newName); }
20 ...
21
22 private:
23     std::string name;
24 };
25
26 class Widget {                                // Approach 2:
27 public:                                         // use universal
28     template<typename T>                      // reference
29     void setName(T&& newName)
30     { name = std::forward<T>(newName); }
31 ...
32
33 class Widget {                                // Approach 3:
34 public:                                         // pass by value
35     void setName(std::string newName)
36     { name = std::move(newName); }
37 ...
38 }
```

36 And here are the two calling scenarios we’ve examined:

```

1 Widget w;
2 ...
3 std::string widgetID("Bart");
4 w.setName(widgetID); // pass lvalue
5 ...
6 w.setName(widgetID + "Jenne"); // pass rvalue
7 Now consider the cost, in terms of copy and move operations, of setting a Widget's
8 name for the two calling scenarios and each of the three setName implementations
9 we've discussed:
10 • Overloading: Regardless of whether an lvalue or an rvalue is passed, the caller's argument is bound to a reference called newName. That costs nothing, in terms of copy and move operations. In the lvalue overload, newName is copied into Widget::name. In the rvalue overload, it's moved. Cost summary: 1 copy for lvalues, 1 move for rvalues.
11 • Using a universal reference: As with overloading, the caller's argument is bound to the reference newName. This is a no-cost operation. Due to the use of std::forward (see Item 25) lvalue arguments are copied into Widget::name, while rvalue arguments are moved. The cost summary is the same as with overloading: 1 copy for lvalues, 1 move for rvalues. This is to be expected. A template taking a universal reference parameter instantiates into two functions, one taking an lvalue reference parameter and one taking its rvalue reference counterpart.
12 • Passing by value: Regardless of whether an lvalue or an rvalue is passed, the parameter newName must be constructed. If an lvalue is passed, this costs a copy operation. If an rvalue is passed, it typically costs a move. In the body of the function, newName is then unconditionally moved into Widget::name. The cost summary is thus 1 copy plus 1 move for lvalues, and 2 moves for rvalues.
13 Look again at this Item's title:
14 Consider pass by value for cheap-to-move parameters that are always copied.
15 It's worded the way it is for a reason. Three reasons, in fact.

```

1 First, you should only *consider* using pass by value. Yes, it requires writing only
2 one function. Yes, it generates only one function in the object code. Yes, it avoids
3 the interface issues associated with universal references (i.e., failure cases and un-
4 pleasant error messages). But it has a higher cost than the alternatives. In particu-
5 lar, it costs an extra move for both lvalue and rvalue arguments.

6 Which brings me to the second caveat. Consider pass by value only *for cheap-to-*
7 *move parameters*. When moves are cheap, the cost of the extra move is likely to be
8 negligible. But when moves are not cheap, that extra move can incur an expense
9 you can't afford. Item 31 explains that not all types are cheap to move—not even
10 all types in the Standard Library. When moves are not cheap, performing an un-
11 necessary move is analogous to performing an unnecessary copy, and the im-
12 portance of avoiding unnecessary copy operations is what lead to the C++98 rule
13 about avoiding pass by value in the first place!

14 Finally, you should consider pass by value only for parameters that are *always copied*. For setter functions (as well as for constructors taking initialization arguments
15 for an object's data members), this condition is typically fulfilled, but consider a
16 function that adds a value to a data structure only if the value satisfies a constraint.
17 Using pass by value, it could be written like this:

```
19 class Widget {  
20 public:  
21     bool insert(std::string s)  
22     {  
23         if ((s.length() >= MinLen) && (s.length() <= MaxLen)) {  
24             values.insert(s);  
25             return true;  
26         }  
27         else {  
28             return false;  
29         }  
30     }  
31     ...  
32 private:  
33     std::unordered_set<std::string> values;  
34 };
```

35 In this function, we'll pay to construct `s`, even if it's not copied, e.g., if the value
36 passed in is too short or too long. For example, given this code,

```
1 Widget w;
2
3 std::string finalGWTWWords("Tomorrow is another day");
4 ...
5 auto status = w.insert(finalGWTWWords);
6 we'll pay to copy finalGWTWWords, even if its length is less than MinLen or great-
7 er than MaxLen. (Note that we'll really pay for a copy, not a move, because final-
8 GWTWWords is an lvalue.) This is hardly an efficiency win.
9 Even when you're dealing with a function performing an unconditional copy on a
10 type that's known to be cheap to move, there are times when pass by value may
11 not be a suitable design decision. For code that has to be as fast as possible, avoid-
12 ing even cheap moves can be important. Besides, it's not always clear how many
13 moves are truly being performed. In our Widget::setName example, pass by val-
14 ue incurs only a single extra move operation, but suppose that Widget::setName
15 called Widget::validateName, and this function also passed by value. (Presum-
16 ably it has a reason for always copying its parameter, e.g., to store it in a data
17 structure of all values it validates.) And suppose that validateName called a third
18 function that also passed by value...
19 You can see where this is headed. When there are chains of function calls, each of
20 which employs pass by value because "it costs only one inexpensive move," the
21 cost for the entire chain of calls may not be something you can tolerate. Using by-
22 reference parameter passing (as is the case with lvalue and rvalue overloads and
23 the use of universal references), chains of calls don't incur this kind of accumulated
24 overhead.
25 If you were paying particularly close attention during this Item, you probably no-
26 ticed my comment that there's "generally" no catch to using pass by value, provid-
27 ed the constraints we've discussed are satisfied, i.e., that moving the parameter
28 type is cheap and that the parameter is unconditionally copied. "Generally?," you
29 probably wondered. "Generally?!" "What's up with 'generally'?!"
30 What's up with "generally" is the slicing problem. Pass by value is susceptible to it.
31 Pass by reference isn't. This is well-trod C++98 ground, so I won't dwell on it, but if
```

1 you have a function that is designed to accept a parameter of a base class type *or*
2 *any type derived from it*, you don't want to declare a pass-by-value parameter of
3 that type, because you'll "slice off" the derived-class characteristics of any derived
4 type object that may be passed in:

```
5 class Widget { ... };                                // base class
6 class SpecialWidget: public Widget { ... };      // derived class
7 void processWidget(Widget w);    // func for any kind of Widget,
8                                         // including derived types;
9 ...                                         // suffers from slicing problem
10 SpecialWidget sw;
11 ...
12 processWidget(sw);           // processWidget sees a
13                               // Widget, not a SpecialWidget!
```

14 If you're not familiar with the slicing problem, search engines and the Internet are
15 your friend; there's lots of information available. You'll find that the existence of
16 the slicing problem is another reason (on top of the efficiency hit) why pass by
17 value has a shady reputation in C++98. There are good reasons why one of the first
18 things you probably learned about C++ programming was to avoid passing objects
19 by value.

20 C++11 doesn't fundamentally change the C++98 wisdom regarding pass by value.
21 In general, pass by value still entails a performance hit you'd prefer to avoid, and
22 pass by value can still lead to the slicing problem. What's new in C++11 is the dis-
23 tinction between lvalue and rvalue arguments. Declaring functions that take ad-
24 vantage of move semantics for rvalues requires either writing multiple functions
25 (i.e., overloading for lvalues and rvalues) or using universal references, both of
26 which have drawbacks. For the special case of cheap-to-move types passed to
27 functions that always copy them and where slicing is not a concern, pass by value
28 offers an easy-to-implement alternative that's nearly as efficient as its pass-by-
29 reference competitors, but avoids their disadvantages.

1 **Things to Remember**

- 2 ♦ For cheap-to-move parameters that are unconditionally copied, pass by value
3 is nearly as efficient as pass by reference, it's easier to implement, and it can
4 generate less object code.
- 5 ♦ Pass by value is subject to the slicing problem, so it's typically inappropriate
6 for base class parameter types.

7 **Item 18: Consider emplacement instead of insertion.**

8 If you have a container holding, say, `std::strings`, it seems logical that when you
9 add a new element via an insertion function (i.e., `insert`, `push_front`,
10 `push_back`, or, for `std::forward_list`, `insert_after`), the type of element
11 you'll pass to the function will be `std::string`. After all, that's what the container
12 has in it.

13 Logical though this may be, it's not always true. Consider this code:

```
14   std::vector<std::string> vs;                            // container of std::string
15   vs.push_back("xyzzy");                                    // add string literal
```

16 Here, the container holds `std::strings`, but what you have in hand—what you're
17 actually trying to `push_back`—is a string literal (i.e., a sequence of characters in-
18 side quotes). A string literal is not a `std::string`, and that means that the argu-
19 ment you're passing to `push_back` is not of the type held by the container.

20 `push_back` for `std::vector` is overloaded for lvalues and rvalues as follows:

```
21   template <class T,                                        // from the C++11
22            class Allocator = allocator<T> >            // Standard
23   class vector {
24   public:
25   ...
26   void push_back(const T& x);                            // insert lvalue
27   void push_back(T&& x);                                // insert rvalue
28   ...
29 };
```

30 In the call

```
31   vs.push_back("xyzzy");
```

1 compilers see a mismatch between the type of the argument (`const char[6]`—
2 see Item 1) and the type of the parameter taken by `push_back` (`std::string`).
3 They address the mismatch by generating code to create a temporary
4 `std::string` object from the string literal, and they pass that temporary object to
5 `push_back`. In other words, they treat the call as if it had been written like this:

6 `vs.push_back(std::string("xyzzy")); // create temp. std::string`
7 `// and pass it to push_back`

8 The code compiles and runs, and everybody goes home happy. Everybody except
9 the performance freaks, that is, because the performance freaks recognize that this
10 code isn't as efficient as it should be.

11 To create a new element in a container of `std::strings`, they understand, a
12 `std::string` constructor is going to have to be called, but the code above doesn't
13 call just one constructor. It calls two. Furthermore, it tacks on a call to the
14 `std::string` destructor, which seems not just gratuitous, but downright, well,
15 tacky.

16 Here's what happens at runtime in the call to `push_back`:

17 1. A temporary `std::string` object is created from the string literal, "xyzzy".
18 This object has no name; we'll call it *temp*. Construction of *temp* is the first
19 `std::string` construction. Because it's a temporary object, *temp* is an rvalue.

20 2. *temp* is passed to the rvalue overload for `push_back`, where it's bound to the
21 rvalue reference parameter, *x*. A copy of *x* is then constructed in the memory
22 for the `std::vector`. This construction—the *second* one—is what actually
23 creates a new object inside the `std::vector`. (The constructor that's used to
24 copy *x* into the `std::vector` is the move constructor, because *x*, being an
25 rvalue reference, gets cast to an rvalue before it's copied. For information
26 about the casting of rvalue reference parameters to rvalues, see Item 27.)

27 3. When `push_back` returns, *temp* is destroyed, thus calling the `std::string`
28 destructor.

29 The performance freaks can't help but notice that if there were a way to take the
30 string literal and pass it directly to the code in step 2 that constructs the

1 std::string object inside the std::vector, i.e., that performs the only con-
2 struction that's actually required, we could avoid constructing and destroying
3 *temp*. That would be maximally efficient, and even the performance freaks would
4 go home happy.

Because you're a C++ programmer, there's an above-average chance you're a performance freak. If you're not, you're still probably sympathetic to their point of view. (If you're not at all interested in performance, shouldn't you be in the Python room down the hall?) So I'm pleased to tell you that there is a way to do exactly what is needed for maximal efficiency in the call to `push_back`. It's to not call `push_back`. `push_back` is the wrong function. The function you're looking for is `emplace_back`.

12 `emplace_back` does exactly what we want: it uses whatever argument is passed to
13 it to construct a new `std::string` directly inside the `std::vector`. No tempo-
14 raries are involved:

```
15 vs.emplace_back("xyzzy");    // construct std::string inside
16                                // vs directly from "xyzzy"
17 emplace_back uses perfect forwarding, so, as long as you don't bump into one of
18 perfect forwarding's limitations (see Item 32), you can pass any number of argu-
19 ments of any combination of types through emplace_back. For example, if you'd
20 like to create a std::string in vs via the std::string constructor taking a
21 character and a repeat count, this would do it:
```

24 `emplace_back` is available for every standard container that supports
25 `push_back`. Similarly, every standard container that supports `push_front` sup-
26 ports `emplace_front`. And every standard container that supports `insert`
27 (which is all but `std::forward_list` and `std::array`) supports `emplace`. The
28 associative containers offer `emplace_hint` to complement their `insert` functions
29 that take a “hint” iterator, and `std::forward_list` has `emplace_after` to match
30 its `insert_after`.

1 All emplacement functions work the same way as `emplace_back`: they perfect-
2 forward their arguments to the code that creates a new object in the container. All
3 avoid the cost of type conversions that arise through use of the insertion functions
4 when the type of the passed-in argument doesn't match the type stored in the con-
5 tainer. Hence:

```
6 std::vector<std::string> vs;           // as before
7 ...
8 auto midPoint = vs.begin() + vs.size() / 2;
9 vs.insert(midPoint, "hello");          // create temp std::string,
10                                // move temp into vector,
11                                // destroy temp
12 ...
13 midPoint =                         // recompute midpoint
14     vs.begin() + vs.size() / 2;
15 vs.emplace(midPoint, "world");       // create std::string
16                                // directly in vector
```

16 One of the nicest things about the emplacement functions is that when the type
17 passed is the same as the type stored in the container (i.e., when there's no type
18 mismatch), they're no less efficient than calling their insertion counterparts. This
19 means that you can safely fall into the habit of using the emplacement functions all
20 the time. They're often more efficient than their `insert/push_back/push_front`
21 siblings, and they're never less efficient.

22 At the same time, emplacement functions have a couple of sharp edges, and if
23 you're not careful, you can cut yourself. For example, suppose you have a container
24 of `std::shared_ptr<Widget>`s,

```
25 std::list<std::shared_ptr<Widget>> ptrs;
26 and you want to add a std::shared_ptr to a Widget that should be released via
27 a custom deleter (see Item 21). Item 23 explains that you should use
28 std::make_shared to create std::shared_ptrs whenever you can, but it also
29 concedes that there are situations where you can't. One such situation is when you
30 want to specify a custom deleter. In that case, you must call new directly to get the
31 raw pointer to be managed by the std::shared_ptr.
```

32 If the custom deleter is this function,

```
1 void killWidget(Widget* pWidget);  
2 the code using an insertion function could look like this:  
3 ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));  
4 It could also look like this, though the meaning would be the same:  
5 ptrs.push_back( { new Widget, killWidget } );  
6 Either way, a temporary std::shared_ptr would be constructed before calling  
7 push_back. push_back's parameter is a reference to a std::shared_ptr, so  
8 there has to be a std::shared_ptr for this parameter to refer to.  
9 The creation of the temporary std::shared_ptr is what emplace_back would  
10 avoid, but in this case, that temporary is worth far more than it costs. Consider the  
11 following potential sequence of events:  
12 1. In either call above, a temporary std::shared_ptr<Widget> is constructed  
13 to hold the raw pointer resulting from "new Widget". Call this object p.  
14 2. push_back takes p by reference. During allocation of a list node to hold a copy  
15 of p, an out-of-memory exception gets thrown.  
16 3. As the exception propagates out of push_back, p is destroyed. Being the sole  
17 std::shared_ptr referring to the Widget it's managing, it automatically re-  
18 leases that Widget, in this case by calling killWidget.  
19 Even though an exception occurred, nothing leaks: the Widget created via "new  
20 Widget" in the call to push_back is released in the destructor of the  
21 std::shared_ptr that was created to manage it (p). Life is good.  
22 But now consider what happens if emplace_back is called instead of push_back:  
23 ptrs.emplace_back(new Widget, killWidget);  
24 1. The raw pointer resulting from "new Widget" is perfect-forwarded to the point  
25 inside push_back where a list node is to be allocated. That allocation fails, and  
26 an out-of-memory exception is thrown.
```

1 2. As the exception propagates out of `push_back`, the raw pointer that was the
2 only way to get at the `Widget` on the heap is lost. That `Widget` (and any re-
3 sources it owns) is leaked.

4 In this scenario, life is *not* good, and the fault doesn't lie with `std::shared_ptr`.
5 The same kind of problem can arise through the use of `std::unique_ptr` with a
6 custom deleter. Fundamentally, the effectiveness of resource-managing classes like
7 `std::shared_ptr` and `std::unique_ptr` is predicated on resources (such as
8 raw pointers returned from `new`) being *immediately* passed to the constructors of
9 the resource-managing objects. The fact that functions like `std::make_shared`
10 and `std::make_unique` automate this is one of the reasons they're so important.

11 In calls to the insertion functions of containers holding resource-managing objects
12 (e.g., `std::list<std::shared_ptr<Widget>>`), the functions' parameter types
13 generally ensure that nothing gets between acquisition of a resource (e.g., use of
14 naked `new`) and construction of the object managing the resource. In the emplace-
15 ment functions, perfect-forwarding defers the creation of the resource-managing
16 objects until they can be constructed in the container's memory, and that opens a
17 window during which exceptions can lead to resource leaks. When working with
18 containers of resource-managing objects, you must take care to ensure that if you
19 choose an emplacement function over its insertion counterpart, you're not paying
20 for improved code efficiency with diminished exception safety.

21 The fundamental difference between insertion and emplacement functions is that
22 insertion functions take *objects to be inserted*, while emplacement functions take
23 *constructor arguments* for objects to be inserted. One consequence of this—the one
24 driving this Item—is that emplacement functions can be more efficient, but a sec-
25 ond consequence is that you may get less benefit from `explicit` constructors
26 than you expect. For example, suppose, in celebration of C++11's support for regu-
27 lar expressions, you create a container of regular expression objects:

28 `std::vector<std::regex> regexes;`

29 Distracted by your colleagues' quarreling over the ideal number of times per day
30 to check one's Facebook account, you accidentally write the following seemingly
31 meaningless code:

```
1  regexes.emplace_back(nullptr);      // add nullptr to container
2                                         // of regexes?

3  You don't notice the error as you type it, and your compilers accept the code with-
4  out complaint, so you end up wasting a bunch of time debugging. At some point,
5  you discover that you appear to have inserted a null pointer into your container of
6  regular expressions. But how is that possible? Pointers aren't regular expressions,
7  and if you tried to do something like this,
```



```
8  std::regex r = nullptr;           // error! won't compile
```



```
9  compilers would reject your code. Interestingly, they would also reject it if you
10 called push_back instead of emplace_back:
```



```
11 regexes.push_back(nullptr);       // error! won't compile
```



```
12 The curious behavior you're experiencing stems from the fact that std::regex
13 objects can be constructed from character strings, i.e., from const char* pointers.
14 That's what makes useful code like this legal:
```



```
15 std::regex upperCaseWord("[A-Z]+");
```



```
16 However, because nullptr implicitly converts to all pointer types, the em-
17 place_back call that unexpectedly compiles is treated by compilers more or less
18 as if you'd written this:
```



```
19 regexes.emplace_back(static_cast<const char*>(nullptr));
```



```
20 Creation of a std::regex from a character string can exact a comparatively large
21 runtime cost, so, to minimize the likelihood that such an expense will be incurred
22 unintentionally, the std::regex constructor taking a const char* pointer is ex-
23 plicit. That's why these lines don't compile:
```



```
24 std::regex r = nullptr;           // error! won't compile
```



```
25 regexes.push_back(nullptr);      // error! won't compile
```



```
26 In both cases, we're requesting an implicit conversion from a pointer to a
27 std::regex, and the explicitness of that constructor prevents such conver-
28 sions.
```

1 In the call to `emplace_back`, however, we're not claiming to pass a `std::regex`
2 object. Instead, we're passing a *constructor argument* for a `std::regex` object.
3 That's not considered an implicit conversion request. Rather, it's viewed as if you'd
4 written this code:

5 `std::regex r(nullptr); // compiles`

6 If the laconic comment "compiles" suggests a lack of enthusiasm, that's good, be-
7 cause this code, though it will compile, has undefined behavior. The `std::regex`
8 constructor taking a `const char*` pointer requires that the pointed-to string com-
9 prise a valid regular expression, and the null pointer fails that requirement. If you
10 write and compile such code, the best you can hope for is that it crashes at
11 runtime. If you're not so lucky, you and your debugger could be in for a special
12 bonding experience.

13 Setting aside `push_back`, `emplace_back`, and bonding for a moment, notice how
14 these very similar initialization syntaxes yield different results:

15 `std::regex r1 = nullptr; // error! won't compile`

16 `std::regex r2(nullptr); // compiles`

17 In the official terminology of the Standard, there are two kinds of initialization:
18 *copy initialization* and *direct initialization*. The syntax used to initialize `r1` (em-
19 ploying the equals-sign) is defined to be copy initialization. The syntax used to ini-
20 tialize `r2` (with the parentheses, although braces may also be used) is defined to be
21 direct initialization. Copy initialization is not permitted to use `explicit` construc-
22 tors. Direct initialization is. That's why the line initializing `r1` doesn't compile, but
23 the line initializing `r2` does.

24 But back to `push_back` and `emplace_back` (and, more generally, the insertion
25 functions versus the emplacement functions). Insertion functions employ copy ini-
26 tialization, which means they can't make use of `explicit` constructors. Emplace-
27 ment functions use direct initialization, so they can. Hence:

28 `regexes.emplace_back(nullptr); // fine, direct init permits`
29 `// use of explicit`
30 `// pointer→std::regex ctor`

```
1 regexes.push_back(nullptr);      // error! copy init forbids
2                               // use of explicit
3                               // pointer→std::regex ctor
4 The lesson to take away is that when you use an emplacement function, be espe-
5 cially careful to make sure you're passing the correct arguments, because even ex-
6 plicit constructors will be considered by compilers as they try to find a way to
7 interpret your code as valid.
```

8 Things to Remember

- 9 ♦ Emplacement functions are often more efficient than their insertion counter-
- 10 parts, and they're never less efficient.
- 11 ♦ For containers of resource-managing objects, emplacement functions may suf-
- 12 fer resource leaks that would not arise through use of insertion functions.
- 13 ♦ Emplacement functions may perform type conversions that would be rejected
- 14 by insertion functions.

15 Item 19: Understand special member function generation.

16 In official C++ parlance, the “special member functions” are the ones that C++ gen-
17 erates on its own. C++98 has four such functions: the default constructor, the de-
18 structor, the copy constructor, and the copy assignment operator. There’s fine
19 print, of course. These functions are generated only if they’re needed, i.e., if some
20 code uses them without their being expressly declared in the class. A default con-
21 structor is generated only if the class declares no constructors at all. (This pre-
22 vents compilers from creating a default constructor for a class where you’ve speci-
23 fied that constructor arguments are required.) Generated special member func-
24 tions are implicitly `inline`, and they’re nonvirtual unless the function in question
25 is a destructor in a derived class inheriting from a base class with a virtual destruc-
26 tor. In that case, the compiler-generated destructor is also virtual.

27 But you already know these things. Yes, yes, ancient history: Mesopotamia, the
28 Shang dynasty, FORTRAN, C++98. But times have changed, and the rules for special
29 member function generation in C++ have changed with them. It’s important to be
30 aware of the new rules, because few things are as central to effective C++ pro-

1 gramming as knowing when compilers silently insert member functions into your
2 classes.

3 As of C++11, the special member functions club has two more inductees: the move
4 constructor and the move assignment operator. For a class `Widget`, their signa-
5 tures are:

```
6 class Widget {  
7 public:  
8     ...  
9     Widget(Widget&& rhs);           // move constructor  
10    Widget& operator=(Widget&& rhs); // move assignment operator  
11    ...  
12};
```

13 The rules governing their generation and behavior are analogous to those for their
14 copying siblings. The move operations are generated only if they're needed, and if
15 they are generated, they perform "memberwise moves" on the data members of
16 the class. That means that the move constructor move-constructs each data mem-
17 ber of the class from the corresponding member of its parameter, `rhs`, and the
18 move assignment operator move-assigns each data member of the class from the
19 corresponding member of its parameter, `rhs`. Furthermore, the move constructor
20 move-constructs its base class parts (if there are any), and the move assignment
21 operator move-assigns its base class parts, too.

22 Now, when I refer to a move operation move-constructing or move-assigning a
23 data member or base class, there is no guarantee that a move will actually take
24 place. "Memberwise moves" are, in reality, more like memberwise move *requests*,
25 because types that aren't *move-enabled* (i.e., that offer no special support for move
26 operations, e.g., most C++98 legacy classes) will be "moved" via their copy opera-
27 tions. The heart of each memberwise "move" is application of `std::move` to the
28 source to be moved from, and the result of this application is used during function
29 overload resolution to determine whether a move or a copy should be performed.
30 Item 25 covers this process in detail. For this Item, simply remember that a mem-
31 berwise move consists of move operations on data members and base classes that
32 support move operations, but for data members and base classes that offer only
33 copy operations, a memberwise "move" entails making a copy.

1 As is the case with the copy operations, the move operations aren't generated if
2 you declare them yourself. However, the precise conditions under which they are
3 generated differ a bit from those for the copy operations.

4 The two copy operations are independent: declaring one doesn't prevent compil-
5 ers from generating the other. So if you declare a copy constructor, but no copy
6 assignment operator, then write code that requires copy assignment, compilers
7 will generate the copy assignment operator for you. Similarly, if you declare a copy
8 assignment operator, but no copy constructor, yet your code requires copy con-
9 struction, compilers will generate the copy constructor for you. That was true in
10 C++98, and it's still true in C++11.

11 The two move operations are not independent. If you declare either, that prevents
12 compilers from generating the other. The rationale is that if you declare, say, a
13 move constructor for your class, you're indicating that there's something about
14 how move construction should be implemented that's different from the default
15 memberwise move that compilers would generate. And if there's something wrong
16 with memberwise move construction, compilers reason, there'd probably be
17 something wrong with memberwise move assignment, too. So declaring a move
18 constructor prevents a move assignment operator from being generated, and de-
19 claring a move assignment operator prevents compilers from generating a move
20 constructor.

21 Furthermore, move operations won't be generated for any class that explicitly de-
22 clares a copy operation. The justification is that declaring a copy operation (con-
23 struction or assignment) indicates that the normal approach to copying an object
24 (memberwise copy) isn't appropriate for the class, and compilers figure that if
25 memberwise copy isn't appropriate for the copy operations, memberwise move
26 probably isn't appropriate for the move operations.

27 This goes in the other direction, too. Declaring a move operation (construction or
28 assignment) in a class prevents compilers from generating copy operations. After
29 all, if memberwise move isn't the proper way to move an object, there's no reason
30 to expect that memberwise copy is the proper way to copy it. This may sound like
31 it could break C++98 code, because the conditions under which the copy opera-

1 tions are generated are more constrained in C++11 than in C++98, but further re-
2 flection reveals that this is not the case. C++98 code can't have move operations,
3 because there was no such thing as "moving" objects in C++98. The only way a leg-
4 acy class can have user-declared move operations is if they were added for C++11,
5 and classes that are modified to take advantage of move semantics have to play by
6 the C++11 rules for special member function generation.

7 Perhaps you've heard of a guideline known as the *Rule of Three*. The Rule of Three
8 emerged fairly early in the C++ era (the early 1990s), and it states that if you de-
9 clare any of a copy constructor, copy assignment operator, or destructor, you
10 should declare all three. It grew out of the observation that the need to take over
11 the meaning of a copy operation almost always stemmed from the class perform-
12 ing some kind of resource management, and that almost always implied that (1)
13 whatever resource management was being done in one copy operation probably
14 needed to be done in the other copy operation and (2) the class destructor would
15 also be participating in management of the resource (e.g., releasing it). The classic
16 resource to be managed was memory, and this is why all the Standard Library
17 classes that manage memory (e.g., the STL containers that perform dynamic
18 memory management) all declare "the big three:" both copy operations and a de-
19 structor.

20 A consequence of the Rule of Three is that the presence of a user-declared destruc-
21 tor indicates that simple memberwise copy is unlikely to be appropriate for the
22 copying operations in the class. That, in turn, suggests that if a class declares a de-
23 structor, the copy operations probably shouldn't be automatically generated, be-
24 cause they wouldn't do the right thing. At the time C++98 was adopted, the signifi-
25 cance of this line of reasoning was not fully appreciated, so in C++98, the existence
26 of a user-declared destructor had no impact on compilers' willingness to generate
27 copy operations. That continues to be the case in C++11, but only because restrict-
28 ing the conditions under which the copy operations are generated would break too
29 much legacy code.

30 The reasoning behind the Rule of Three remains valid, however, and combined
31 with the observation that declaration of a copy operation precludes the implicit

1 generation of the move operations, C++11 does *not* generate move operations for a
2 class with a user-declared destructor.

3 So move operations are generated for classes (when needed) only if all of these
4 three things are true:

5 • No copy operations are declared in the class.

6 • No move operations are declared in the class.

7 • No destructor is declared in the class.

8 At some point, analogous rules may be extended to the copy operations, because
9 C++11 *deprecates* the automatic generation of copy operations for classes declar-
10 ing copy operations or a destructor. This puts such function generation on death
11 row. Feature deprecation is the Standard's way of issuing a warning that behavior
12 valid in the current Standard is officially frowned upon, and the feature may be
13 removed in a future Standard.

14 In practice, this means that if you have code that depends on the generation of
15 copy operations in classes declaring a destructor or one of the copy operations,
16 you should think about upgrading these classes to eliminate the dependence. Pro-
17 vided the behavior of the compiler-generated functions is correct (i.e., if member-
18 wise copying of the class's nonstatic data members is what you want), your job is
19 easy, because C++11's “= **default**” lets you say that explicitly:

```
20 class Widget {  
21 public:  
22     ...  
23     ~Widget(); // user-declared dtor  
24     ...  
25     Widget(const Widget&) = default; // behavior is OK  
26     Widget& operator=(const Widget&) = default; // behavior is OK  
27     ...  
28 };
```

31 In fact, you may want to adopt a policy of manually declaring all the copy and move
32 functions you want your class to support, then using “= **default**” to force compil-

1 ers to generate the default implementations for the functions where these imple-
2 mentations would be appropriate. It's a bit more work than having compilers si-
3 lently generate those functions on their own, but it makes your intentions clearer,
4 and it can help you side-step some fairly subtle bugs. For example, suppose you
5 have a class representing a string table, i.e., a data structure that permits fast
6 lookups of string values via an integer ID:

```
7 class StringTable {
8 public:
9     StringTable() {}
10    ...           // functions for insertion, erasure, lookup,
11                  // etc., but no copy/move/dtor functionality
12 private:
13     std::map<int, std::string> values;
14 };
```

15 Assuming that the class declares no copy operations, no move operations, and no
16 destructor, compilers will automatically generate these functions if they are used.
17 That's very convenient.

18 But suppose that sometime later, it's decided that logging the default construction
19 and the destruction of such objects would be useful. Adding that functionality is
20 easy:

```
21 class StringTable {
22 public:
23     StringTable()
24     { makeLogEntry("Creating StringTable object"); } // added
25     ~StringTable() // also
26     { makeLogEntry("Destroying StringTable object"); } // added
27     ... // other funcs as before
28 private:
29     std::map<int, std::string> values; // as before
30 };
```

31 This seems innocuous, but it has a potentially significant side effect: it prevents the
32 move operations from being generated. To be specific, the existence of the user-
33 declared destructor precludes their generation. That's why I've highlighted only
34 the destructor declaration above. The modification to the default constructor and
35 the body of the destructor is irrelevant for this discussion.

1 Although the addition of the destructor prevents the move operations from coming
2 into existence, generation of the class's copy operations is unaffected. Your code is
3 therefore likely to compile, run, and pass all your functional testing. That includes
4 testing its move functionality, because even though this class is no longer move-
5 enabled, requests to move it will compile and run. Such requests will, as noted ear-
6 lier in this Item, cause copies to be made. Which means that code "moving"
7 `StringTable` objects actually copies them, i.e., copies the underlying
8 `std::vector<std::string>` objects. And copying a
9 `std::vector<std::string>` is likely to be *orders of magnitude* slower than
10 moving it. The simple act of adding a destructor to the class could thereby have
11 introduced a significant performance problem!

12 Fixing the problem is easy: just tell your compilers to start generating the move
13 operations again:

```
14 class StringTable {  
15 public:  
16     StringTable() { ... }                                // as before  
17     ~StringTable() { ... }                             // as before  
18     StringTable(StringTable&&) = default;           // move ctor  
19     StringTable& operator=(StringTable&&) = default; // move op=  
20     ...                                              // other funcs as before  
21 private:  
22     std::map<int, std::string> values;    // as before  
23 };
```

24 But as long as you're doing that, you might as well do the same for the copy opera-
25 tions. That clearly expresses that you desire the compiler-generated copy behavior
26 for this class, and it also future-proofs the class against the day when copy opera-
27 tions are no longer created for classes with user-declared destructors. The result
28 looks like this:

```
29 class StringTable {  
30 public:  
31     StringTable() { ... }                                // as before  
32     ~StringTable() { ... }                             // as before  
33     StringTable(const StringTable&) = default;        // copy ctor  
34     StringTable(StringTable&&) = default;            // move ctor
```

```
1  StringTable&
2      operator=(const StringTable&) = default;           // copy op=
3  StringTable& operator=(StringTable&&) = default;    // move op=
4  ...
5  private:
6      std::map<int, std::string> values;                 // as before
7  };
```

8 Now, having endured my endless blathering about the rules governing the copy
9 and move operations in C++11, you may wonder when I'll turn my attention to the
10 two other special member functions, the default constructor and the destructor.
11 That time is now, but only for this sentence, because nothing has changed for these
12 member functions: the rules in C++11 are the same as in C++98.

13 The C++11 rules governing the special member functions are thus:

- 14 • **Default constructor:** Same rules as C++98. Generated only if the class con-
15 tains no user-declared constructors.
- 16 • **Destructor:** Same rules as C++98. Virtual only if a base class destructor is vir-
17 tual.
- 18 • **Copy constructor:** Same runtime behavior as C++98. Performs memberwise
19 copying of (nonstatic) class data members. Generated only if the class contains
20 neither a user-declared copy constructor nor any move operations. Generation
21 of this function in a class with a user-declared copy assignment operator or de-
22 structor is deprecated.
- 23 • **Copy assignment operator:** Same runtime behavior as C++98. Performs
24 memberwise copying of (nonstatic) class data members. Generated only if the
25 class contains neither a user-declared copy assignment operator nor any move
26 operations. Generation of this function in a class with a user-declared copy
27 constructor or destructor is deprecated.
- 28 • **Move constructor and move assignment operator:** Performs memberwise
29 moving of (nonstatic) class data members. Generated only if the class contains
30 neither user-declared copy operations, move operations, nor destructor.

1 Note, by the way, that there's nothing in the rules about the existence of a member
2 function *template* preventing compilers from generating the special member func-
3 tions. That means that if **Widget** looks like this,

```
4 class Widget {  
5     ...  
6     template<typename T>           // copy-construct Widget  
7         Widget(const T& rhs);       // from anything  
8     template<typename T>           // copy-assign Widget  
9         Widget& operator=(const T& rhs); // from anything  
10    ...  
11};
```

12 compilers will still generate copy and move operations for **Widget** (assuming the
13 usual conditions governing their generation are fulfilled), even though these tem-
14 plates could be instantiated to produce the signatures for the copy constructor and
15 copy assignment operator. (That would be the case when *T* is **Widget**.) In all like-
16 lihood, this will strike you as an edge case barely worth acknowledging, but there's
17 a reason I'm mentioning it. As Item 28 explains, it can have important consequenc-
18 es.

19 **Things to Remember**

- 20 ♦ The special member functions are those compilers may generate on their own:
21 the default constructor, destructor, copy operations, and move operations.
- 22 ♦ Move operations are generated only for classes lacking explicitly-declared
23 move operations, copy operations, and a destructor.
- 24 ♦ The copy constructor is generated only for classes lacking an explicitly-
25 declared copy constructor and the move operations. The copy assignment op-
26 erator is generated only for classes lacking an explicitly-declared copy assign-
27 ment operator and the move operations. Generation of the copy operations in
28 classes with an explicitly-declared destructor is deprecated,
- 29 ♦ Member function templates never suppress generation of special member
30 functions.

1 **Chapter 4 Smart Pointers**

2 Poets and songwriters have a thing about love. And sometimes about counting.
3 Occasionally both. Inspired by the rather different takes on love and counting by
4 Elizabeth Barrett Browning (“How do I love thee? Let me count the ways.”) and
5 Paul Simon (“There must be 50 ways to leave your lover”), we might try to enu-
6 merate the reasons why a raw pointer is hard to love:

7 1. Its declaration doesn’t indicate whether it points to a single object or to an ar-
8 ray.

9 2. Its declaration reveals nothing about whether you should destroy what it
10 points to when you’re done using it, i.e., if the pointer *owns* the thing it points
11 to.

12 3. If you somehow determine that you should destroy what the pointer points to,
13 there’s no way to tell how. Should you use `delete`, or is there a different de-
14 struction mechanism (e.g., a dedicated destruction function the pointer should
15 be passed to)?

16 4. Even if you manage to find out that `delete` is the way to go, Reason 1 means
17 it’s rarely possible to know whether to use the object form (“`delete`”) or the
18 array form (“`delete []`”). If you use the wrong form, results are undefined.

19 5. Assuming you ascertain that the pointer owns what it points to and you dis-
20 cover how to destroy it, it’s difficult to ensure that you perform the destruction
21 *exactly once* along every control path in your code (including those due to ex-
22 ceptions). Missing a path leads to resource leaks, and doing the destruction
23 more than once leads to undefined behavior.

24 6. There’s typically no way to tell if the pointer dangles, i.e., points to memory
25 that no longer holds the object the pointer is supposed to point to. Dangling
26 pointers arise when objects are destroyed while pointers still point to them.

27 Raw pointers are powerful tools, to be sure, but decades of experience have
28 demonstrated that with only the slightest lapse in concentration or discipline,
29 these tools can turn on their ostensible masters. Surely we can do better.

1 *Smart pointers* are one way to address these issues. Smart pointers are wrappers
2 around raw pointers that act much like the raw pointers they wrap, but that avoid
3 many of their pitfalls. You should therefore prefer smart pointers to raw pointers.
4 Smart pointers can do virtually everything raw pointers can, but with far fewer
5 opportunities for error.

6 There are four smart pointers in C++11: `std::auto_ptr`, `std::unique_ptr`,
7 `std::shared_ptr`, and `std::weak_ptr`. All are designed to help manage the life-
8 time of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that
9 such objects are destroyed in the appropriate manner at the appropriate time (in-
10 including in the event of exceptions).

11 `std::auto_ptr` is a deprecated leftover from C++98. It was an attempt to stand-
12 ardize what later became C++11's `std::unique_ptr`. Doing the job right required
13 move semantics, but C++98 didn't have them. As a workaround, `std::auto_ptr`
14 co-opted its copy operations for moves. This led to surprising code (copying a
15 `std::auto_ptr` sets it to null!) and frustrating usage restrictions (e.g., it's not
16 possible to store `std::auto_ptr`s in containers).

17 `std::unique_ptr` does everything `std::auto_ptr` does, plus more. It does it as
18 efficiently, and it does it without warping what it means to copy an object. It's bet-
19 ter than `std::auto_ptr` in every way. The only legitimate use case for
20 `std::auto_ptr` is a need to compile code with C++98 compilers. Unless you have
21 that constraint, you should replace `std::auto_ptr` with `std::unique_ptr` and
22 never look back.

23 The smart pointer APIs are remarkably varied. About the only functionality com-
24 mon to all is default construction. Because comprehensive references for these
25 APIs are widely available in both electronic and print form, I'll focus my discus-
26 sions on information that's often missing from API overviews, e.g., noteworthy use
27 cases, runtime cost analyses, etc. Mastering such information can be the difference
28 between merely using these smart pointers and using them *effectively*.

1 **Item 20: Use `std::unique_ptr` for exclusive-ownership re-**
2 **source management.**

3 When you reach for a smart pointer, `std::unique_ptr` should generally be the
4 one closest at hand. By default, `std::unique_ptr`s are the same size as raw
5 pointers, and for most operations (including dereferencing), they execute exactly
6 the same instructions. This means you can use them even in situations where
7 memory and cycles are tight. If a raw pointer is small enough and fast enough for
8 you, a `std::unique_ptr` almost certainly is, too.

9 Like `std::auto_ptr` before it, `std::unique_ptr` embodies *exclusive ownership*
10 semantics. A non-null `std::unique_ptr` always owns what it points to. Moving a
11 `std::unique_ptr` transfers ownership from the source pointer to the destination
12 pointer. (The source pointer is set to null.) Copying a `std::unique_ptr` isn't al-
13 lowed, because if you could copy a `std::unique_ptr`, you'd end up with two
14 `std::unique_ptr`s to the same resource, each thinking it owned (and should
15 therefore destroy) that resource. `std::unique_ptr` is thus a *move-only type*. Up-
16 on destruction, a non-null `std::unique_ptr` destroys its resource. By default,
17 resource destruction is accomplished by applying `delete` to the raw pointer in-
18 side the `std::unique_ptr`.

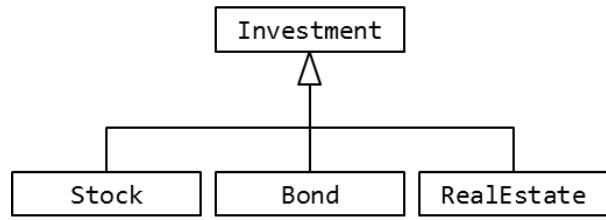
19 A common use for `std::unique_ptr` is as a factory function return type for ob-
20 jects in a hierarchy. Suppose we have a hierarchy for types of investments (e.g.,
21 stocks, bonds, real estate, etc.) with a base class `Investment`.

```
class Investment { ... };

class Stock:
    public Investment { ... };

class Bond:
    public Investment { ... };

class RealEstate:
    public Investment { ... };
```



22 A factory function for such a hierarchy typically allocates an object on the heap and
23 returns a pointer to it, with the caller being responsible for deleting the object

1 when it's no longer needed. That's a perfect match for `std::unique_ptr`, because
2 the caller acquires responsibility for the resource returned by the factory (i.e., ex-
3 clusive ownership of it), and the `std::unique_ptr` automatically deletes what it
4 points to when it's destroyed. A factory function for the `Investment` hierarchy
5 could be declared like this:

```
6 template<typename... Ts>           // return std::unique_ptr
7 std::unique_ptr<Investment>        // to an object created
8     makeInvestment(Ts&&... args);  // from the given args
```

9 Callers could use the returned `std::unique_ptr` in a single scope, as follows,

```
10 {
11 ...
12     auto pInvestment =           // pInvestment is of type
13         makeInvestment( arguments ); // std::unique_ptr<Investment>
14 ...
15 }                                // destroy *pInvestment
```

16 but they could also use it in ownership-migration scenarios, such as when the
17 `std::unique_ptr` returned from the factory is moved into a container, the con-
18 tainer element is subsequently moved into a data member of an object, and that
19 object is later destroyed. When that happens, the object's `std::unique_ptr` data
20 member would also be destroyed, and its destruction would cause the resource
21 returned from the factory to be destroyed. If the ownership chain got interrupted
22 due to an exception or other atypical control flow (e.g., premature function return
23 or `break` from a loop), the `std::unique_ptr` owning the managed resource
24 would eventually have its destructor called,[†] and the resource it was managing
25 would thereby be destroyed.

26 By default, that destruction would take place via `delete`, but, during construction,
27 `std::unique_ptr` objects can be configured to use *custom deleters*: arbitrary

[†] There are a few exceptions to this rule. All stem from abnormal program termination. If an exception propagates out of a thread's primary function (e.g., `main`, for the program's initial thread) or if a `noexcept` specification is violated (see Item 16), local objects may not be destroyed, and if `std::abort` is called, they definitely won't be.

1 functions (or function objects, including those arising from lambda expressions) to
2 be invoked when it's time for their resources to be destroyed. If the object created
3 by `makeInvestment` shouldn't be directly deleted, but instead should first have a
4 log entry written, `makeInvestment` could be implemented as follows. (An expla-
5 nation follows the code, so don't worry if you see something whose motivation is
6 less than obvious.)

```
7 auto delInvmt = [](Investment* pInvestment)           // custom
8     {
9         makeLogEntry(pInvestment);                      // (a lambda
10        delete pInvestment;                            // expression)
11    };
12
13 template<typename... Ts>                                // revised
14 std::unique_ptr<Investment, decltype(delInvmt)>      // return type
15 makeInvestment(Ts&... args)
16 {
17     std::unique_ptr<Investment, decltype(delInvmt)> // ptr to be
18     pInv(nullptr, delInvmt);                          // returned
19
20     if ( /* a Stock object should be created */ )
21     {
22         pInv.reset(new Stock(std::forward<Ts>(args)...));
23     }
24     else if ( /* a Bond object should be created */ )
25     {
26         pInv.reset(new Bond(std::forward<Ts>(args)...));
27     }
28     else if ( /* a RealEstate object should be created */ )
29     {
30         pInv.reset(new RealEstate(std::forward<Ts>(args)...));
31     }
32
33     return pInv;
34 }
```

32 In a moment, I'll explain how this works, but first consider how things look if
33 you're a caller. Assuming you store the result of the `makeInvestment` call in an
34 `auto` variable, you frolic in blissful ignorance of the fact that the resource you're
35 using requires special treatment during deletion. In fact, you veritably bathe in
36 bliss, because the use of `std::unique_ptr` means you need not concern yourself
37 with when the resource should be destroyed, much less ensure that the destruc-
38 tion happens exactly once along every control path in the program.

1 `std::unique_ptr` takes care of all those things automatically. From a client's
2 perspective, `makeInvestment`'s interface is sweet.

3 The implementation is pretty nice, too, once you understand the following:

4 • `delInvmt` is the custom deleter for the object returned from `makeInvest-`
5 `ment`. All custom deletion functions accept a raw pointer to the object to be de-
6 stroyed, then do what is necessary to destroy that object. In this case, the ac-
7 tion is to call `makeLogEntry` and then apply `delete`. Using a lambda expres-
8 sion to create `delInvmt` is convenient, but, as we'll see shortly, it's also more
9 efficient than writing a conventional function.

10 • When a custom deleter is to be used, its type must be specified as the second
11 type argument to `std::unique_ptr`. In this case, that's the type of `delInvmt`,
12 and that's why the return type of `makeInvestment` is
13 `std::unique_ptr<Investment, decltype(delInvmt)>`. (For information
14 about `decltype`, see Item 3.)

15 • The basic strategy of `makeInvestment` is to create a null `std::unique_ptr`,
16 make it point to an object of the appropriate type, and then return it. To asso-
17 ciate the custom deleter (`delInvmt`) with `pInv`, we pass that as its second
18 constructor argument.

19 • As Item 23 explains, it's preferable to use `std::make_unique` rather than `new`
20 to create an object for a `std::unique_ptr` to point to, yet this code uses `new`
21 in several places. That's because `std::make_unique` offers no support for
22 custom deleters. (Of course, it would be possible to encapsulate the use of `new`
23 inside a `std::make_unique`-like template that supported the specification of
24 a custom deleter. If you've been waiting for an opportunity to write some code
25 whose development is "left as an exercise for the reader," here's your chance.)

26 • Attempting to assign a raw pointer (e.g., from `new`) to a `std::unique_ptr`
27 won't compile, because it would constitute an implicit conversion from a raw
28 to a smart pointer. Such implicit conversions can be problematic, so C++11's
29 smart pointers prohibit them. That's why `reset` is used to have `pInv` assume
30 ownership of the object created via `new`.

- 1 • With each use of `new`, we perfect-forward the arguments passed to
 2 `makeInvestment` (see Item 27). This makes all the information provided by
 3 callers available to the constructors of the objects being created.
- 4 • The custom deleter takes a parameter of type `Investment*`. Regardless of the
 5 actual type of object created inside `makeInvestment` (i.e., `Stock`, `Bond`, or `Re-`
 6 `alEstate`), it will ultimately be deleted inside the lambda expression as an
 7 `Investment*` object. This means we'll be deleting a derived class object via a
 8 base class pointer. For that to work, the base class—`Investment`—must have
 9 a virtual destructor:

```
10   class Investment {
11     public:
12     ...
13     virtual ~Investment(); // essential
14     ...
15   }; // design
16   // component!
```

16 In C++14, the existence of function return type deduction (see Item 3) means that
 17 `makeInvestment` could be implemented in this simpler and more encapsulated
 18 fashion:

```
19 template<typename... Ts>
20 auto makeInvestment(Ts&&... args) // C++14 only
21 {
22   auto delInvmt = [](Investment* pInvestment) // this is now
23   {
24     makeLogEntry(pInvestment); // make-
25     delete pInvestment; // Investment
26   };
27
28   std::unique_ptr<Investment, decltype(delInvmt)> // as
29   pInv(nullptr, delInvmt); // before
30
31   if ( ... ) // as before
32   {
33     pInv.reset(new Stock(std::forward<Ts>(args)...));
34   }
35   else if ( ... ) // as before
36   {
37     pInv.reset(new Bond(std::forward<Ts>(args)...));
38   }
39   else if ( ... ) // as before
40   {
41     pInv.reset(new RealEstate(std::forward<Ts>(args)...));
42   }
43 }
```

```
1      }
2      return pInv;                                // as before
3  }

4 At the beginning of this Item, I emphasized that, when using the default deleter
5 (i.e., delete), std::unique_ptr objects are the same size as raw pointers. When
6 custom deleters enter the picture, this may no longer be the case. Deleters that are
7 function pointers generally cause the size of the std::unique_ptr to grow from
8 one word to two. For deleters that are function objects, the change in size depends
9 on how much state is stored in the function object. Stateless function objects (e.g.,
10 from lambda expressions with no captures) incur no size penalty, and this means
11 that when a custom deleter can be implemented as either a function or a capture-
12 less lambda expression, the lambda is preferable:
```

```
13 auto delInvmt1 = [](Investment* pInvestment)           // custom
14 {                                                       // deleter
15     makeLogEntry(pInvestment);                         // as
16     delete pInvestment;                               // stateless
17 };                                                 // Lambda

18 template<typename... Ts>                            // return type
19 std::unique_ptr<Investment, decltype(delInvmt1)> // has size of
20 makeInvestment(Ts&... args);                      // Investment*
21

22 void delInvmt2(Investment* pInvestment)           // custom
23 {                                                       // deleter
24     makeLogEntry(pInvestment);                     // as function
25     delete pInvestment;
26 }

27 template<typename... Ts>                            // return type has
28 std::unique_ptr<Investment, // size of Investment*
29         void (*)(Investment*)> // plus at least size
30 makeInvestment(Ts&... args);                      // of function pointer!
```

31 Function object deleters with extensive state can yield `std::unique_ptr` objects
32 of significant size. However, large function objects are often ill-advised on other
33 grounds, and there are ways to make them smaller; see **Error! Reference source**
34 **not found..** If you find yourself with a custom deleter with significant state, you
35 probably need to change your design.

36 Factory functions are not the only common use case for `std::unique_ptr`s.
37 They're even more popular as a mechanism for implementing the Pimpl Idiom. The

1 code for that isn't complicated, but in some cases it's less than straightforward, so
2 I'll refer you to Item 24, which is dedicated to the topic.

3 `std::unique_ptr` comes in two forms, one for individual objects
4 (`std::unique_ptr<T>`) and one for arrays (`std::unique_ptr<T[]>`). As a re-
5 sult, there's never any ambiguity about what kind of entity a `std::unique_ptr`
6 points to. The `std::unique_ptr` API is designed to match the form you're using.
7 For example, there's no indexing operator (`operator[]`) for the single-object
8 form, while the array form lacks dereferencing operators (`operator*` and opera-
9 `tor->`).

10 The existence of `std::unique_ptr` for arrays should be of only intellectual inter-
11 est to you, because `std::array`, `std::vector`, and `std::string` are virtually
12 always better data structure choices than raw arrays. The only situation I can con-
13 ceive of when a `std::unique_ptr<T[]>` would make sense would be when
14 you're using a C-like API that returns a raw pointer to a heap array that you as-
15 sume ownership of.

16 `std::unique_ptr` is the C++11 way to express exclusive ownership, but one of its
17 most attractive features is that it easily and efficiently converts to a
18 `std::shared_ptr`:

```
19 std::shared_ptr<Investment> sp =      // converts std::unique_ptr
20   makeInvestment( arguments );          // to std::shared_ptr
```

21 This is a key part of why `std::unique_ptr` is so well suited as a factory function
22 return type. Factory functions can't know whether callers will want to use exclu-
23 sive-ownership semantics for the object they return or whether shared ownership
24 (i.e., `std::shared_ptr`) would be more appropriate. By returning a
25 `std::unique_ptr`, factories provide callers with the most efficient smart pointer,
26 but they don't impede callers from replacing it with its more flexible sibling. (For
27 information about `std::shared_ptr`, proceed to Item 21.)

28 Things to Remember

- 29 ♦ `std::unique_ptr` is a small, fast, move-only smart pointer for managing re-
30 sources with exclusive-ownership semantics.

- 1 ♦ By default, resource destruction takes place via `delete`, but custom deleters
- 2 can be specified. Stateful deleters and function pointers as deleters increase
- 3 the size of `std::unique_ptr` objects.
- 4 ♦ Converting a `std::unique_ptr` to a `std::shared_ptr` is easy.

5 **Item 21: Use `std::shared_ptr` for shared-ownership re- 6 source management.**

7 Programmers using languages with garbage collection point and laugh at what C++
8 programmers go through to prevent resource leaks. “How primitive!”, they jeer.
9 “Didn’t you get the memo from Lisp in the 1960s? Machines should manage re-
10 source lifetimes, not humans.” C++ developers roll their eyes. “You mean the
11 memo where the only resource is memory and the timing of resource reclamation
12 is nondeterministic? We prefer the generality and predictability of destructors,
13 thank you.” But our bravado is part bluster. Garbage collection really is convenient,
14 and manual lifetime management really can seem akin to constructing a mnemonic
15 memory circuit using stone knives and bear skins. Why can’t we have the best of
16 both worlds: a system that works automatically (like garbage collection), yet ap-
17 plies to all resources and has predictable timing (like destructors)?

18 `std::shared_ptr` is the C++11 way of binding these worlds together. An object
19 accessed via `std::shared_ptr`s has its lifetime managed by those pointers
20 through *shared ownership*. No specific `std::shared_ptr` owns the object. Instead,
21 all `std::shared_ptr`s pointing to it collaborate to ensure its destruction at the
22 point where it’s no longer needed. When the last `std::shared_ptr` pointing to an
23 object stops pointing there (e.g., because the `std::shared_ptr` is destroyed or
24 made to point to a different object), that `std::shared_ptr` destroys the object it
25 points to. As with garbage collection, clients need not concern themselves with
26 managing the lifetime of pointed-to objects, but as with destructors, the timing of
27 the objects’ destruction is deterministic.

28 A `std::shared_ptr` can tell whether it’s the last one pointing to a resource by
29 consulting the resource’s *reference count*: a value associated with the resource that
30 keeps track of how many `std::shared_ptr`s point to it. `std::shared_ptr` con-
31 structors increment this count (usually—see below), `std::shared_ptr` destruc-

1 tors decrement it, and copy assignment operators do both. (If `sp1` and `sp2` are
2 `std::shared_ptr`s, the assignment “`sp1 = sp2`” modifies `sp1` such that it points
3 to the object pointed to by `sp2`. The net effect of the assignment, then, is that the
4 reference count for the object originally pointed to by `sp1` is decremented, while
5 that for the object pointed to by `sp2` is incremented.) If a `std::shared_ptr` sees a
6 reference count of zero after performing a decrement, no more
7 `std::shared_ptr`s point to the resource, so the `std::shared_ptr` destroys it.

8 The existence of the reference count has performance implications:

- 9 • **std::shared_ptr**s are twice the size of a raw pointer, because they internally
10 contain a raw pointer to the resource as well as a raw pointer to the resource’s
11 reference count.[†]

- 12 • **Memory for the reference count must be dynamically allocated.** Conceptually,
13 the reference count is associated with the object being pointed to, but
14 pointed-to objects know nothing about this. They thus have no place to store a
15 reference count. Storage for the count is hence dynamically allocated. Item 23
16 explains that the cost of this allocation (but not the allocation itself) is avoided
17 when the `std::shared_ptr` is created by `std::make_shared`, but there are
18 situations where `std::make_shared` can’t be used. Either way, the reference
19 count is stored as dynamically allocated runtime data.

- 20 • **Increments and decrements of the reference count must be atomic**, be-
21 cause there can be simultaneous readers and writers in different threads. For
22 example, a `std::shared_ptr` pointing to a resource in one thread could be
23 executing its destructor (hence decrementing the reference count for the re-
24 source it points to), while, in a different thread, a `std::shared_ptr` to the
25 same object could be copied (and therefore incrementing the same reference
26 count). Atomic operations are typically slower than non-atomic operations, so

[†] This implementation is not required by the Standard, but there are sound technical reasons why every Standard Library implementation I’m familiar with employs it.

1 even though reference counts are usually only a word in size, you should assume
2 that reading and writing them is comparatively costly.

3 Did I pique your curiosity when I wrote that `std::shared_ptr` constructors only
4 “usually” increment the reference count for the object they point to? Creating a
5 new `std::shared_ptr` pointing to an object always yields one more
6 `std::shared_ptr` pointing to that object, so why mustn’t we *always* increment
7 the reference count?

8 Move construction, that’s why. Move-constructing a `std::shared_ptr` from another
9 `std::shared_ptr` sets the source `std::shared_ptr` to null, and that means that the old
10 `std::shared_ptr` stops pointing to the resource at the moment the new
11 `std::shared_ptr` starts. As a result, no reference count manipulation is required.
12 Moving `std::shared_ptr`s is therefore faster than copying them: copying requires
13 incrementing the reference count, but moving doesn’t. This is as true for assignment
14 as for construction, so move construction is faster than copy construction, and move
15 assignment is faster than copy assignment.

16 Like `std::unique_ptr` (see Item 20), `std::shared_ptr` uses `delete` as its default
17 resource-destruction mechanism, but it also supports custom deleters. The design
18 of this support differs from that for `std::unique_ptr`, however. For
19 `std::unique_ptr`, the type of the deleter is part of the type of the smart pointer.
20 For `std::shared_ptr`, it’s not:

```
21 auto loggingDel = [](Widget *pw)           // custom deleter
22 {                                         // (as in Item 20)
23     makeLogEntry(pw);
24     delete pw;
25 };
26 std::unique_ptr<                         // deleter type is
27     Widget, decltype(loggingDel)>          // part of ptr type
28 > upw(new Widget, loggingDel);
29 std::shared_ptr<Widget>                  // deleter type is not
30 spw(new Widget, loggingDel);             // part of ptr type
```

31 The `std::shared_ptr` design is more flexible. Consider two
32 `std::shared_ptr<Widget>`s, each with a custom deleter of a unique type (e.g.,
33 because the custom deleters are specified via lambda expressions):

```
1 auto customDeleter1 = [](Widget *pw) { ... }; // custom deleters,
2 auto customDeleter2 = [](Widget *pw) { ... }; // each with a
3 // different type
4 std::shared_ptr<Widget> pw1(new Widget, customDeleter1);
5 std::shared_ptr<Widget> pw2(new Widget, customDeleter2);
```

6 Because `pw1` and `pw2` have the same type, they can be placed in a container of ob-
7 jects of that type:

```
8 std::vector<std::shared_ptr<Widget>> vpw { pw1, pw2 };
```

9 They could also be assigned to one another, and they could each be passed to a
10 function taking a parameter of type `std::shared_ptr<Widget>`. None of these
11 things can be done with `std::unique_ptr`s that differ in the types of their cus-
12 tom deleters, because the type of the custom deleter would affect the type of the
13 `std::unique_ptr`.

14 In another difference from `std::unique_ptr`, specifying a custom deleter doesn't
15 change the size of a `std::shared_ptr` object. Regardless of deleter, a
16 `std::shared_ptr` object is two pointers in size. That's great news, but it should
17 make you vaguely uneasy. Custom deleters can be function objects, and function
18 objects can contain arbitrary amounts of data. That means they can be arbitrarily
19 large. How can a `std::shared_ptr` refer to a deleter of arbitrary size without us-
20 ing any more memory?

21 It can't. It may have to use more memory. However, that memory isn't part of the
22 `std::shared_ptr` object. It's on the heap (or, if the creator of the
23 `std::shared_ptr` took advantage of `std::shared_ptr` support for custom allo-
24 cators, it's wherever the memory managed by that allocator is located). I remarked
25 earlier that a `std::shared_ptr` object contains a pointer to the reference count
26 for the object it points to. That's true, but it's a bit misleading, because the refer-
27 ence count is part of a larger data structure known as the *control block*. There's a
28 control block for each object managed by `std::shared_ptr`s. The control block
29 contains, in addition to the reference count, a copy of the custom deleter, if one has
30 been specified. If a custom allocator was specified, the control block contains a
31 copy of that, too. (As Item 23 explains, it also contains a secondary reference count
32 known as the weak count, but we'll ignore that in this Item.)

1 The control block is created by the function creating the first `std::shared_ptr` to
2 the object. At least that's what's supposed to happen. In general, it's impossible for
3 a function creating a `std::shared_ptr` to an object to know whether some other
4 `std::shared_ptr` already points to that object, so the following rules for control
5 block creation are used:

- 6 • **`std::make_shared` (see Item 23) always creates a control block.** It manu-
7 factures a new object to point to, so there is certainly no control block for that
8 object at the time `std::make_shared` is called.
- 9 • **When a `std::shared_ptr` constructor is called with a raw pointer, it creates
10 a control block for the object the raw pointer points to.** The thinking here
11 is that if you wanted to create a `std::shared_ptr` from an object that already
12 had a control block, you'd use a `std::shared_ptr` or a `std::weak_ptr` (see
13 Item 22) as a constructor argument, not a raw pointer. `std::shared_ptr`
14 constructors taking `std::shared_ptr`s or `std::weak_ptr`s as constructor
15 arguments don't create new control blocks, because they know that the smart
16 pointers passed to them already point to control blocks.

17 If a `std::shared_ptr` is created from a raw pointer that's null, by the way,
18 there's no object to associate the control block with, so no control block is cre-
19 ated.

20 A consequence of these rules is that constructing more than one
21 `std::shared_ptr` from a single raw pointer gives you a complimentary ride on
22 the particle accelerator of undefined behavior, because the pointed-to object will
23 have multiple control blocks. Multiple control blocks means multiple reference
24 counts, and multiple reference counts means the object will be destroyed multiple
25 times (once for each reference count). That means that code like this is bad, bad,
26 bad:

```
27 auto pw = new Widget;  
28 ...  
29 std::shared_ptr<Widget> spw1(pw, loggingDel); // create control  
30 // block for *pw  
31 ...
```

```
1 std::shared_ptr<Widget> spw2(pw, loggingDel); // create 2nd
2 // control block
3 // for *pw!
```

4 The creation of the raw pointer `pw` to a dynamically allocated object is bad, because it runs contrary to the advice behind this entire chapter: to prefer smart pointers to raw pointers. (If you've forgotten the motivation for that advice, turn to page 148 to refresh your memory.) But set that aside. The line creating `pw` is a stylistic abomination, but at least it doesn't cause undefined program behavior. The real problem here is that the constructor for `spw1` is called with a raw pointer, so it creates a control block (and thereby a reference count) for what's pointed to. In this case, that's `*pw` (i.e., the object pointed to by `pw`). In and of itself, that's okay, but the constructor for `spw2` is called with the same raw pointer, so it also creates a control block (hence a reference count) for `*pw`. `*pw` thus has two reference counts, each of which will eventually become zero, and that will ultimately lead to an attempt to destroy `*pw` twice. The second destruction is responsible for the undefined behavior.

17 There are at least two lessons regarding `std::shared_ptr` use here. First, try to
18 avoid passing raw pointers to a `std::shared_ptr` constructor. The usual alternative
19 is to use `std::make_shared` (see Item 23), but in the example above, we're
20 using custom deleters, and that's not possible with `std::make_shared`. Second, if
21 you must pass a raw pointer to a `std::shared_ptr` constructor, pass the result of
22 `new` directly instead of going through a raw pointer variable. If the first part of the
23 code above were rewritten like this,

```
24 std::shared_ptr<Widget> spw1(new Widget, // direct use of new
25 loggingDel);
```

26 it'd be a lot harder to create a second `std::shared_ptr` from the same raw
27 pointer. Instead, the author of the code creating `spw2` would naturally use `spw1` as
28 an initialization argument (i.e., would call the `std::shared_ptr` copy constructor), and that would pose no problem whatsoever:

```
30 std::shared_ptr<Widget> spw2(spw1); // spw2 uses same
31 // control block as spw1
```

1 An especially surprising way that using raw pointer variables as
2 `std::shared_ptr` constructor arguments can lead to multiple control blocks in-
3 volves the `this` pointer. Suppose our program uses `std::shared_ptr`s to man-
4 age `Widget` objects, and we have a data structure that keeps track of `Widgets` that
5 have been processed:

6 `std::vector<std::shared_ptr<Widget>> processedWidgets;`

7 Further suppose that `Widget` has a member function that does the processing:

8 `class Widget {`
9 `public:`
10 `...`
11 `void process();`
12 `...`
13 `}`

14 Here's a reasonable-looking approach for `Widget::process`:

15 `void Widget::process()`
16 `{`
17 `...` // process the `Widget`
18 `processedWidgets.emplace_back(this);` // add it to list of
19 `}` // processed Widgets;
20 // this is wrong!

21 The comment about this being wrong says it all—or at least most of it. (The part
22 that's wrong is the passing of `this`, not the use of `emplace_back`. If you're not fa-
23 miliar with `emplace_back`, see Item 18.) This code will compile, but it's passing a
24 raw pointer (`this`) to a container of `std::shared_ptr`s. The `std::shared_ptr`
25 thus constructed will create a new control block for the pointed-to `Widget`
26 (`*this`). That doesn't sound harmful until you realize that if there are
27 `std::shared_ptr`s outside the member function that already point to that `Widg-`
28 `et`, it's game, set, and match for undefined behavior.

29 The `std::shared_ptr` API includes a facility for just this kind of situation. It has
30 probably the oddest of all names in the Standard C++ Library:
31 `std::enable_shared_from_this`. That's a template for a base class you inherit
32 from if you want a class managed by `std::shared_ptr`s to be able to safely cre-

1 ate a `std::shared_ptr` from a `this` pointer. In our example, `Widget` would inherit from `std::enable_shared_from_this` as follows:

```
3 class Widget: public std::enable_shared_from_this<Widget> {
4 public:
5     ...
6     void process();
7     ...
8 }
```

9 As I said, `std::enable_shared_from_this` is a base class template. Its type parameter is always the name of the class being derived, so `Widget` inherits from `std::enable_shared_from_this<Widget>`. If the idea of a derived class inheriting from a base class templated on the derived class makes your head hurt, try 13 not to think about it. The code is completely legal, and the design pattern behind it 14 is so well established, it has a standard name, albeit one that's almost as odd as 15 `std::enable_shared_from_this`. The name is *The Curiously Recurring Template Pattern (CRTP)*. If you'd like to learn more about it, unleash your search engine, because here we need to get back to `std::enable_shared_from_this`.

18 `std::enable_shared_from_this` defines a member function that creates a 19 `std::shared_ptr` to the current object, but it does it without duplicating control 20 blocks. The member function is `shared_from_this`, and you use it in member 21 functions whenever you want a `std::shared_ptr` that points to the same object 22 as the `this` pointer. Here's a safe implementation of `Widget::process`:

```
23 void Widget::process()
24 {
25     // as before, process the Widget
26     ...
27     // add std::shared_ptr to current object to processedWidgets
28     processedWidgets.emplace_back(shared_from_this());
29 }
```

30 Internally, `shared_from_this` looks up the control block for the current object, 31 and it creates a new `std::shared_ptr` that refers to that control block. The de- 32 sign relies on the current object having an associated control block. For that to be 33 the case, there must be an existing `std::shared_ptr` (e.g., one outside the mem- 34 ber function calling `shared_from_this`) that points to the current object. If no

1 such `std::shared_ptr` exists (i.e., if the current object has no associated control
2 block), `shared_from_this` throws an exception.

3 At this point, you may only dimly recall that our discussion of control blocks was
4 motivated by a desire to understand the costs associated with
5 `std::shared_ptrs`. Now that we understand how to avoid creating too many
6 control blocks, let's return to the original topic.

7 A control block is typically only a few words in size, although custom deleters and
8 allocators may make it larger. The usual control block implementation is more so-
9 phisticated than you might expect. It makes use of inheritance, and there's even a
10 virtual function. (It's used to ensure that the pointed-to object is properly de-
11 stroyed.) That means that using `std::shared_ptrs` also incurs the cost of the
12 machinery for the virtual function used by the control block.

13 Having read about dynamically allocated control blocks, arbitrarily large deleters
14 and allocators, virtual function machinery, and atomic reference count manipula-
15 tions, your enthusiasm for `std::shared_ptrs` may have waned somewhat. That's
16 fine. They're not the best solution to every resource management problem. But for
17 the functionality they provide, `std::shared_ptrs` exact a very reasonable cost.
18 Under typical conditions, where the default deleter and default allocator are used
19 and where the `std::shared_ptr` is created by `std::make_shared`, the control
20 block is only about three words in size, and its allocation is essentially free. (It's
21 incorporated into the memory allocation for the object being pointed to. For de-
22 tails, see Item 23.) Dereferencing a `std::shared_ptr` is no more expensive than
23 dereferencing a raw pointer. Performing an operation requiring a reference count
24 manipulation (e.g., copy construction or copy assignment, destruction) entails one
25 or two atomic operations, but these operations typically map to individual ma-
26 chine instructions, so although they may be expensive compared to non-atomic
27 instructions, they're still just single instructions. The virtual function machinery in
28 the control block is used only once per object managed by `std::shared_ptrs`:
29 when the object is destroyed.

30 In exchange for these rather modest costs, you get automatic lifetime management
31 of dynamically allocated resources. Most of the time, using `std::shared_ptr` is

1 vastly preferable to trying to manage the lifetime of an object with shared owner-
2 ship by hand. If you find yourself doubting whether you can afford use of
3 `std::shared_ptr`, reconsider whether you really need shared ownership. If ex-
4 clusive ownership will do or even *may* do, `std::unique_ptr` is a better choice. Its
5 performance profile is close to that for raw pointers, and “upgrading” from
6 `std::unique_ptr` to `std::shared_ptr` is easy, because a `std::shared_ptr`
7 can be created from a `std::unique_ptr`.

8 The reverse is not true. Once you’ve turned lifetime management of a resource
9 over to a `std::shared_ptr`, there’s no changing your mind. Even if the reference
10 count is one, you can’t reclaim ownership of the resource in order to, say, have a
11 `std::unique_ptr` manage it. The ownership contract between a resource and the
12 `std::shared_ptr`s that point to it is of the ‘til-death-do-us-part variety. No di-
13 vorce, no annulment, no exceptions.

14 Something else `std::shared_ptr`s can’t do is work with arrays. In yet another
15 difference from `std::unique_ptr`, `std::shared_ptr` has an API that’s designed
16 only for pointers to single objects. There’s no `std::shared_ptr<T[]>`. From time
17 to time, “clever” programmers stumble on the idea of using a
18 `std::shared_ptr<T>` to point to an array, specifying a custom deleter to perform
19 an array delete (i.e., `delete []`). This can be made to compile, but it’s a horrible
20 idea. For one thing, `std::shared_ptr` offers no `operator[]`, so indexing into the
21 array requires awkward expressions based on pointer arithmetic. For another,
22 `std::shared_ptr` supports derived-to-base pointer conversions that make sense
23 for single objects, but that open holes in the type system when applied to arrays.
24 (For this reason, the `std::unique_ptr<T[]>` API prohibits such conversions.)
25 Most importantly, given the variety of C++11 alternatives to built-in arrays (e.g.,
26 `std::array`, `std::vector`, `std::string`), declaring a smart pointer to a dumb
27 array is almost always a sign of bad design.

28 **Things to Remember**

- 29 ♦ `std::shared_ptr`s offer convenience approaching that of garbage collection
30 for the shared lifetime management of arbitrary resources.

- 1 ♦ Compared to `std::unique_ptr`, `std::shared_ptr` objects are twice as big,
- 2 incur overhead for control blocks, and require atomic reference count manipu-
- 3 lations.
- 4 ♦ Default resource destruction is via `delete`, but custom deleters are supported.
- 5 The type of the deleter is independent of the type of the `std::shared_ptr`.
- 6 ♦ Avoid creating `std::shared_ptr`s from variables of raw pointer type.

7 **Item 22: Use `std::weak_ptr` for `std::shared_ptr`-like 8 pointers that can dangle.**

9 Paradoxically, it can be convenient to have a smart pointer that acts like a
10 `std::shared_ptr` (see Item 21), but that doesn't participate in the shared own-
11 ership of the pointed-to resource. In other words, a pointer like
12 `std::shared_ptr` that doesn't affect an object's reference count. This kind of
13 smart pointer has to contend with a problem unknown to `std::shared_ptr`s: the
14 possibility that what it points to has been destroyed. A truly smart pointer would
15 deal with this problem by tracking when it *dangles*, i.e., when the object it is sup-
16 posed to point to no longer exists. That's precisely the kind of smart pointer
17 `std::weak_ptr` is.

18 You may be wondering how a `std::weak_ptr` could be useful. You'll probably
19 wonder even more when you examine the `std::weak_ptr` API. It looks anything
20 but smart. `std::weak_ptr`s can't be dereferenced, nor can they be tested for null-
21 ness. That's because `std::weak_ptr` isn't a standalone smart pointer. It's an
22 augmentation of `std::shared_ptr`.

23 The relationship begins at birth. `std::weak_ptr`s are typically created from
24 `std::shared_ptr`s. They point to the same place as the `std::shared_ptr`s ini-
25 tializing them, but they don't affect the reference count of the object they point to:

```
26 auto spw =                      // after spw is constructed,  
27     std::make_shared<Widget>();    // the pointed-to Widget's  
28                                         // ref count (RC) is 1. (See  
29                                         // Item 23 for info on  
30                                         // std::make_shared.)  
31 ...
```

```

1 std::weak_ptr<Widget> wpw(spw); // wpw points to same Widget
2 // as spw. RC remains 1
3 ...
4 spw = nullptr; // RC goes to 0, and the
5 // Widget is destroyed.
6 // wpw now dangles
7 std::weak_ptrs that dangle are said to have expired. You can test for this direct-
8 ly,
9 if (!wpw.expired()) ... // if wpw points to an object...
10 but often what you desire is a check to see if a std::weak_ptr has expired and, if
11 it hasn't (i.e., if it's not dangling), to access the object it points to. Alas, this is easier
12 desired than done. Because std::weak_ptrs lacks dereferencing operations,
13 there's no way to write the code. Even if there were, separating the check and the
14 dereference would introduce a race condition: between the call to expired and
15 the dereferencing action, another thread might reassign or destroy the last
16 std::shared_ptr pointing to the object, thus causing that object to be destroyed.
17 In that case, your dereference would yield undefined behavior.
18 What you need is an atomic operation that checks to see if the std::weak_ptr has
19 expired and, if not, gives you access to the object it points to. This is done by creat-
20 ing a std::shared_ptr from the std::weak_ptr. The operation comes in two
21 forms, depending on what you'd like to have happen if the std::weak_ptr has
22 expired when you try to create a std::shared_ptr from it. One form is
23 std::weak_ptr::lock, which returns a std::shared_ptr. The
24 std::shared_ptr is null if the std::weak_ptr has expired:
25 std::shared_ptr<Widget> spw1 = wpw.lock(); // if wpw's expired,
26 // spw1 is null
27 auto spw2 = wpw.lock(); // same as above,
28 // but uses auto
29 The other form is the std::shared_ptr constructor taking a std::weak_ptr as
30 an argument. In this case, if the std::weak_ptr has expired, an exception is
31 thrown:

```

```

1 std::shared_ptr<Widget> spw3(wpw);      // if wpw's expired,
2                                         // throw std::bad_weak_ptr
3 But you're probably still wondering about how std::weak_ptrs can be useful.
4 Consider a factory function that produces smart pointers to read-only objects
5 based on a unique ID. In accord with Item 20's advice regarding factory function
6 return types, it returns a std::unique_ptr:
7 std::unique_ptr<const Widget> loadWidget(WidgetID id);
8 If loadWidget is an expensive call (e.g., because it performs file or database I/O)
9 and it's common for IDs to be used repeatedly, a reasonable optimization would be
10 to write a function that does what loadWidget does, but also caches its results.
11 Clogging the cache with every Widget that has ever been requested can lead to
12 performance problems of its own, however, so another reasonable optimization
13 would be to destroy cached Widgets when they're no longer in use.
14 For this caching factory function, a std::unique_ptr return type is not a good fit.
15 The caller should certainly receive a smart pointer to the cached object, and the
16 caller should certainly determine the lifetime of that object, but the cache needs a
17 pointer to the object, too. The cache's pointer needs to be able to detect when it
18 dangles, because when a factory client is finished using an object returned by the
19 factory, that object will be destroyed, and the corresponding cache entry will dan-
20 gle. The cached pointer should therefore be a std::weak_ptr—a pointer that can
21 detect when it dangles. That means that the factory's return type should be a
22 std::shared_ptr, because std::weak_ptrs can detect when they dangle only
23 when an object's lifetime is managed by std::shared_ptrs.
24 Here's a quick-and-dirty implementation of a caching version of loadWidget:
25 std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
26 {
27     using SPType = std::shared_ptr<const Widget>;
28     using WPType = std::weak_ptr<const Widget>;
29     static std::unordered_map<WidgetID, WPType> cache;
30     auto retPtr = cache[id].lock();      // retPtr points to cached
31                                         // object or is null

```

```
1 if (!retPtr) {                                // if not in
2                                         // cache,
3     retPtr = static_cast<SPTYPE>(loadWidget(id)); // load it
4     cache[id] = static_cast<WPTYPE>(retPtr);      // cache it
5 }
6 return retPtr;
7 }
```

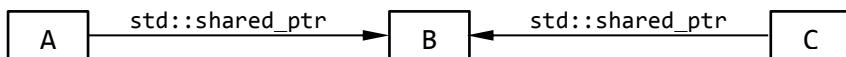
8 This implementation employs one of C++11's hash table containers
9 (`std::unordered_map`), though it doesn't show the `WidgetID` hashing and equal-
10 ity-comparison functions that would also have to be present.

11 There are two smart pointer type conversions in the code, both made obvious by
12 the use of `static_casts`. In the assignment to `retPtr` (a `std::shared_ptr`), the
13 `std::unique_ptr` returned from `loadWidget` is used to create a temporary
14 `std::shared_ptr` (via the alias `SPTYPE`) to act as the source of the assignment. In
15 the following line, `retPtr` (still a `std::shared_ptr`) is used to create a tempo-
16 rary `std::weak_ptr` (via the alias `WPTYPE`) that is moved into the cache (via as-
17 signment to `cache[id]`). Neither cast is doing anything underhanded. Each simply
18 causes a smart pointer object of one type to be constructed from a smart pointer
19 object of a different type.

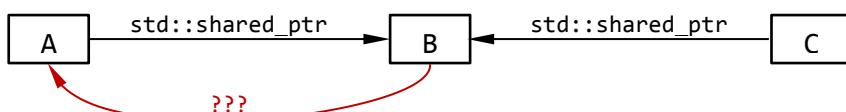
20 This implementation of `fastLoadWidget` ignores the fact that the cache may ac-
21 cumulate expired `std::weak_ptr`s corresponding to `Widgets` that are no longer
22 in use (and have therefore been destroyed). The implementation can be refined,
23 but rather than spend time on an issue that lends no additional insight into
24 `std::weak_ptr`s, let's consider a second use case: the Observer design pattern.
25 The primary components of this pattern are subjects (objects whose state may
26 change) and observers (objects to be notified when state changes occur). In most
27 implementations, each subject contains a data member holding pointers to its ob-
28 servers. That makes it easy for subjects to issue state change notifications. Subjects
29 have no interest in controlling the lifetime of their observers (i.e., when they're
30 destroyed), but they have a great interest in making sure that if an observer gets
31 destroyed, subjects don't try to subsequently access it. A reasonable design is for
32 each subject to hold a container of `std::weak_ptr`s to its observers, thus making

1 it possible for the subject to ensure that the pointers don't dangle when it tries to
2 use them.

3 As a final example of `std::weak_ptr` utility, consider a data structure with ob-
4 jects A, B, and C in it, where A and C share ownership of B and therefore hold
5 `std::shared_ptr` to it:



6 Suppose it'd be useful to also have a pointer from B back to A. What kind of pointer
7 should this be?
8



9
10 There are three choices:

11 • **A raw pointer.** With this approach, if A is destroyed, but C continues to point
12 to B, B will contain a pointer to A that will dangle. B won't be able to detect that,
13 so B may inadvertently dereference the dangling pointer. That would yield un-
14 defined behavior.

15 • **A `std::shared_ptr`.** In this design, A and B contain `std::shared_ptr`s to one
16 another. The resulting `std::shared_ptr` cycle (A points to B and B points to
17 A) will prevent both A and B from being destroyed. Even if A and B are un-
18 reachable from other program data structures (i.e., because C no longer points
19 to B), each will have a reference count of one. If that happens, A and B will have
20 been leaked, for all practical purposes: it will be impossible for the program to
21 access them, yet their resources will never be reclaimed.

22 • **A `std::weak_ptr`.** This avoids both problems above. If A is destroyed, B's point-
23 er back to it will dangle, but B will be able to detect that. Furthermore, though
24 A and B will point to one another, B's pointer won't affect A's reference count,
25 hence can't keep A from being destroyed when `std::shared_ptr`s no longer
26 point to it.

1 The `std::weak_ptr` is clearly the best of these choices. However, it's worth noting
2 that the need to employ `std::weak_ptr`s to break prospective cycles of
3 `std::shared_ptr`s is not terribly common. In strictly hierachal data structures
4 such as trees, child nodes are typically owned only by their parents. When a parent
5 node is destroyed, its child nodes should be destroyed, too. Links from parents to
6 children are thus generally best represented by `std::unique_ptr`s. Back-links
7 from children to parents can be safely implemented as raw pointers, because a
8 child node should never have a lifetime longer than its parent. There's thus no risk
9 of a child node dereferencing a dangling parent pointer.

10 Of course, not all pointer-based data structures are strictly hierarchical, and when
11 that's the case, as well as in situations such as caching and the implementation of
12 lists of observers, it's nice to know that `std::weak_ptr` stands at the ready.

13 From an efficiency perspective, the `std::weak_ptr` story is essentially the same as
14 that for `std::shared_ptr`. `std::weak_ptr` objects are the same size as
15 `std::shared_ptr` objects, they make use of the same control blocks as
16 `std::shared_ptr`s (see Item 21), and operations such as construction, destruction,
17 and assignment involve atomic reference count manipulations. That probably
18 surprises you, because I wrote at the beginning of this Item that `std::weak_ptr`s
19 don't participate in reference counting. Except that's not quite what I wrote. What
20 I wrote was that `std::weak_ptr`s don't participate in the *shared ownership* of ob-
21 jects and hence don't affect the *pointed-to object's reference count*. There's actually
22 a second reference count in the control block, and it's this second reference count
23 that `std::weak_ptr`s manipulate. For details, continue on to Item 23.

24 **Things to Remember**

- 25 ♦ Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle.
26 ♦ Potential use cases for `std::weak_ptr` include caching, observer lists, and the
27 prevention of `std::shared_ptr` cycles.

1 **Item 23: Prefer `std::make_unique` and `std::make_shared`
2 to direct use of `new`.**

3 Let's begin by leveling the playing field for `std::make_unique` and
4 `std::make_shared`. `std::make_shared` is part of C++11, but, sadly,
5 `std::make_unique` isn't. It joined the Standard Library as of C++14. If you're us-
6 ing C++11, never fear, because a basic version of `std::make_unique` is easy to
7 write yourself. Here, look:

```
8 template<typename T, typename... Args>
9 std::unique_ptr<T> make_unique(Args&&... args)
10 {
11     return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
12 }
```

13 As you can see, `make_unique` simply perfect-forwards its arguments to the con-
14 structor of the object being created, constructs a `std::unique_ptr` from the raw
15 pointer `new` produces, and returns the `std::unique_ptr` so created. This form of
16 the function doesn't support arrays or custom deleters (see Item 20), but it
17 demonstrates that, with only a little effort, you can create `make_unique` yourself,
18 if you need to.[†]

19 `std::make_unique` and `std::make_shared` are two of the three *make functions*:
20 functions that take an arbitrary set of arguments, perfect-forward them to the con-
21 structor for a dynamically-allocated object, and return a smart pointer to that ob-
22 ject. The third make function is `std::allocate_shared`. It acts just like
23 `std::make_shared`, except its first argument is an allocator object to be used for
24 the dynamic memory allocation.

25 Even the most trivial comparison of smart pointer creation using and not using a
26 `make` function reveals the first reason why using such functions is preferable. Con-
27 sider:

[†] To create a full-featured `make_unique` with the smallest effort possible, search for the standardization document that gave rise to it, then copy the implementation you'll find there. The document you want is N3588 by Stephan T. Lavavej, dated 2013-03-15.

```
1 auto upw1(std::make_unique<Widget>()); // with make func
2 std::unique_ptr<Widget> upw2(new Widget); // without make func
3
4 auto spw1(std::make_shared<Widget>()); // with make func
5 std::shared_ptr<Widget> spw2(new Widget); // without make func
6 I've highlighted the essential difference: the versions using new repeat the type
7 being created, but the make functions don't. Repeating types runs afoul of a key
8 tenet of software engineering: code duplication should be avoided. Duplication in
9 source code increases compilation times, can lead to bloated object code, and gen-
10 erally renders a code base more difficult to work with. It often evolves into incon-
11 sistency in a code base, and inconsistency in a code base often leads to bugs. Be-
12 sides, typing something twice takes more effort than typing it once, and who's not
13 a fan of reducing their typing burden?
```

14 The second reason to prefer make functions has to do with exception safety. Sup-
15 pose we have a function to process a `Widget` in accord with some priority:

```
16 void processWidget(std::shared_ptr<Widget> spw, int priority);
17 Passing the std::shared_ptr by value may look suspicious, but Item 17 explains
18 that if processWidget always makes a copy of the std::shared_ptr (e.g., by
19 storing it in a data structure tracking Widgets that have been processed), this can
20 be a reasonable design choice.
```

21 Now suppose we have a function to compute the relevant priority,

```
22 int computePriority();
23 and we use that in a call to processWidget that uses new instead of
24 std::make_shared:
```

```
25 processWidget(std::shared_ptr<Widget>(new Widget), // potential
26                  computePriority()); // resource
27                                         // leak!
```

28 As the comment indicates, this code could leak the `Widget` conjured up by `new`. But
29 how? Both the calling code and the called function are using `std::shared_ptr`s,
30 and `std::shared_ptr`s are designed to prevent resource leaks. They automati-

1 call^y destroy what they point to when the last `std::shared_ptr` pointing there
2 goes away. If everybody is using `std::shared_ptr`s everywhere, how can this
3 code leak?

4 The answer has to do with compilers' translation of source code into object code.
5 At runtime, the arguments for a function must be evaluated before the function can
6 be invoked, so in the call to `processWidget`, the following things must occur be-
7 fore `processWidget` can begin execution:

- 8 • The expression “`new Widget`” must be evaluated, i.e., a `Widget` must be creat-
9 ed on the heap.
- 10 • The constructor for the `std::shared_ptr<Widget>` responsible for manag-
11 ing the pointer produced by `new` must be executed.
- 12 • `computePriority` must run.

13 Compilers are not required to generate code that executes them in this order. “`new`
14 `Widget`” must be executed before the `std::shared_ptr` constructor may be
15 called, because the result of that `new` is used as an argument to that constructor,
16 but `computePriority` may be executed before those calls, after them, or, crucial-
17 ly, *between* them. That is, compilers may emit code to execute the operations
18 above in this order:

- 19 1. Perform “`new Widget`”.
- 20 2. Execute `computePriority`.
- 21 3. Run `std::shared_ptr` constructor.

22 If such code is generated and, at runtime, `computePriority` produces an excep-
23 tion, the dynamically allocated `Widget` from Step 1 will be leaked, because it will
24 never be stored in the `std::shared_ptr` that's supposed to start managing it in
25 Step 3.

26 Using `std::make_shared` avoids this problem. Calling code would look like this:

```
27 processWidget(std::make_shared<Widget>(), // no potential  
28                 computePriority()); // resource leak
```

1 At runtime, either `std::make_shared` or `computePriority` will be called first. If
2 it's `std::make_shared`, the raw pointer to the dynamically allocated `Widget` is
3 safely stored in the returned `std::shared_ptr` before `computePriority` is
4 called. If `computePriority` then yields an exception, the `std::shared_ptr` de-
5 structor will see to it that the `Widget` it owns is destroyed. And if `computePrior-
6 ity` is called first and yields an exception, `std::make_shared` will not be invoked,
7 and there will hence be no dynamically allocated `Widget` to worry about.

8 If we replace `std::shared_ptr` and `std::make_shared` with
9 `std::unique_ptr` and `std::make_unique`, exactly the same reasoning applies.
10 Using `std::make_unique` instead of `new` is thus just as important in writing ex-
11 ception-safe code as using `std::make_shared`.

12 A special feature of `std::make_shared` (compared to direct use of `new`) is im-
13 proved efficiency. Using `std::make_shared` allows compilers to generate smaller,
14 faster code that employs leaner data structures. Consider the following direct use
15 of `new`:

16 `std::shared_ptr<Widget> spw(new Widget);`

17 It's obvious that this code entails a memory allocation, but it actually requires two.
18 Item 21 explains that every `std::shared_ptr` points to a control block contain-
19 ing, among other things, the reference count for the pointed-to object. Memory for
20 this control block is allocated in the `std::shared_ptr` constructor. Direct use of
21 `new`, then, requires one memory allocation for the `Widget` and a second allocation
22 for the control block.

23 If `std::make_shared` is used instead,

24 `auto spw = std::make_shared<Widget>();`

25 one allocation suffices. That's because `std::make_shared` allocates a single
26 chunk of memory to hold both the `Widget` object and the control block. This opti-
27 mization reduces the static size of the program, because the code contains only
28 one memory allocation call, and it increases the speed of the executable code, be-
29 cause memory is allocated only once. Furthermore, using `std::make_shared` ob-

1 viates the need for some of the bookkeeping information in the control block, thus
2 reducing the total memory footprint for the program.

3 The efficiency analysis for `std::make_shared` is equally applicable to
4 `std::allocate_shared`, so the performance advantages of `std::make_shared`
5 extend to that `make` function, as well.

6 Clearly, the argument for preferring `make` functions over direct use of `new` is a
7 strong one. Despite their software engineering, exception-safety, and efficiency
8 advantages, however, this Item's guidance is to *prefer* the `make` functions, not to
9 rely on them exclusively. That's because there are circumstances where they can't
10 or shouldn't be used.

11 For example, none of the `make` functions permit the specification of custom
12 deleters (see Items 20 and 21), but both `std::unique_ptr` and
13 `std::shared_ptr` have constructors that do. Given a custom deleter for a `Widget`,

15 `auto widgetDeleter = [](Widget* pw) { ... };`

16 creating a smart pointer using it is straightforward using `new`:

17 `std::unique_ptr<Widget, decltype(widgetDeleter)>`
18 `upw(new Widget, widgetDeleter);`
19 `std::shared_ptr<Widget> spw(new Widget, widgetDeleter);`

20 There's no way to do the same thing with a `make` function.

21 Item 7 explains that when creating an object whose type overloads constructors
22 both with and without `std::initializer_list` parameters, creating the object
23 using braces calls the `std::initializer_list` constructor, while creating the
24 object using parentheses calls the non-`std::initializer_list` constructor. The
25 `make` functions perfect-forward their parameters to an object's constructor, but do
26 they do so using parentheses or using braces? For some types, this seemingly trivial
27 syntactic detail can make a big difference. For example, in these calls,

28 `auto upv = std::make_unique<std::vector<int>>(10, 20);`

29 `auto spv = std::make_shared<std::vector<int>>(10, 20);`

1 do the resulting smart pointers point to `std::vectors` with 10 elements, each of
2 value 20, or to `std::vectors` with two elements, one with value 10 and the other
3 with value 20? Or is the result indeterminate?

4 The good news is that it's not indeterminate: both calls create `std::vectors` of
5 size 10 with all values set to 20. That means that within the `make` functions, the
6 perfect forwarding code uses parentheses, not braces. The bad news is that if you
7 want to achieve the effect of using braces, you must use `new` directly. Using a `make`
8 function would require the ability to perfect-forward a braced initializer through a
9 function, but, as Item 32 explains, braced initializers can't be perfect-forwarded.

10 For `std::unique_ptr`, these two scenarios (custom deleters and braced initializ-
11 ers) are the only ones where its `make` function can't be used. For
12 `std::shared_ptr` and its `make` functions, there are two more. Both are edge cas-
13 es, but some developers live on the edge, and you may be one of them.

14 Some classes define their own versions of `operator new` and `operator delete`.
15 The presence of these functions implies that the global memory allocation and
16 deallocation routines for objects of these types are inappropriate. Often, these
17 class-specific routines are designed only to allocate and deallocate chunks of
18 memory of precisely the size of objects of the class, e.g., `operator new` and `opera-`
19 `tor delete` for class `Widget` are often designed only to handle allocation and
20 deallocation of chunks of memory of exactly size `sizeof(Widget)`. Such routines
21 are a poor fit for `std::shared_ptr`'s support for custom allocation (via
22 `std::allocate_shared`) and deallocation (via custom deleters), because the
23 amount of memory that `std::allocate_shared` requests isn't the size of the dy-
24 namically allocated object, it's the size of that object *plus* the size of a control
25 block. Consequently, using `make` functions to create objects of types with class-
26 specific versions of `operator new` and `operator delete` is typically a poor idea.

27 The size and speed advantages of `std::make_shared` vis-à-vis direct use of `new`
28 stem from `std::shared_ptr`'s control block being placed in the same chunk of
29 memory as the managed object. When that object's reference count goes to zero,
30 the object is destroyed (i.e., its destructor is called). However, the memory it occu-

1 pies can't be released until the control block has also been destroyed, because the
2 same chunk of dynamically allocated memory contains both.

3 As I noted, the control block contains bookkeeping information beyond just the
4 reference count itself. The reference count tracks how many `std::shared_ptr`s
5 refer to the control block, but the control block contains a second reference count,
6 one that tallies how many `std::weak_ptr`s refer to the control block. This second
7 reference count is known as the *weak count*.[†] When a `std::weak_ptr` checks to
8 see if it has expired (see Item 21), it does so by examining the reference count (not
9 the weak count) in the control block that it refers to. If the reference count is zero
10 (i.e., if the pointed-to object has no `std::shared_ptr`s referring to it and has thus
11 been destroyed), the `std::weak_ptr` has expired. Otherwise, it hasn't.

12 As long as `std::weak_ptr`s refer to a control block (i.e., the weak count is greater
13 than zero), that control block must continue to exist. And as long as a control block
14 exists, the dynamically allocated memory containing it must remain allocated. The
15 memory allocated by a `std::shared_ptr make` function, then, can't be deallocated
16 until the last `std::shared_ptr` and the last `std::weak_ptr` referring to it has
17 been destroyed.

18 When the final `std::shared_ptr` referring to a control block is destroyed, the
19 object it referred to is destructed, but the memory that object occupied remains
20 allocated until the final `std::weak_ptr` referring to the control block is also de-
21 stroyed. If the object type is quite large and the time between destruction of the
22 last `std::shared_ptr` and the last `std::weak_ptr` is significant, the result could
23 be notable lags between when an object is destroyed and when the memory it oc-
24 cupied is freed:

[†] In practice, the value of the weak count isn't always equal to the number of outstanding `std::weak_ptr`s, because library implementers have found ways to encode additional information that permits the generation of better code. For purposes of this Item, we'll ignore this discrepancy and assume that the weak count's value is the number of `std::weak_ptr`s referring to the control block.

```
1 class ReallyBigType { ... };

2 auto pBigObj =           // create very large
3     std::make_shared<ReallyBigType>(); // object via
4                             // std::make_shared

5 ...                   // create std::shared_ptrs and std::weak_ptrs to
6 ...                   // large object, use them to work with it

7 ...                   // final std::shared_ptr to object destroyed here,
8 ...                   // but std::weak_ptrs to it remain

9 ...                   // during this period, memory formerly occupied
10 ...                   // by large object remains allocated

11 ...                   // final std::weak_ptr to object destroyed here;
12 ...                   // memory for control block and object is released
```

13 With a direct use of `new`, the memory for the `ReallyBigType` object can be re-
14 leased as soon as the last `std::shared_ptr` to it is destroyed:

```
15 class ReallyBigType { ... };           // as before

16 std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
17                               // create very large
18                               // object via new

19 ...                   // as before, create std::shared_ptrs and
20 ...                   // std::weak_ptrs to object, use them with it

21 ...                   // final std::shared_ptr to object destroyed here,
22 ...                   // but std::weak_ptrs to it remain;
23 ...                   // memory for object is deallocated

24 ...                   // during this period, only memory for the
25 ...                   // control block remains allocated

26 ...                   // final std::weak_ptr to object destroyed here;
27 ...                   // memory for control block is released
```

28 Should you find yourself in a situation where use of `std::make_shared` is impos-
29 sible or inappropriate, you'll want to guard yourself against the kind of exception-
30 safety problems we saw earlier. The best way to do that is to make sure that when
31 you use `new` directly, you pass the result to a smart pointer constructor in *a state-*
32 *ment that does nothing else*. This prevents compilers from generating code corre-
33 sponding to operations being inserted between the use of `new` and the call of the
34 constructor for the smart pointer that will manage the newed object.

1 As an example, consider a minor revision to the exception-unsafe call to the pro-
2 cessWidget function we examined earlier. This time, we'll specify a custom
3 deleter:

4 `void cusDel(Widget *ptr);` // custom deleter

5 Here's the exception-unsafe call:

6 `processWidget(` // as before,
7 `std::shared_ptr<Widget>(new Widget, cusDel),` // potential
8 `computePriority()` // resource
9 `);` // Leak!

10 Recall: if `computePriority` is called after “`new Widget`” but before the
11 `std::shared_ptr` constructor, and if `computePriority` yields an exception, the
12 dynamically allocated `Widget` will be leaked.

13 In this example, the need for a custom deleter precludes use of
14 `std::make_shared`, so the way to avoid the problem is to put the allocation of the
15 `Widget` and the construction of the `std::shared_ptr` into their own statement,
16 then call `processWidget` with the `std::shared_ptr` object:

17 `std::shared_ptr<Widget> spw(new Widget, cusDel);`
18 `processWidget(spw, computePriority());`

19 This works, because a `std::shared_ptr` assumes ownership of the raw pointer
20 passed to its constructor, even if that constructor yields an exception. In this ex-
21 ample, if `pw`'s constructor throws an exception (e.g., due to an inability to dynami-
22 cally allocate memory for a control block), it's still guaranteed that `cusDel` will be
23 invoked on the pointer resulting from “`new Widget`”.

24 That's interesting and worth knowing, but it's also typically irrelevant, because in
25 most cases, there's no reason not to use a `make` function. And unless you have a
26 compelling reason for doing otherwise, using a `make` function is what you should
27 do.

- 1 **Things to Remember**
- 2 ♦ Compared to direct use of `new`, the `make` functions eliminate source code du-
3 plication, improve exception safety, and, for `std::make_shared` and
4 `std::allocate_shared`, generate code that's smaller and faster.
- 5 ♦ Situations where use of the `make` functions is inappropriate include the need to
6 specify custom deleters and a desire to pass braced initializers.
- 7 ♦ For `std::shared_ptr`s, additional situations where `make` functions may be
8 ill-advised include (1) classes with custom memory management and (2) sys-
9 tems with memory concerns, very large objects, and `std::weak_ptr`s that
10 outlive the corresponding `std::shared_ptr`s.

11 **Item 24: When using the Pimpl Idiom, define special mem-
12 ber functions in the implementation file.**

13 If you've ever had to combat excessive build times, you're familiar with the *Pimpl*
14 ("pointer to implementation") *Idiom*. That's the technique whereby you replace the
15 data members of a class with a pointer to an implementation class (or struct), put
16 the data members that used to be in the primary class into the implementation
17 class, and access those data members indirectly through the pointer. For example,
18 suppose `Widget` looks like this:

```
19  class Widget {                                // in header "widget.h"
20  public:
21      Widget();
22      ...
23
24  private:
25      std::string name;
26      std::vector<double> data;
27      Gadget g1, g2, g3;
28  };
```

29 Because `Widget`'s data members are of types `std::string`, `std::vector`, and
30 `Gadget`, headers for those types must be present for `Widget` to compile, and that
31 means that `Widget` clients must `#include <string>`, `<vector>`, and `gadget.h`.
32 Those headers increase the compilation time for `Widget` clients, plus they make
33 those clients dependent on the contents of the headers. If a header's content
changes, `Widget` clients must recompile. The standard headers `<string>` and

1 <vector> don't change very often, but it could be that `gadget.h` is subject to frequent revision.

3 Applying the Pimpl Idiom in C++98 could have `Widget` replace its data members
4 with a raw pointer to a struct that has been declared, but not defined:

```
5 class Widget {           // still in header "widget.h"
6 public:
7     Widget();
8     ~Widget();           // dtor is needed—see below
9     ...
10
11 private:
12     struct Impl;        // declare implementation struct
13     Impl *pImpl;        // and pointer to it
14 };
```

14 Because `Widget` no longer mentions the types `std::string`, `std::vector`, and
15 `Gadget`, `Widget` clients no longer need to `#include` the headers for these types.
16 That speeds compilation, and it also means that if something in these headers
17 changes, `Widget` clients are unaffected.

18 A type that has been declared, but not defined, is known as an *incomplete type*.
19 `Widget::Impl` is such a type. There are very few things you can do with an incomplete type,
20 but declaring a pointer to it is one of them. The Pimpl Idiom takes advantage of that.

22 Part 1 of the Pimpl Idiom is the declaration of a data member that's a pointer to an incomplete type. Part 2 is the dynamic allocation and deallocation of the object that holds the data members that used to be in the original class. The allocation and deallocation code goes in the implementation file, e.g., for `Widget`, in `widget.cpp`:

```
27 #include "widget.h"           // in impl. file "widget.cpp"
28 #include "gadget.h"
29 #include <string>
30 #include <vector>
31
32 struct Widget::Impl {         // definition of Widget::Impl
33     std::string name;          // with data members formerly
34     std::vector<double> data;   // in Widget
35     Gadget g1, g2, g3;
36 };
```

```
1 Widget::Widget()           // allocate data members for
2   : pImpl(new Impl)        // this Widget object
3 {}
4
5 Widget::~Widget()         // destroy data members for
6 { delete pImpl; }          // this object
7
8 Here I'm showing #include directives to make clear that the overall dependen-
9 cies on the headers for std::string, std::vector, and Gadget continue to ex-
10 ist. However, these dependencies have been moved from widget.h (which is visi-
11 ble to and used by Widget clients) to widget.cpp (which is visible to and used
12 only by the Widget implementer). I've also highlighted the code that dynamically
13 allocates and deallocates the Impl object. The need to deallocate this object when
14 a Widget is destroyed is what necessitates the Widget destructor.
```

```
15 But I've shown you C++98 code, and that reeks of a bygone millennium. It uses raw
16 pointers and raw new and raw delete and it's all just so...raw. This chapter is built
17 on the idea that smart pointers are preferable to raw pointers, and if what we want
18 is to dynamically allocate a Widget::Impl object inside the Widget constructor
19 and have it destroyed at the same time the Widget is, std::unique_ptr (see
20 Item 20) is precisely the tool we need. Replacing the raw pImpl pointer with a
21 std::unique_ptr yields this code for the header file,
```

```
22 class Widget {                      // in "widget.h"
23 public:
24   Widget();
25   ...
26 private:
27   struct Impl;
28   std::unique_ptr<Impl> pImpl;      // use smart pointer
29 };                                    // instead of raw pointer
```

28 and this for the implementation file:

```
30 #include "widget.h"                // in "widget.cpp"
31 #include "gadget.h"
32 #include <string>
33 #include <vector>
34
35 struct Widget::Impl {             // as before
36   std::string name;
37   std::vector<double> data;
```

```

1   Gadget g1, g2, g3;
2 }

3 Widget::Widget()           // per Item 23, create
4 : pImpl(std::make_unique<Impl>()) // std::unique_ptr
5 {}                          // via std::make_unique

6 You'll note that the Widget destructor is no longer present. That's because we
7 have no code to put into it. std::unique_ptr automatically deletes what it points
8 to when it (the std::unique_ptr) is destroyed, so we need not delete anything
9 ourselves. That's one of the attractions of smart pointers: they eliminate the need
10 for us to sully our hands with manual resource release.

11 This code compiles, but, alas, the most trivial client use doesn't:

12 #include "widget.h"

13 Widget w;                  // error!

14 The error message you get depends on the compiler you're using, but the text gen-
15 erally mentions something about applying sizeof or delete to an incomplete
16 type. Those operations aren't among the very few things you can do with such
17 types.

18 This apparent failure of the Pimpl Idiom using std::unique_ptrs is alarming,
19 because (1) std::unique_ptr is advertised as supporting incomplete types, and
20 (2) the Pimpl Idiom is one of std::unique_ptr's most common use cases. Fortu-
21 nately, getting the code to work is easy. All that's required is a basic understanding
22 of the cause of the problem.

23 The issue arises due to the code that's generated when w is destroyed (e.g., goes
24 out of scope). At that point, its destructor is called. In the class definition using
25 std::unique_ptr, we didn't declare a destructor, because we didn't have any
26 code to put into it. In accord with the usual rules for compiler-generated special
27 member functions (see Item 19), the compiler generates a destructor for us. Within
28 that destructor, the compiler inserts code to call the destructor for Widget's da-
29 ta member, pImpl. pImpl is a std::unique_ptr<Widget::Impl>, i.e., a
30 std::unique_ptr using the default deleter. The default deleter is a function that
31 uses delete on the raw pointer inside the std::unique_ptr. Prior to using de-
```

1 `lete`, however, implementations typically have the default deleter employ C++11's
2 `static_assert` to ensure that the raw pointer doesn't point to an incomplete
3 type. When the compiler generates code for the destruction of the `Widget w`, then,
4 it encounters a `static_assert` that fails, and that's what leads to the error mes-
5 sage.[†] This message is associated with the point where `w` is destroyed, because
6 `Widget`'s destructor, like all compiler-generated special member functions, is im-
7 plicitly `inline`. The message itself often refers to the line where `w` is created, be-
8 cause it's the source code explicitly creating the object that leads to its later implic-
9 it destruction.

10 Fixing the problem is easy. You just need to make sure that at the point where the
11 code to destroy the `std::unique_ptr<Widget::Impl>` is generated, `Widg-`
12 `et::Impl` is a complete type. The type becomes complete when its definition has
13 been seen, and `Widget::Impl` is defined inside `widget.cpp`. The key to success-
14 ful compilation, then, is to have the compiler see the body of `Widget`'s destructor
15 (i.e., the place where the compiler will generate code to destroy the
16 `std::unique_ptr` data member) only inside `widget.cpp` after `Widget::Impl`
17 has been defined.

18 Arranging for that is simple. Declare (but don't define!) `Widget`'s destructor in
19 `widget.h`,

```
20 class Widget {                                // as before, in "widget.h"  
21 public:  
22     Widget();  
23     ~Widget();                                // declaration only!  
24     ...  
25 private:                                     // as before  
26     struct Impl;  
27     std::unique_ptr<Impl> pImpl;  
28 };
```

[†] Implementations need not have a `static_assert` in the default deleter. Nor, for that matter, are they required to reject code deleting an incomplete type. But all implementations I'm familiar with take steps to ensure that code involving the destruction of a `std::unique_ptr` to an incomplete type won't compile.

1 and define it in `widget.cpp` after `Widget::Impl` has been defined:

```
2 #include "widget.h"           // as before, in "widget.cpp"
3 #include "gadget.h"
4 #include <string>
5 #include <vector>
6
7 struct Widget::Impl {        // as before, definition of
8     std::string name;         // Widget::Impl
9     std::vector<double> data;
10    Gadget g1, g2, g3;
11 };
12
13 Widget::Widget()            // as before
14 : pImpl(std::make_unique<Impl>())
15 {}
16
17 Widget::~Widget()           // ~Widget definition
18 {}
```

19 This works well, and it requires the least typing, but if you want to emphasize that
20 the compiler-generated destructor would do the right thing—that the only reason
21 you declared it was to cause its definition to be generated in `Widget`'s implemen-
22 tation file, you can define the destructor body with `=default`:

```
20 Widget::~Widget() = default; // same effect as above
21
22 Classes using the Pimpl Idiom are natural candidates for move support, because
23 compiler-generated move operations do exactly what's desired: perform a move
24 on the underlying std::unique_ptr. As Item 19 explains, the declaration of a de-
25 structor in Widget prevents compilers from generating the move operations, so if
26 you want move support, you must declare the functions yourself. Given that the
27 compiler-generated versions would behave correctly, you might be tempted to im-
28 plement them thusly:
```

```
28 class Widget {
29 public:                                // still in
30     Widget();                            // "widget.h"
31     ~Widget();
32
33     Widget(Widget&& rhs) = default;      // right idea,
34     Widget& operator=(Widget&& rhs) = default; // wrong code!
35
36     ...
37 }
```

```
1 private: // as before
2     struct Impl;
3     std::unique_ptr<Impl> pImpl;
4 };

5 This approach leads to the same kind of problem as declaring the class without a
6 destructor, and for the same fundamental reason. The compiler-generated move
7 assignment operator needs to destroy the object pointed to by pImpl before reas-
8 signing it, but in the Widget header file, pImpl points to an incomplete type. The
9 situation is different for the move constructor. The problem there is that compilers
10 typically generate code to destroy pImpl in the event that an exception arises in-
11 side the move constructor, and destroying pImpl requires that Impl be complete.
```

```
12 Because the problem is the same as before, so is the fix: move the definition of the
13 move operations into the implementation file:
```

```
14 class Widget { // still in "widget.h"
15 public:
16     Widget();
17     ~Widget();

18     Widget(Widget&& rhs); // declarations
19     Widget& operator=(Widget&& rhs); // only

20     ...

21 private: // as before
22     struct Impl;
23     std::unique_ptr<Impl> pImpl;
24 };

25
26 #include <string> // as before,
27 ... // in "widget.cpp"

28 struct Widget::Impl { ... }; // as before

29 Widget::Widget() // as before
30 : pImpl(std::make_unique<Impl>())
31 {}

32 Widget::~Widget() = default; // as before

33 Widget::Widget(Widget&& rhs) = default; // definitions
34 Widget& Widget::operator=(Widget&& rhs) = default; //
```

1 The Pimpl Idiom is a way to reduce compilation dependencies between a class's
2 implementation and the class's clients, but, conceptually, use of the idiom doesn't
3 change what the class represents. The original `Widget` class contained
4 `std::string`, `std::vector`, and `Gadget` data members, and, assuming that
5 `Gadgets`, like `std::strings` and `std::vectors`, can be copied, it would make
6 sense for `Widget` to support the copy operations. We have to write these functions
7 ourselves, because (1) compilers won't generate copy operations for classes with
8 move-only types like `std::unique_ptr` and (2) even if they did, the generated
9 functions would copy only the `std::unique_ptr` (i.e., perform a *shallow copy*),
10 and we want to copy what the pointer points to (i.e., perform a *deep copy*).

11 In a ritual that is by now familiar, we declare the functions in the header file and
12 implement them in the implementation file:

```
13 class Widget {                                     // still in "widget.h"
14 public:
15     ...                                         // other funcs, as before
16     Widget(const Widget& rhs);                 // declarations
17     Widget& operator=(const Widget& rhs);      // only
18 private:                                         // as before
19     struct Impl;
20     std::unique_ptr<Impl> pImpl;
21 };
22
23 #include "widget.h"                                // as before,
24 ...                                         // in "widget.cpp"
25 struct Widget::Impl { ... };                      // as before
26 Widget::~Widget() = default;                     // other funcs, as before
27 Widget::Widget(const Widget& rhs)                // copy ctor
28 : pImpl(std::make_unique<Impl>(*rhs.pImpl))
29 {}
30 Widget& Widget::operator=(const Widget& rhs)    // copy operator=
31 {
32     if (this != &rhs) {                           // detect self-assignment
33         *pImpl = *rhs.pImpl;
34     }
35 }
```

```
1     return *this;
2 }
```

3 Both function implementations are conventional. In each case, we simply copy the
4 fields of the `Impl` struct from the source object (`rhs`) to the destination object
5 (`*this`). Rather than copy the fields one by one, we take advantage of the fact that
6 compilers will create the copy operations for `Impl`, and these operations will copy
7 each field automatically. We thus implement `Widget`'s copy operations by calling
8 `Widget::Impl`'s compiler-generated copy operations. In the copy constructor,
9 note that we still follow the advice of Item 23 to prefer use of `std::make_unique`
10 over direct use of `new`, and in the copy assignment operator, we adhere to common
11 C++ convention by doing nothing if we detect that an object is being assigned to
12 itself.

13 For purposes of implementing the Pimpl Idiom, `std::unique_ptr` is the smart
14 pointer to use, because the `pImpl` pointer inside an object (e.g., inside a `Widget`)
15 has exclusive ownership of the corresponding implementation object (e.g., the
16 `Widget::Impl` object). Still, it's interesting to note that if we were to use
17 `std::shared_ptr` instead of `std::unique_ptr` for `pImpl`, we'd find that the ad-
18 vice of this Item no longer applied. There'd be no need to declare a destructor in
19 `Widget`, and without a user-declared destructor, compilers would happily gener-
20 ate the move operations, which would do exactly what we'd want them to. That is,
21 given this code in `widget.h`,

```
22 class Widget {                                // in "widget.h"
23 public:
24     Widget();
25     ...
26     // no declarations for dtor
27     // or move operations
27 private:
28     struct Impl;
29     std::shared_ptr<Impl> pImpl;      // std::shared_ptr
30 };                                         // instead of std::unique_ptr
```

31 and this client code that `#includes` `widget.h`,

```
32 Widget w1;
33 auto w2(std::move(w1));                      // move-construct w2
34 w1 = std::move(w2);                          // move-assign w1
```

1 everything would compile and run as we'd hope: `w1` would be default-constructed,
2 its value would be moved into `w2`, that value would be moved back into `w1`, and
3 then both `w1` and `w2` would be destroyed (thus causing the pointed-to `Widget::Impl`
4 object to be destroyed).

5 The difference in behavior between `std::unique_ptr` and `std::shared_ptr`
6 for `pImpl` pointers stems from the differing ways that these smart pointers sup-
7 port custom deleters. For `std::unique_ptr`, the type of the deleter is part of the
8 type of the smart pointer, and this makes it possible for compilers to generate
9 smaller runtime data structures and faster runtime code. A consequence of this
10 greater efficiency is that pointed-to types must be complete when compiler-
11 generated special functions (e.g., destructors or move operations) are used. For
12 `std::shared_ptr`, the type of the deleter is not part of the type of the smart
13 pointer. This necessitates larger runtime data structures and somewhat slower
14 code, but pointed-to types need not be complete when compiler-generated special
15 functions are employed.

16 For the Pimpl Idiom, there's not really a trade-off between the characteristics of
17 `std::unique_ptr` and `std::shared_ptr`, because the relationship between
18 classes like `Widget` and classes like `Widget::Impl` is exclusive ownership, and
19 that makes `std::unique_ptr` the proper tool for the job. Nevertheless, it's worth
20 knowing that in other situations—situations where shared ownership exists (and
21 `std::shared_ptr` is hence a good design choice), there's no need to jump
22 through the function-definition hoops that use of `std::unique_ptr` entails.

23 **Things to Remember**

- 24 ♦ The Pimpl Idiom decreases build times by reducing compilation dependencies
25 between class implementations and class clients.
- 26 ♦ For `std::unique_ptr` `pImpl` pointers, declare special member functions in
27 the class header, but implement them in the implementation file. Do this even
28 if the default function implementations are acceptable.
- 29 ♦ The above advice applies to `std::unique_ptr`, but not to `std::shared_ptr`.

1 **Chapter 5 Rvalue References, Move Semantics, and** 2 **Perfect Forwarding**

3 When you first learn about them, move semantics and perfect forwarding seem
4 pretty straightforward:

- 5 • **Move semantics** makes it possible for compilers to replace expensive copying
6 operations with less expensive moves. In the same way that copy constructors
7 and copy assignment operators give developers control over what it means to
8 copy objects, move constructors and move assignment operators offer control
9 over the semantics of moving.
- 10 • **Perfect forwarding** makes it possible to write function templates that take
11 arbitrary arguments and forward them to other functions such that the target
12 functions receive exactly the same arguments as were passed to the forward-
13 ing functions.

14 Rvalue references are the glue that ties these two rather disparate features together. They're the underlying language mechanism that makes both move semantics
15 and perfect forwarding possible.

17 The more experience you have with these features, the more you realize that your
18 initial impression was based on only the metaphorical tip of the proverbial ice-
19 berg. The world of move semantics, perfect forwarding, and rvalue references is
20 more nuanced than it appears. `std::move` doesn't move anything, for example,
21 and perfect forwarding is imperfect. Move operations aren't always cheaper than
22 copying; when they are, they're not always as cheap as you'd expect; and they're
23 not always called in a context where moving is valid. The construct `T&&` doesn't
24 always represent an rvalue reference.

25 No matter how far you dig into these features, it can seem that there's always more
26 to uncover. Fortunately, there is a limit to the depths of move semantics, perfect
27 forwarding, and rvalue references. This chapter will take you to the bedrock. Once
28 you arrive, this part of C++11 will make a lot more sense. You'll know the usage
29 conventions for `std::move` and `std::forward`, for example. You'll be comforta-

1 ble with the ambiguous nature of “`T&&`”. You’ll understand the reasons for the sur-
2 prisingly varied behavioral profiles of move operations. All those pieces will come
3 together and fall into place. At that point, you’ll be back where you started, because
4 move semantics, perfect forwarding, and rvalue references will once again seem
5 pretty straightforward. But this time, they’ll stay that way.

6 In the Items in this chapter, it’s especially important to bear in mind that a param-
7 eter is always an lvalue, even if its type is an rvalue reference. That is, given

8 `void f(Widget&& w);`

9 the parameter `w` is an lvalue, even though its type is rvalue-reference-to-`Widget`.
10 (If this surprises you, please review the overview of lvalues and rvalues that be-
11 gins on page **Error! Bookmark not defined..**)

12 **Item 25: Understand `std::move` and `std::forward`.**

13 It’s useful to approach `std::move` and `std::forward` in terms of what they *don’t*
14 do. `std::move` doesn’t move anything. `std::forward` doesn’t forward anything.
15 At runtime, neither does anything at all. That’s because they generate no executa-
16 ble code. Not a single byte.

17 `std::move` and `std::forward` are merely functions that perform casts.
18 `std::move` unconditionally casts its argument to an rvalue, while `std::forward`
19 performs this cast only if a particular condition is fulfilled. That’s it. The explana-
20 tion leads to a new set of questions, but, fundamentally, that’s the complete story.

21 To make the story more concrete, here’s a sample implementation of `std::move`
22 in C++11. It’s not fully conforming to the details of the Standard, but it’s very close.

```
23 template<typename T> // in namespace std
24 typename remove_reference<T>::type&&
25 move(T&& param)
26 {
27     using ReturnType = // alias declaration;
28     typename remove_reference<T>::type&&; // see Item 9
29
30     return static_cast<ReturnType>(param);
31 }
```

1 I've highlighted two parts of the code for you. One is the name of the function, be-
2 cause the return type specification is rather noisy, and I don't want you to lose
3 your bearings in the din. The other thing is the cast that comprises the essence of
4 the function. As you can see, `std::move` takes a reference to an object (a universal
5 reference, to be precise (see Item 26)) and it returns a reference to the same ob-
6 ject.

7 The “`&&`” part of the function's return type implies that `std::move` returns an
8 rvalue reference, but, as Item 30 explains, if the type `T` happens to be an lvalue ref-
9 erence, `T&&` would become an lvalue reference. To prevent this from happening,
10 the type trait (see Item 9) `std::remove_reference` is applied to `T`, thus ensuring
11 that “`&&`” is applied to a type that isn't a reference. That guarantees that
12 `std::move` truly returns an rvalue reference, and that's important, because rvalue
13 references returned from functions are rvalues. (Why? Because the C++ Standard
14 says they are.) Thus, `std::move` casts its argument to an rvalue, and that's all it
15 does.

16 As an aside, `std::move` can be implemented with less fuss in C++14. Thanks to
17 function return type deduction (see Item 3) and to the Standard Library's alias
18 template `std::remove_reference_t` (see Item 9), `std::move` can be written
19 this way:

```
20 template<typename T> // C++14 only; still
21 decltype(auto) move(T&& param) // in namespace std
22 {
23     using ReturnType = remove_reference_t<T>&&;
24     return static_cast<ReturnType>(param);
25 }
```

26 Easier on the eyes, no?

27 Because `std::move` does nothing but cast its argument to an rvalue, there have
28 been suggestions that a better name for it might be something like `rvalue_cast`.
29 Be that as it may, the name we have is `std::move`, so it's important to remember
30 what `std::move` does and doesn't do. It does cast. It doesn't move.

31 Of course, rvalues are candidates for moving, so applying `std::move` to an object
32 tells the compiler that that object is eligible to be moved from. That's why

`std::move` has the name it does: to make it easy to designate objects that may be moved from.

3 Actually, rvalues are only *usually* candidates for moving. Suppose you're writing a
4 function taking a `std::string` parameter, and you know that inside your func-
5 tion, you'll copy that parameter. Flush with the advice proffered by Item 17, you
6 declare a by-value parameter:

```
7 void f(std::string s);           // s to be copied, so per Item 17,  
8                                // pass by value
```

9 But suppose you also know that inside `f`, you only need to read `s`'s value; you'll
10 never need to modify it. In accord with the time-honored tradition of using `const`
11 whenever possible, you revise your declaration such that `s` is `const`:

```
12 void f(const std::string s);
```

Finally, assume that at the end of `f`, `s` will be copied into some data structure. Except that you don't want to pay for the copy, so, again in accord with Item 17, you apply `std::move` to `s` to turn it into an rvalue:

```
16 struct SomeDataStructure {
17     std::string name;
18     ...
19 };
20 SomeDataStructure sds;
21 void f(const std::string s)
22 {
23     ... // read operations on s
24     sds.name = std::move(s); // "move" s into sds.name; this code
25 } // doesn't do what it seems to!
```

26 This code compiles. This code links. This code runs. This code sets `sds.name` to
27 the value you expect. The only thing separating this code from a perfect realization
28 of your vision is that `s` is not moved into `sds.name`, it's *copied*. Sure, `s` is cast to an
29 rvalue by `std::move`, but `s` is declared as a `const std::string`, so before the
30 cast, `s` is an lvalue `const std::string`, and the result of the cast is an rvalue
31 `const std::string`, but throughout it all, `s` remains `const`.

1 Consider the effect that has when compilers have to determine which
2 `std::string` assignment operator to call. There are two possibilities:

```
3 class string {           // std::string is actually a
4     public:             // typedef for std::basic_string<char>
5     ...
6     string& operator=(const string& rhs); // copy assignment
7     string& operator=(string&& rhs);      // move assignment
8     ...
9 };
```

10 In our function `f`, the result of `std::move(s)` is an rvalue of type `const`
11 `std::string`. That rvalue can't be passed to `std::string`'s move assignment
12 operator, because the move assignment operator takes an rvalue reference to a
13 *non-const* `std::string`. The rvalue can, however, be passed to the copy assign-
14 ment operator, because an lvalue reference-to-`const` is permitted to bind to a
15 `const` rvalue. The statement

```
16 sds.name = std::move(s);
```

17 therefore invokes the *copy* assignment operator in `std::string`, even though `s`
18 has been cast to an rvalue! Such behavior is essential to maintaining `const`-
19 correctness. Moving a value out of an object generally modifies the object, so the
20 language should not permit `const` objects to be passed to functions (such as move
21 assignment operators) that could modify them.

22 There are two lessons to be drawn from this example. First, don't declare objects
23 `const` if you want to be able to move from them. Move requests on `const` objects
24 are silently transformed into copy operations.

25 Second, `std::move` not only doesn't actually move anything, it doesn't even guar-
26 antee that the object it's casting will be eligible to be moved. The only thing you
27 know for sure about an object that has been `std::moved` is that it's an rvalue.

28 The story for `std::forward` is similar to that for `std::move`, but whereas
29 `std::move` *unconditionally* casts its argument to an rvalue, `std::forward` does it
30 only under certain conditions. `std::forward` is a *conditional* cast. To understand
31 when it casts and when it doesn't, recall how `std::forward` is typically used. The

1 most common scenario is a function template taking a universal reference parameter that is to be passed to another function:

```
3 void process(const Widget& lvalParam);      // process lvalues
4 void process(Widget&& rvalParam);           // process rvalues
5 template<typename T>                      // template that passes
6 void logAndProcess(T&& param)             // param to process
7 {
8     auto now =                                // get current time
9         std::chrono::system_clock::now();
10    makeLogEntry("Calling 'process'", now);
11    process(std::forward<T>(param));
12}
13
```

14 Consider two calls to `logAndProcess`, one with an lvalue, the other with an rvalue:

```
16 Widget w;
17 logAndProcess(w);                         // call with lvalue
18 logAndProcess(std::move(w));               // call with rvalue
```

19 Inside `logAndProcess`, the parameter `param` is passed to the function `process`.
20 `process` is overloaded for lvalues and rvalues. When we call `logAndProcess`
21 with an lvalue, we naturally expect that lvalue to be forwarded to `process` as an
22 lvalue, and when we call `logAndProcess` with an rvalue, we expect the rvalue
23 overload of `process` to be invoked.

24 But `param`, like all function parameters, is an lvalue. Every call to `process` inside
25 `logAndProcess` will thus want to invoke the lvalue overload for `process`. To prevent this,
26 we need a mechanism for `param` to be cast to an rvalue if and only if the argument
27 with which `param` was initialized—the argument passed to `logAndProcess`—was an rvalue.
28 This is precisely what `std::forward` does. That's why `std::forward` is a *conditional* cast:
29 it casts to an rvalue only if its argument was initialized with an rvalue.
30

31 You may wonder how `std::forward` can know whether its argument was initialized
32 with an rvalue. In the code above, for example, how can `std::forward` tell
33 whether `param` was initialized with an lvalue or an rvalue? The brief answer is that

1 that information is encoded in `logAndProcess`'s template parameter `T`. That pa-
2 rameter is then passed to `std::forward`, which recovers the encoded infor-
3 mation. Item 30 covers all the details, but for now, simply accept that `T` will be an
4 lvalue reference only if `param` was initialized with an lvalue. That means that
5 `std::forward` can, in concept, be implemented something like this:

```
6 template<typename T> // conceptual impl. of
7     T&& // std::forward (in
8     forward(T&& param) // namespace std)
9 {
10     if (is_lvalue_reference<T>::value) { // if T indicates lvalue
11         return param; // return param as lvalue
12     } else { // else
13         return move(param); // return param as rvalue
14     }
15 }
```

16 Don't be misled by `forward`'s return type declaration into thinking that it returns
17 an rvalue reference. Item 30 will make clear that when an lvalue is being returned
18 from this function, `std::forward`'s return type will be an lvalue reference.

19 Note that this is a *conceptual* implementation. This code won't even compile for
20 some uses, and when it does compile, it'll generate runtime code. Real
21 `std::forward` implementations compile for all uses, and they never generate
22 runtime code. For purposes of understanding how `std::forward` determines
23 whether to cast to an rvalue, however, this suffices.

24 Given that both `std::move` and `std::forward` boil down to casts, the only differ-
25 ence being that `std::move` always casts, while `std::forward` only sometimes
26 does, you may wonder whether we can dispense with `std::move` and just use
27 `std::forward` everywhere. From a purely technical perspective, the answer is
28 yes: `std::forward` can do it all. `std::move` isn't necessary. (Strictly speaking,
29 neither function is *necessary*, because we could write our own casts everywhere,
30 but I hope we agree that that would be, well, yucky.)

31 `std::move`'s advantage is convenience. Consider a conventional move constructor
32 for a class with a single data member of type `std::string`:

```
33 class Widget {
34 public:
```

```
1   Widget(Widget&& rhs)           // conventional
2     : s(std::move(rhs.s)) {}      // move constructor
3   ...
4 
5   private:
6     std::string s;
7   };
8 
```

7 If we wanted to implement the same behavior using `std::forward` instead of
8 `std::move`, we'd have to write it this way:

```
9 class Widget {
10 public:
11   Widget(Widget&& rhs)           // move ctor using
12     : s(std::forward<std::string>(rhs.s)) {} // std::forward
13   ...
14 };
15 
```

15 Note first that `std::move` requires only a function argument (i.e., `rhs.s`), while
16 `std::forward` requires both a function argument (`rhs.s`) and a template type
17 argument (`std::string`). Note second that the type we pass to `std::forward`
18 should be a non-reference, because that's the standard mechanism for encoding
19 that the argument being passed is an rvalue (see Item 30). Together, this means
20 that `std::move` requires less typing than `std::forward`, and it spares us the
21 trouble of passing a type argument that encodes that the argument we're passing
22 is an rvalue.

23 More importantly, the use of `std::move` conveys the use of an unconditional cast
24 to an rvalue, while the use of `std::forward` indicates a cast to an rvalue only for
25 references to which rvalues have been bound. Those are two very different ac-
26 tions. The first one typically sets up a move, while the second one just passes—
27 *forwards*—an object to another function in a way that retains its original lvalue-
28 ness or rvalueness. Because these actions are so different, it's good that we have
29 two different functions (and function names) to distinguish them.

30 Things to Remember

- 31 ♦ `std::move` performs an unconditional cast to an rvalue. In and of itself, it
32 doesn't move anything.

- 1 ♦ std::forward casts its argument to an rvalue only if that argument is bound
- 2 to an rvalue.
- 3 ♦ Neither std::move nor std::forward do anything at runtime.

4 **Item 26: Distinguish universal references from rvalue references.**

- 6 It's been said that the truth shall set you free, but under the right circumstances, a
- 7 well-chosen lie can be equally liberating. This entire Item is such a lie. Because
- 8 we're dealing with software, however, let's eschew the word "lie" and instead say
- 9 that this Item comprises an *abstraction*.

- 10 To declare an rvalue reference to some type T, you write T&&. It thus seems reasonable to assume that if you see "T&&" in source code, you're looking at an rvalue reference. Alas, it's not quite that simple:

```
13 void f(Widget&& param);           // rvalue reference
14 Widget&& var1 = Widget();          // rvalue reference
15 auto&& var2 = var1;               // not rvalue reference
16 template<typename T>
17 void f(std::vector<T>&& param);   // rvalue reference
18 template<typename T>
19 void f(T&& param);              // not rvalue reference
```

- 20 In fact, "T&&" has two different meanings. One is rvalue reference, of course. Such references behave exactly the way you expect: they bind only to rvalues, and their primary *raison d'être* is to identify objects that may be moved from.

- 23 The other possible meaning for "T&&" is *either* rvalue reference *or* lvalue reference.
- 24 Such references look like rvalue references in the source code (i.e., "T&&"), but they
- 25 can behave as if they were lvalue references (i.e., "T&"). Their dual nature permits
- 26 them to bind to rvalues (like rvalue references) as well as lvalues (like lvalue references). Furthermore, they can bind to const or non-const objects, to volatile
- 27 or non-volatile objects, even to objects that are both const and volatile. They
- 28 can bind to virtually *anything*. Such unprecedentedly flexible references deserve a
- 29 name of their own. I call them *universal references*.

1 Universal references arise in two contexts. The most common is function template
2 parameters, such as this example from the sample code above:

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

5 The second context is `auto` declarations, including this one from the sample code
6 above:

```
7 auto&& var2 = var1;           // var2 is a universal reference
```

8 What these contexts have in common is the presence of *type deduction*. In the tem-
9 plate `f`, the type of `param` is being deduced, and in the declaration for `var2`, `var2`'s
10 type is being deduced. Compare that with the following examples (also from the
11 sample code above), where type deduction is missing. If you see “`T&&`” without
12 type deduction, you’re looking at an rvalue reference:

```
13 void f(Widget&& param);      // no type deduction;
14                               // param is an rvalue reference
```

```
15 Widget&& var1 = Widget();    // no type deduction;
16                               // var1 is an rvalue reference
```

17 Because universal references are references, they must be initialized. The initializ-
18 er for a universal reference determines whether it represents an rvalue reference
19 or an lvalue reference. If the initializer is an rvalue, the universal reference corre-
20 sponds to an rvalue reference. If the initializer is an lvalue, the universal reference
21 corresponds to an lvalue reference. For universal references that are function pa-
22 rameters, the initializer is provided at the call site:

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

```
25 Widget w;
26 f(w);                     // lvalue passed to f; param's type is
27                               // Widget& (i.e., an lvalue reference)
```

```
28 f(std::move(w));          // rvalue passed to f; param's type is
29                               // Widget&& (i.e., an rvalue reference)
```

30 For a reference to be universal, type deduction is necessary, but it’s not sufficient.
31 The *form* of the reference declaration must also be correct, and that form is quite

1 constrained. It must be precisely “`T&&`”. Look again at this example from the sam-
2 ple code we saw earlier:

```
3 template<typename T>
4 void f(std::vector<T>&& param); // param is an rvalue reference
```

5 When `f` is invoked, the type `T` will be deduced (unless the caller explicitly specifies
6 it, an edge case we’ll not concern ourselves with). But the form of `param`’s type
7 declaration isn’t “`T&&`,” it’s “`std::vector<T>&&`.” That rules out the possibility
8 that `param` is a universal reference. `param` is therefore an rvalue reference, some-
9 thing that your compilers will be happy to confirm for you if you try to pass an
10 lvalue to `f`:

```
11 std::vector<int> v;
12 f(v); // error! can't bind lvalue to
13 // rvalue reference
```

14 Even the simple presence of a `const` qualifier is enough to disqualify a reference
15 from being universal:

```
16 template<typename T>
17 void f(const T&& param); // param is an rvalue reference
```

18 If you’re in a template and you see a function parameter of type “`T&&`,” you might
19 think you can assume that it’s a universal reference. You can’t. That’s because be-
20 ing in a template doesn’t guarantee the presence of type deduction. Consider this
21 `push_back` member function in `std::vector`:

```
22 template<class T, class Allocator = allocator<T>> // from C++
23 class vector { // Standard
24 public:
25     void push_back(T&& x);
26     ...
27 };
```

28 `push_back`’s parameter certainly has the right form for a universal reference, but
29 there’s no type deduction in this case. That’s because `push_back` can’t exist with-
30 out a particular `vector` instantiation for it to be part of, and the type of that in-
31 stantiation fully determines the declaration for `push_back`. That is, saying

```
32 std::vector<Widget> v;
```

33 causes

```
1 class vector<Widget, allocator<Widget>> {
2 public:
3     void push_back(Widget&& x); // rvalue reference
4     ...
5 }
```

6 to be generated, and now you can see clearly that `push_back` employs no type de-
7 duction. This `push_back` for `vector<T>` (there are two—the function is overload-
8 ed) always declares a parameter of type rvalue-reference-to-T.

9 In contrast, the conceptually similar `emplace_back` member function in
10 `std::vector` does employ type deduction:

```
11 template<class T, class Allocator = allocator<T>> // still from
12 class vector { // C++
13 public: // Standard
14     template <class... Args>
15     void emplace_back(Args&&... args);
16     ...
17 }
```

18 Here, the type parameter `Args` is independent of `vector`'s type parameter `T`, so
19 `Args` must be deduced each time `emplace_back` is called. (Okay, `Args` is really a
20 parameter pack, not a type parameter, but for purposes of this discussion, we can
21 treat it as if it were a type parameter.)

22 The fact that `emplace_back`'s type parameter is named `Args`, yet it's still a uni-
23 versal reference, reinforces my earlier comment that it's the *form* of a universal
24 reference that must be “`T&&`”. There's no requirement that you use the name `T`. For
25 example, the following template takes a universal reference, because the form
26 (“`type&&`”) is right, and `param`'s type will be deduced (again, excluding the corner
27 case where the caller explicitly specifies the type):

```
28 template<typename MyTemplateType> // param is a
29 void someFunc(MyTemplateType&& param); // universal reference
```

30 I remarked earlier that `auto` variables can also be universal references. To be
31 more precise, variables declared with the type `auto&&` are universal references,
32 because type deduction takes place and they have the correct form (“`T&&`”). `auto`
33 universal references are not as common as universal references used for function
34 template parameters, but they do crop up from time to time in C++11. They crop

1 up a lot more in C++14, because C++14 lambda expressions may declare `auto&&`
2 parameters. For example, if you wanted to write a C++14 lambda to record the
3 time taken in an arbitrary function invocation, you could do this:

```
4 auto timeFuncInvocation = [](auto&& func, auto&&... args)
5 {
6     start timer;
7     std::forward<decltype(func)>(func)(          // invoke func on
8         std::forward<decltype(args)>(args)...      // args in C++14
9     );
10    stop timer and record elapsed time;
11};
```

12 If your reaction to the “`std::forward<decltype(bLah bLah bLah)>`” code in-
13 side the lambda is, “What the...?!”, that probably just means you haven’t yet read
14 Item 35. Don’t worry about it. The important thing in this Item is the `auto&&` pa-
15 rameters that the lambda declares. `func` is a universal reference that can be bound
16 to any callable entity, lvalue or rvalue. `args` is one or more universal references
17 (i.e., a universal reference parameter pack) that can be bound to any number of
18 objects of arbitrary types. The result, thanks to `auto` universal references, is that
19 `timeFuncInvocation` can time pretty much any function execution. (For infor-
20 mation on the difference between “any” and “pretty much any,” turn to Item 32.)

21 Bear in mind that this entire Item—the foundation of universal references—is a
22 lie...er, an abstraction. The underlying truth is known as *reference-collapsing*, a
23 topic to which Item 30 is dedicated. But the truth doesn’t make the abstraction any
24 less useful. Distinguishing between rvalue references and universal references will
25 help you read source code more accurately (“Does that “`T&&`” I’m looking at bind to
26 rvalues only or to everything?”), and it will avoid ambiguities when you communi-
27 cate with your colleagues (“I’m using a universal reference here, not an rvalue ref-
28 erence...”). It will also allow you to make sense of Items 27 and 28, which rely on
29 the distinction. So embrace the abstraction. Revel in it. Just as Newton’s laws of
30 motion (which are technically incorrect) are typically just as useful as and easier to
31 apply than Einstein’s theory of general relativity (“the truth”), so is the notion of
32 universal references normally preferable to working through the details of refer-
33 ence-collapsing.

1 **Things to Remember**

- 2 ♦ If a variable or parameter has type `T&&` for some deduced type `T`, it's a universal reference.
- 3
- 4 ♦ If the form of the type isn't precisely `T&&`, or if type deduction does not occur, `T&&` denotes an rvalue reference.
- 5
- 6 ♦ Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues.
- 7

8 **Item 27: Use `std::move` on rvalue references, `std::forward` on universal references.**

10 Rvalue references bind only to objects that are candidates for moving. If you have
11 an rvalue reference parameter, you *know* that the object it's bound to may be
12 moved:

```
13 class Widget {  
14     Widget(Widget&& rhs);           // rhs definitely refers to an  
15     ...                                // object eligible for moving  
16 };
```

17 That being the case, you'll want to pass such objects to other functions in a way
18 that permits those functions to take advantage of the object's rvalue-ness. The way
19 to do that is to cast parameters bound to such objects to rvalues. As Item 25 explains,
20 that's not only what `std::move` does, it's what it was created for:

```
21 class Widget {  
22 public:  
23     Widget(Widget&& rhs)           // rhs is rvalue reference  
24     : name(std::move(rhs.name)),  
25         p(std::move(rhs.p))  
26     { ... }  
27     ...  
  
28 private:  
29     std::string name;  
30     std::shared_ptr<SomeDataStructure> p;  
31 };
```

32 A universal reference, on the other hand (see Item 26), *might* be bound to an object
33 that's eligible for moving. Universal references should be cast to rvalues only if

1 they were initialized with rvalues. Item 25 explains that this is precisely what
2 `std::forward` does:

```
3 class Widget {  
4 public:  
5     template<typename T>  
6     void setName(T&& newName)           // newName is  
7     { name = std::forward<T>(newName); }   // universal reference  
8     ...  
9 };
```

10 In short, rvalue references should be *unconditionally cast* to rvalues (via
11 `std::move`) when forwarding them to other functions, because they're *always*
12 bound to rvalues, and universal references should be *conditionally cast* to rvalues
13 (via `std::forward`) when forwarding them, because they're only *sometimes*
14 bound to rvalues.

15 Item 25 explains that using `std::forward` on rvalue references can be made to
16 exhibit the proper behavior, but the source code is wordy and unidiomatic, so you
17 should avoid using `std::forward` with rvalue references. Even worse is the idea
18 of using `std::move` with universal references, because that can have the effect of
19 unexpectedly modifying lvalues (e.g., local variables):

```
20 class Widget {  
21 public:  
22     template<typename T>  
23     void setName(T&& newName)           // universal reference  
24     { name = std::move(newName); }         // compiles, but is  
25     ...                                     // bad, bad, bad!  
26 private:  
27     std::string name;  
28     std::shared_ptr<SomeDataStructure> p;  
29 };  
30 std::string getWidgetName();           // factory function  
31 Widget w;  
32 auto n = getWidgetName();             // n is local variable  
33 w.setName(n);                      // moves n into w!  
34 ...                                // n's value now unknown
```

1 Here, the local variable `n` is passed to `w.setName`, which the caller can be forgiven
2 for assuming is a read-only operation. But because `setName` internally uses
3 `std::move` to unconditionally cast its reference parameter to an rvalue, `n`'s value
4 will be moved into `w.name`, and `n` will come back from the call to `setName` with an
5 unspecified value. That's the kind of behavior that can move callers to despair—
6 possibly to violence.

7 You might argue that `setName` shouldn't have declared its parameter to be a uni-
8 versal reference. Such references can't be `const` (see Item 26), yet `setName` sure-
9 ly shouldn't modify its parameter. You might point out that if `setName` had simply
10 been overloaded for `const` lvalues and for rvalues, the whole problem could have
11 been avoided. Like this:

```
12 class Widget {  
13 public:  
14     void setName(const std::string& newName)           // set from  
15     { name = newName; }                                // const lvalue  
16     void setName(std::string&& newName)           // set from  
17     { name = std::move(newName); }                  // rvalue  
18     ...  
19 };
```

20 That would certainly work in this case, but there are drawbacks. First, it's more
21 source code to write and maintain (two functions instead of a single template).
22 Second, it can be less efficient. For example, consider this use of `setName`:

```
23 w.setName("Adela Novak");
```

24 With the version of `setName` taking a universal reference, the string literal "Adela
25 Novak" would be passed to `setName`, where it would be conveyed to the construc-
26 tor for the `std::string` inside `w`. `w`'s `name` data member would thus be construct-
27 ed directly from the string literal; no temporary `std::string` objects would arise.
28 With the overloaded versions of `setName`, however, a temporary `std::string`
29 object would be created for `setName`'s parameter to bind to, and this temporary
30 `std::string` would then be moved into `w`'s data member. A call to `setName`
31 would thus entail execution of one `std::string` constructor (to create the tem-
32 porary), one `std::string` move constructor (to move `newName` into `w.name`), and

1 one `std::string` destructor (to destroy the temporary). That's almost certainly a
2 more expensive execution sequence than invoking only the `std::string` con-
3 structor taking a `const char*` pointer. The additional cost is likely to vary from
4 implementation to implementation, and whether that cost is worth worrying
5 about will vary from application to application and library to library, but the fact is
6 that replacing a template taking a universal reference with a pair of functions
7 overloaded on lvalue references and rvalue references is likely to incur a runtime
8 cost in some cases. If we generalize the example such that `Widget`'s data member
9 may be of an arbitrary type (rather than knowing that it's `std::string`), the per-
10 formance gap can widen considerably, because not all types are as cheap to move
11 as `std::string` (see Item 31).

12 The most serious problem with overloading on lvalues and rvalues, however, isn't
13 the volume or idiomticity of the source code, nor is it the code's runtime perfor-
14 mance. It's its lack of scalability. `Widget::setName` takes only one parameter, so
15 only two overloads are necessary in this example, but for functions taking more
16 parameters, each of which could be an lvalue or an rvalue, the number of over-
17 loads grows geometrically: n parameters necessitates 2^n overloads. And that's not
18 the worst of it. Some functions—function templates, actually—take an *unlimited*
19 number of parameters, each of which could be an lvalue or rvalue. The poster chil-
20 dren for such functions are `std::make_shared`, and, as of C++14,
21 `std::make_unique` (see Item 23). Check out the declarations of their most com-
22 monly-used overloads:

```
23 template<class T, class... Args> // from C++11
24     shared_ptr<T> make_shared(Args&&... args); // Standard
25 template<class T, class... Args> // from C++14
26     unique_ptr<T> make_unique(Args&&... args); // Standard
```

27 For functions like this, overloading on lvalues and rvalues is not an option: univer-
28 sal references are the only way to go. And inside such functions, I assure you,
29 `std::forward` is applied to the universal reference parameters when they're
30 passed to other functions. Which is exactly what you should do.

31 Well, usually. Eventually. But not necessarily initially. In some cases, you'll want to
32 use the object bound to an rvalue reference or a universal references more than

1 once in a single function, and you'll want to make sure that it's not moved from
2 until you're otherwise done with it. In that case, you'll want to apply `std::move`
3 (for rvalue references) or `std::forward` (for universal references) to only the
4 *final* use of the reference. For example:

```
5 template<typename T> // text is
6 void setSignText(T&& text) // univ. reference
7 {
8     sign.setText(text); // use text, but
9                     // don't modify it
10    auto now = // get current time
11        std::chrono::system_clock::now();
12
13    signHistory.add(now,
14                     std::forward<T>(text)); // conditionally cast
15                     // text to rvalue
```

15 Here, we want to make sure that `text`'s value doesn't get changed by
16 `sign.setText`, because we want to use that value when we call `signHisto-`
17 `ry.add`. Ergo the use of `std::forward` on only the final use of the universal ref-
18 erence.

19 For `std::move`, the same thinking applies (i.e., apply `std::move` to an rvalue ref-
20 erence the last time it's used), but it's important to note that in a few cases, you'll
21 want to call `std::move_if_noexcept` instead of `std::move`. To learn when and
22 why, consult Item 16.

23 If you're in a function that returns *by value*, and you're returning an object bound
24 to an rvalue reference or a universal reference, you'll want to apply `std::move` or
25 `std::forward` when you return the reference. To see why, consider an opera-
26 `tor+` function to add two matrices together, where the left-hand matrix is known
27 to be an rvalue (and can hence have its storage reused to hold the sum of the ma-
28 trices):

```
29 Matrix operator+(Matrix&& lhs, // by-value return
30                      const Matrix& rhs)
31 {
32     lhs += rhs;
33     return std::move(lhs); // move lhs into return value
34 }
```

1 By casting `lhs` to an rvalue in the `return` statement (via `std::move`), `lhs` will be
2 moved into the function's return value location. If the call to `std::move` were
3 omitted,

```
4 Matrix operator+(Matrix&& lhs,           // as above
5                           const Matrix& rhs)
6 {
7     lhs += rhs;
8     return lhs;                         // copy lhs into return value
9 }
```

10 the fact that `lhs` is an lvalue would force compilers to instead *copy* it into the re-
11 turn value location. Assuming that the `Matrix` type supports move construction
12 that is more efficient than copy construction, using `std::move` in the `return`
13 statement yields more efficient code.

14 If `Matrix` does not support moving, casting it to an rvalue won't hurt, because the
15 rvalue will simply be copied by `Matrix`'s copy constructor (see Item 25). If `Matrix`
16 is later revised to support moving, `operator+` will benefit from that change the
17 next time it (`operator+`) is compiled. That being the case, there's nothing to be
18 lost (and much to be gained) by applying `std::move` to rvalue references being
19 returned from functions that return by value.

20 The situation is similar for universal references and `std::forward`. Consider a
21 function (really a function template) `reduceAndCopy` that takes a possibly-
22 unreduced `Fraction` object, reduces it, and then returns a copy of the reduced
23 value. If the original object is an rvalue, its value should be moved into the return
24 value (thus avoiding the expense of making a copy), but if the original is an lvalue,
25 an actual copy must be created. Hence:

```
26 template<typename T>           // universal reference param,
27 Fraction reduceAndCopy(T&& frac) // by-value return
28 {
29     frac.reduce();
30     return std::forward<T>(frac); // move rvalue into return
31 }                                // value, copy lvalue
```

32 If the call to `std::forward` were omitted, `frac` would be unconditionally copied
33 into `reduceAndCopy`'s return value.

1 Some programmers take the information above and try to extend it to cover situations where it doesn't apply. "If applying `std::move` to an rvalue reference parameter being copied into a return value turns a copy construction into a move construction," they reason, "I can perform the same optimization on local variables that I'm returning." In other words, they figure that given a function returning a local variable by value, such as this,

```
7 Widget makeWidget()          // "Copying" version of makeWidget
8 {
9     Widget w;                // local variable
10    ...
11    return w;                // "copy" w into return value
12 }
```

13 they can "optimize" it by turning the "copy" into a move:

```
14 Widget makeWidget()          // Moving version of makeWidget
15 {
16     Widget w;
17     ...
18     return std::move(w);      // move w into return value
19 }
```

20 My liberal use of quotation marks should have tipped you off that this line of reasoning is flawed. But why is it flawed?

22 It's flawed, because the Standardization Committee is way ahead of such programmers when it comes to this kind of optimization. It was recognized long ago that the "copying" version of `makeWidget` can avoid the need to copy the local variable `w` by constructing it in the memory set aside for the function's return value. This is known as the *return value optimization* (RVO), and it's been expressly blessed by the C++ Standard for as long as there's been one.

28 Wording such a blessing is finicky business, because you want to permit such *copy elision* only in places where it won't affect the observable behavior of the software. Paraphrasing the legalistic (arguably toxic) prose of the Standard, this particular blessing says that compilers may elide the copying (or the moving) of a local variable (or a parameter that was passed by value) in a function returning by value if (1) the type of the local variable is the same as that returned by the function and

1 (2) the local variable is what's being returned. With that in mind, look again at the
2 "copying" version of `makeWidget`:

```
3 Widget makeWidget()           // "Copying" version of makeWidget
4 {
5     Widget w;
6     ...
7     return w;                 // "copy" w into return value
8 }
```

9 Both conditions are fulfilled here, and you can trust me when I tell you that every
10 decent C++ compiler will employ the RVO to avoid copying `w`. That means that the
11 "copying" version of `makeWidget` doesn't, in fact, copy anything—at least not in a
12 release build.

13 The moving version of `makeWidget` does just what its name says it does (assuming
14 `Widget` offers a move constructor): it moves the contents of `w` into `makeWidget`'s
15 return value location. But why don't compilers use the RVO to eliminate the move,
16 again constructing `w` in the memory set aside for the function's return value? The
17 answer is simple: they're not allowed to. Condition (2) stipulates that the RVO may
18 be performed only if what's being returned is a local variable, but that's not what
19 the moving version of `makeWidget` is doing. Look again at its `return` statement:

```
20 return std::move(w);
```

21 What's being returned here isn't `w`, it's *the result of calling a function*, namely,
22 `std::move`. Returning the result of a function call doesn't satisfy the conditions
23 required for the RVO to be valid, so compilers must move `w` into the function's re-
24 turn value location. Developers trying to help their compilers optimize by applying
25 `std::move` to a local variable that's being returned are actually limiting the opti-
26 mization options available to the compilers! There are situations where applying
27 `std::move` to a local variable can be a reasonable thing to do (i.e., when you're
28 passing it to a function and you know you won't be using the variable any more),
29 but as part of a `return` statement when the RVO would otherwise be available is
30 assuredly not one of them.

1 **Things to remember**

- 2 ♦ Apply `std::move` to rvalue references and `std::forward` to universal refer-
3 ences the last time each is used.
- 4 ♦ Do the same thing for rvalue references and universal references being re-
5 turned from functions that return by value.
- 6 ♦ Never apply `std::move` or `std::forward` to local objects (including by-value
7 parameters) if they would otherwise be eligible for the return value optimiza-
8 tion.

9 **Item 28: Avoid overloading on universal references.**

10 Suppose you need to write a function that takes a name as a parameter, logs the
11 current date and time, then adds the name to a global data structure. That is, you
12 need to write a function that looks something like this:

```
13 std::set<std::string> names;           // global data structure
14 void logAndAdd(const std::string& name)
15 {
16     auto now =                         // get current time
17         std::chrono::system_clock::now();
18     log(now, "logAndAdd");            // make log entry
19     names.emplace(name);             // add name to global
20 }                                     // structure; Item 18
21                                         // has info on emplace
```

22 This isn't unreasonable code, but it's not as efficient as it could be. Consider three
23 potential calls:

```
24 std::string petName("Darla");
25 logAndAdd(petName);                  // pass lvalue std::string
26 logAndAdd(std::string("Persephone")); // pass rvalue std::string
27 logAndAdd("Patty Dog");             // pass string literal
```

28 In the first call, `logAndAdd`'s parameter `name` is bound to the variable `petName`.
29 Within `logAndAdd`, `name` is ultimately passed to `names.emplace`. Because `name` is

1 an lvalue, it is copied into `names`. There's no way to avoid that copy, because an
2 lvalue (`petName`) was passed into `logAndAdd` in the first place.

3 In the second call, the parameter `name` is bound to an rvalue (the temporary
4 `std::string` explicitly created from "Persephone"). `name` itself is an lvalue, so
5 it's copied into `names`, but we recognize that, in principle, its value could be moved
6 into `names`. In this call, we pay for a copy, but we should be able to get by with only
7 a move.

8 In the third call, the parameter `name` is again bound to an rvalue, but this time it's
9 to a temporary `std::string` that's implicitly created from "Patty Dog". As in the
10 second call, `name` is copied into `names`, but in this case, the argument originally
11 passed to `logAndAdd` was a string literal. Had that string literal been passed di-
12 rectly to the call to `emplace`, there would have been no need to create a temporary
13 `std::string` at all. Instead, `emplace` would have used the string literal to create
14 the `std::string` object directly inside the `std::set`. In this third call, then,
15 we're paying to copy a `std::string`, yet there's really no reason to pay even for a
16 move, much less a copy.

17 We can eliminate the inefficiencies in the second and third calls by rewriting `lo-`
18 `gAndAdd` to take a universal reference (see Item 26) and, in accord with Item 27,
19 `std::forward`ing this reference to `emplace`. The results speak for themselves:

```
20 template<typename T>
21 void logAndAdd(T&& name)
22 {
23     auto now = std::chrono::system_clock::now();
24     log(now, "logAndAdd");
25     names.emplace(std::forward<T>(name));
26 }

27 std::string petName("Darla");           // as before
28 logAndAdd(name);                     // as before, copy
29                           // lvalue into set
30 logAndAdd(std::string("Persephone")); // move rvalue instead
31                           // of copying it
32 logAndAdd("Patty Dog");             // create std::string
33                           // in set instead of
```

```
1 // copying a temporary  
2 // std::string  
3 Hurray, optimal efficiency!  
4 Were this the end of the story, we could stop here and go home happy, but I ha-  
5 ven't told you that clients don't always have direct access to the names that lo-  
6 gAndAdd requires. Some clients have only an index that logAndAdd uses to look  
7 up the corresponding name in a table. To support such clients, logAndAdd is over-  
8 loaded:
```

```
9 std::string nameFromIdx(int idx);      // return name  
10                           // corresponding to idx  
11 void logAndAdd(int idx)  
12 {  
13     auto now = std::chrono::system_clock::now();  
14     log(now, "logAndAdd");  
15     names.emplace(nameFromIdx(idx));  
16 }
```

17 Resolution of calls to the two overloads works as expected:

```
18 std::string petName("Darla");          // as before  
19 logAndAdd(petName);                  // as before, these  
20 logAndAdd(std::string("Persephone")); // calls all invoke  
21 logAndAdd("Patty Dog");            // the T& overload  
22 logAndAdd(22);                     // calls int overload
```

23 Actually, resolution works as expected only if you don't expect too much. Suppose
24 a client has a `short` holding an index and passes that to `logAndAdd`:

```
25 short nameIdx;  
26 ...                      // give nameIdx a value  
27 logAndAdd(nameIdx);        // error!
```

28 The comment on the last line isn't terribly illuminating, so let me explain what
29 happens here.

30 There are two `logAndAdd` overloads. The one taking a universal reference can de-
31 duce `T` to be `short`, thus yielding an exact match. The overload with an `int` pa-
32 rameter can match the `short` argument only with a promotion. Per the normal

1 overload resolution rules, an exact match beats a match with a promotion, so the
2 universal reference overload is invoked.

3 Within that overload, the parameter `name` is bound to the `short` that's passed in.
4 `name` is then `std::forwarded` to the `emplace` member function on a
5 `std::set<std::string>`, which, in turn, dutifully forwards it to the
6 `std::string` constructor. There is no constructor for `std::string` that takes a
7 `short`, so the `std::string` constructor call inside the call to `set::emplace` in-
8 side the call to `logAndAdd` fails. All because the universal reference overload was a
9 better match for a `short` argument than an `int`.

10 Functions taking universal references are the greediest functions in C++. They in-
11 stantiate to create exact matches for almost any type of argument. (The few kinds
12 of arguments where this isn't the case are described in Item 32.) This is why com-
13 bining overloading and universal references is almost always a bad idea: the uni-
14 versal reference overload vacuums up far more argument types than the developer
15 doing the overloading generally expects.

16 An easy way to topple into this pit is to write a perfect forwarding constructor. A
17 small modification to the `logAndAdd` example demonstrates the problem. Instead
18 of writing a function that can take either a `std::string` or an index that can be
19 used to look up a `std::string`, imagine a class `Person` to which we can pass ei-
20 ther a `std::string` or an index:

```
21 class Person {  
22 public:  
23     template<typename T>  
24     explicit Person(T&& n)           // perfect forwarding ctor;  
25     : name(std::forward<T>(n)) {}    // initializes data member  
26     explicit Person(int idx)          // int ctor  
27     : name(nameFromIdx(idx)) {}  
28     ...  
29 private:  
30     std::string name;  
31 };
```

32 As was the case with `logAndAdd`, passing an integral type other than `int` (e.g.,
33 `std::size_t`, `short`, `long`, etc.) will call the universal reference constructor

1 overload instead of the `int` overload, and that will lead to compilation failures.
2 The problem is much worse in this case, however, because there's more overload-
3 ing present in `Person` that meets the eye. Item 19 explains that, under the appro-
4 priate conditions, C++ will generate both copy and move constructors, and this is
5 true even if the class contains a templated constructor that could be instantiated
6 to produce the signature of the copy or move constructor. If the copy and move
7 constructors for `Person` are thus generated, `Person` will effectively look like this:

```
8 class Person {  
9 public:  
10    template<typename T>           // perfect forwarding ctor  
11    explicit Person(T&& n);  
12  
13    explicit Person(int idx);      // int ctor  
14    Person(const Person& rhs);    // copy ctor (compiler-generated)  
15    Person(Person&& rhs);        // move ctor (compiler-generated)  
16    ...  
17 };
```

17 This leads to behavior that's intuitive only if you've spent so much time around
18 compilers and compiler-writers, you've forgotten what it's like to be human:

```
19 Person p("Bart");  
20 auto clone(p);           // create new Person from p;  
21                           // this won't compile!
```

22 Here we're trying to create a `Person` from another `Person`, which seems like
23 about as obvious a case for copy construction as one can get. (`p`'s an lvalue, so we
24 can banish any thoughts we might have about the "copying" being accomplished
25 through a move operation.) But this code won't call the copy constructor. It will
26 call the perfect-forwarding constructor. That function will then try to initialize
27 `Person`'s `std::string` data member with a `Person` object (`p`). `std::string`
28 having no constructor taking a `Person`, your compilers will throw up their hands
29 in exasperation, possibly punishing you with a long and incomprehensible error
30 message as an expression of their displeasure.

31 "Why," you might wonder, "does the perfect-forwarding constructor get called in-
32 stead of the copy constructor? We're initializing a `Person` with another `Person`!"

1 Indeed we are, but compilers are sworn to uphold the rules of C++, and the rules of
2 relevance here are the ones governing the resolution of calls to overloaded func-
3 tions.

4 Compilers reason as follows. `clone` is being initialized with a non-`const` lvalue
5 (`p`), and that means that the templated constructor can be instantiated to take a
6 non-`const` lvalue of type `Person`. After such instantiation, the `Person` class looks
7 like this:

```
8  class Person {  
9    public:  
10      explicit Person(Person& n); // instantiated from perfect-  
11                                // forwarding template  
12      explicit Person(int idx); // as before  
13      Person(const Person& rhs); // copy ctor (compiler-generated)  
14      ...  
15  };
```

16 In the statement,

```
17  auto clone(p);  
18 p could be passed to either the copy constructor or the instantiated template. Call-  
19 ing the copy constructor would require adding const to p to match the copy con-  
20 structor's parameter's type, but calling the instantiated template requires no such  
21 addition. The overload generated from the template is thus a better match, so  
22 compilers do what they're designed to do: generate a call to the better-matching  
23 function. "Copying" non-const lvalues of type Person is thus handled by the per-  
24 fect-forwarding constructor, not the copy constructor.
```

25 If we change the example slightly so that the object to be copied is `const`, we hear
26 an entirely different tune:

```
27  const Person cp("Bart"); // object is now const  
28  auto clone(cp); // calls copy constructor!
```

1 Because the object to be copied is now `const`, it's an exact match for the parameter taken by the copy constructor. The templated constructor can be instantiated
2 to have the same signature,

```
4 class Person {  
5 public:  
6     explicit Person(const Person& n); // instantiated from  
7                                     // template  
8     Person(const Person& rhs);      // copy ctor  
9                                     // (compiler-generated)  
10    ...  
11};
```

12 but this doesn't matter, because one of the overloading-resolution rules in C++ is
13 that in situations where a template instantiation and a non-template function (i.e.,
14 a "normal" function) are equally good matches for a function call, the normal func-
15 tion is preferred. The copy constructor (a normal function) thereby trumps an in-
16 stantiated template with the same signature.

17 (If you're wondering why compilers generate a copy constructor when they could
18 instantiate a templated constructor to get the signature that the copy constructor
19 would have, review Item 19.)

20 The interaction among perfect-forwarding constructors and compiler-generated
21 copy and move operations develops even more wrinkles when inheritance enters
22 the picture. In particular, the conventional implementations of derived class copy
23 and move operations behave quite surprisingly. Here, take a look:

```
24 class SpecialPerson: public Person {  
25 public:  
26     SpecialPerson(const SpecialPerson& rhs) // copy ctor; calls  
27         : Person(rhs)                      // base class  
28         { ... }                          // forwarding ctor!  
29     SpecialPerson(SpecialPerson&& rhs) // move ctor; calls  
30         : Person(std::move(rhs))        // base class  
31         { ... }                          // forwarding ctor!  
32};
```

33 As the comments indicate, the derived class copy and move constructors don't call
34 their base class's copy and move constructors, they call the base class's perfect-
35 forwarding constructor! To understand why, note that the derived class functions

1 are using arguments of type `SpecialPerson` to pass to their base class (they actually
2 pass `const SpecialPersons`, but in this case, `const` is immaterial), then
3 work through the template instantiation and overloading-resolution consequences
4 for the constructors in class `Person`. Ultimately, the code won't compile, because
5 there's no `std::string` constructor taking a `SpecialPerson`.

6 I hope that by now I've convinced you that overloading on universal reference
7 parameters is something you should avoid if at all possible. But if overloading on
8 universal references is a bad idea, what do you do if you need a function that for-
9 wards most argument types, yet needs to treat some argument types in a special
10 fashion? That egg can be unscrambled in a number of ways. So many, in fact, that
11 I've devoted an entire Item to them. It's Item 29. The next Item. Keep reading,
12 you'll bump right into it.

13 **Things to Remember**

- 14 ♦ Overloading on universal references almost always leads to the universal ref-
15 erence overload being called more frequently than expected.
- 16 ♦ Perfect-forwarding constructors are especially problematic, because they're
17 typically better matches than copy constructors for non-const lvalues, and they
18 can hijack derived class calls to base class copy and move constructors.

19 **Item 29: Familiarize yourself with alternatives to overload- 20 ing on universal references.**

21 Item 28 explains that overloading on universal references can lead to a variety of
22 problems, both for freestanding and for member functions (especially construc-
23 tors). Yet it also gives examples where such overloading could be useful. If only it
24 would behave the way we'd like! This Item explores ways to achieve the desired
25 behavior, either through designs that avoid overloading on universal references or
26 by employing them in ways that constrain the types of arguments they can match.

27 The discussion that follows builds on the examples introduced in Item 28. If you
28 haven't read that Item recently, you'll want to review it before continuing.

1 **Abandoning overloading**

2 The first example in Item 28, `logAndAdd`, is representative of the many functions
3 that can avoid the drawbacks of overloading on universal references by simply us-
4 ing different names for the would-be overloads. The two `logAndAdd` overloads, for
5 example, could be broken into `logAndAddName` and `logAndAddNameIdx`. Alas, this
6 approach won't work for the second example we considered, the `Person` construc-
7 tor, because constructor names are fixed by the language. Besides, who wants to
8 give up overloading?

9 **Passing by `const T&`**

10 An alternative is to revert to C++98 and replace pass-by-universal-reference with
11 pass-by-(lvalue)-reference-to-`const`. In fact, that's the first approach Item 28 con-
12 siders (on page 213). The drawback is that the design isn't as efficient as we'd pre-
13 fer. Knowing what we now know about the interaction of universal references and
14 overloading, giving up some efficiency to keep things simple might be a more at-
15 tractive trade-off than it initially appeared.

16 **Passing by value**

17 An approach that often allows you to dial up performance without any increase in
18 complexity is to replace pass-by-reference parameters with, counterintuitively,
19 pass-by-value. "Pass by reference" here refers to *any kind* of reference. This strate-
20 gy can supplant both C++98's pass-by-lvalue-reference-to-`const` and C++11's
21 pass-by-universal-reference. The design adheres to the advice in Item 17 to pass
22 objects by value when you know you'll copy them, so I'll defer to that Item for a
23 detailed discussion of how things work and how efficient they are. Here, I'll just
24 show how the technique could be used in the `Person` example:

```
25   class Person {  
26   public:  
27       Person(std::string n)           // replaces T& ctor; see  
28       : name(std::move(n)) {}        // Item 17 for use of std::move  
29       Person(int idx)              // as before  
30       : name(nameFromIdx(idx)) {}  
31       ...
```

```
1 private:  
2     std::string name;  
3 };  
  
4 Because there's no std::string constructor taking only an integer, all int and  
5 int-like arguments to a Person constructor (e.g., std::size_t, short, long,  
6 etc.) get funneled to the int overload. Similarly, all arguments of type  
7 std::string (and things from which std::strings can be created, e.g., literals  
8 such as "Ruth") get passed to the constructor taking a std::string. There are  
9 thus no surprises for callers. You could argue, I suppose, that some people might  
10 be surprised that using 0 or NULL to indicate a null pointer would invoke the int  
11 overload, but such people should be referred to Item 8 and required to read it re-  
12 peatedly until the thought of using 0 or NULL as a null pointer makes them recoil.
```

13 Tag dispatch

14 Neither pass by lvalue-reference-to-const nor pass by value offer support for per-
15 fect forwarding. If your motivation for the use of a universal reference is perfect
16 forwarding, you have to use a universal reference; there's no other choice. Yet we
17 don't want to abandon overloading. So if we don't give up overloading and we
18 don't give up universal references, how can we avoid overloading on universal ref-
19 erences?

20 It's actually not that hard. Calls to overloaded functions are resolved by looking at
21 all the parameters of all the overloads as well as all the arguments at the call site,
22 then choosing the function with the best overall match—taking into account all
23 parameter/argument combinations. A universal reference parameter generally
24 provides an exact match for whatever's passed in, but if the universal reference is
25 part of a parameter list containing other parameters that are *not* universal refer-
26 ences, sufficiently poor matches on the non-universal reference parameters can
27 knock an overload with a universal reference out of the running. That's the basis
28 behind the *tag dispatch* approach, and an example will make the foregoing descrip-
29 tion easier to understand.

30 We'll apply tag dispatch to the `logAndAdd` example from page 214. Here's that
31 code, to save you the trouble of looking it up:

```

1 std::set<std::string> names; // global data structure
2 template<typename T>
3 void logAndAdd(T&& name) // make log entry, add
4 { // name to data structure
5     auto now = std::chrono::system_clock::now();
6     log(now, "logAndAdd");
7     names.emplace(std::forward<T>(name));
8 }

```

9 By itself, this function works fine, but were we to introduce the overload taking an
10 `int` that's used to look up objects by index, we'd be back in the troubled land of
11 Item 28. The goal of this Item is to avoid that. Rather than adding the overload,
12 we'll reimplement `logAndAdd` to delegate to two other functions, one for integral
13 values and one for everything else. `logAndAdd` itself will accept all argument
14 types, both integral and non-integral.

15 The two functions doing the real work will be named `logAndAddImpl`, i.e., we'll
16 use overloading. One of the functions will take a universal reference. So we'll have
17 both overloading and universal references. But each function will also take a sec-
18 ond parameter, one that indicates whether the argument being passed is integral.
19 This second parameter is what will prevent us from tumbling into the morass de-
20 scribed in Item 28, because we'll arrange it so that the second parameter will be
21 the factor that determines which overload is selected.

22 Yes, I know, "Blah, blah, blah. Stop talking and show me the code!" No problem.
23 Here's an almost- correct version of the updated `logAndAdd`:

```

24 template<typename T>
25 void logAndAdd(T&& name)
26 {
27     logAndAddImpl(std::forward<T>(name),
28                     std::is_integral<T>()); // not quite correct
29 }

```

30 This function forwards its parameter to `logAndAddImpl`, but it also passes an ar-
31 gument indicating whether the type it received (`T`) is integral. At least, that's what
32 it's supposed to do. For integral arguments that are rvalues, it's also what it does.
33 But, as Item 30 explains, if an lvalue argument is passed to the universal reference
34 `name`, the type deduced for `T` will be an lvalue reference. So if an lvalue of type `int`
35 is passed to `logAndAdd`, `T` will be deduced to be `int&`. `int&` is not an integral type,

1 because references aren't integral types. That means that `std::is_integral<T>`
2 will be false for any lvalue argument, even if the argument really does represent an
3 integral value.

4 Recognizing the problem is tantamount to identifying the solution, because the
5 ever-handy Standard C++ Library has a type trait, `std::remove_reference`, that
6 does both what its name suggests and what we need: remove any reference quali-
7 fiers from a type. The proper way to write `logAndAdd` is therefore:

```
8 template<typename T>
9 void logAndAdd(T&& name)
10 {
11     logAndAddImpl(
12         std::forward<T>(name),
13         std::is_integral<typename std::remove_reference<T>::type>()
14     );
15 }
```

16 This does the trick. (In C++14, you can save a few keystrokes by using
17 `std::remove_reference_t<T>` in place of the highlighted text. For details, see
18 Item 9.)

19 With that taken care of, we can shift our attention to the function being called, `lo-`
20 `gAndAddImpl`. There are two overloads, and the first is applicable only to non-
21 integral types (i.e., to types where `std::is_integral<typename`
22 `std::remove_reference<T>::type>` is false):

```
23 template<typename T>                                // non-integral
24 void logAndAddImpl(T&& name, std::false_type)      // argument:
25 {                                                       // add it to
26     auto now = std::chrono::system_clock::now();       // global data
27     log(now, "logAndAdd");                            // structure
28     names.emplace_back(std::forward<T>(name));
29 }
```

30 This is straightforward code, once you understand the mechanics behind the high-
31 lighted parameter. Conceptually, `logAndAdd` passes a boolean to `logAndAddImpl`
32 indicating whether an integral type was passed to `logAndAdd`, but `true` and
33 `false` are *runtime* values, and we need to use overload resolution—a *compile-time*
34 phenomenon—to choose the correct `logAndAddImpl` overload. That means we
35 need a *type* that corresponds to `true` and a different type that corresponds to

1 `false`. This need is common enough that the Standard Library provides what is
2 required under the names `std::true_type` and `std::false_type`. The argument passed to `logAndAddImpl` by `logAndAdd` is an object of a type that inherits
3 from `std::true_type` if `T` is integral and from `std::false_type` if `T` is not integral.
4 The net result is that this `logAndAddImpl` overload is a viable candidate for
5 the call in `logAndAdd` only if `T` is not an integral type.

7 The second overload covers the opposite case: when `T` is an integral type. In that
8 event, `logAndAddImpl` simply finds the name corresponding to the passed-in index
9 and passes that name back to `logAndAdd`:

```
10  std::string nameFromIdx(int idx);           // as in Item 28
11  void logAndAddImpl(int idx, std::true_type) // integral
12  {                                         // argument: look
13      logAndAdd(anyFromIdx(idx));          // up name and
14  }                                         // call logAndAdd
15                                         // with it
```

16 By having `logAndAddImpl` for an index look up the corresponding name and pass
17 it to `logAndAdd` (from where it will be `std::forwarded` to the other `logAndAddImpl` overload), we avoid the need to put the logging code in both `logAndAddImpl` overloads.

20 In this design, the types `std::true_type` and `std::false_type` are “tags”
21 whose only purpose is to force overload resolution to go the way we want. Notice
22 that we don’t even name those parameters. They serve no purpose at runtime, and
23 in fact we hope that compilers will recognize that tag parameters are unused and
24 will optimize them out of the program’s execution image. (Some compilers do, at
25 least some of the time.) The call to the overloaded implementation functions inside
26 `logAndAdd` “dispatches” the work to the correct overload by causing the proper
27 tag object to be created. Hence the name for this design: *tag dispatch*. It’s a standard
28 building block of template metaprogramming, and the more you look at code
29 inside contemporary C++ libraries, the more often you’ll encounter it.

30 For our purposes, what’s important about tag dispatch is less how it works and
31 more how it permits us to combine universal references and overloading without
32 the problems described in Item 28. The dispatching function—`logAndAdd`—takes

1 an unconstrained universal reference parameter, but this function is not overload-
2 ed. The implementation functions—`logAndAddImpl`—are overloaded, and one
3 takes a universal reference parameter, but resolution of calls to these functions
4 depends not just on the universal reference parameter, but also on the tag param-
5 ter, and the tag values are designed so that no more than one overload will ever be
6 a viable match. As a result, it's the tag that determines which overload gets called.
7 The fact that the universal reference parameter will always generate an exact
8 match for its argument is immaterial.

9 **Tag dispatch constructors**

10 A keystone of tag dispatch is the existence of a single (unoverloaded) function as
11 the client API. This single function dispatches the work to be done to the imple-
12 mentation functions. Creating an unoverloaded dispatch function is usually easy,
13 but the second problem case Item 28 considers, that of the perfect-forwarding
14 constructor for the `Person` class (see page 216), is an exception. Compilers may
15 generate copy and move constructors themselves, so even if you write only one
16 constructor and use tag dispatch within it, some constructor calls may be handled
17 by compiler-generated functions that bypass the tag dispatch system.

18 Frankly, the real problem is not that the compiler-generated functions sometimes
19 bypass the tag dispatch design, it's that they don't *always* pass it by. We virtually
20 always want the copy constructor for a class to handle requests to copy lvalues of
21 that type, but, as Item 28 demonstrates, providing a constructor taking a universal
22 reference causes the universal reference constructor (instead of the copy construc-
23 tor) to be called when copying non-`const` lvalues.

24 What we need is a way to tell compilers, "Look, when I'm constructing an object
25 from another object of the same type, just do the normal thing: copy each member
26 of an lvalue object, and move each member of an rvalue object. Don't worry about
27 whether the initializing object is `const` or not, just do a normal copy or move."
28 Brace yourself for good news, because there is, in fact, a way to say that.

29 Before we see what it is, let's look at a tag-dispatching `Person` constructor. The
30 design is identical to that we saw for `logAndAdd`, but we'll apply a minor twist.
31 We'll take advantage of C++11's constructor delegation to have other constructors

1 act as implementation functions. Using constructor delegation isn't necessary in
2 this example, but it's a good habit to get into in contexts such as this, because it
3 also works with `const` and reference data members, which may be initialized only
4 during construction. The constructors we'll delegate to aren't designed to be called
5 directly, so we'll declare them `private`.

6 Here's the code, where I'm employing C++14's `std::remove_reference_t<T>`
7 instead of C++11's wordier "`typename std::remove_reference<T>::type`":

```
8 class Person {
9 public:
10    template<typename T>                      // dispatch to other ctors
11    explicit Person(T&& initVal)              // via ctor delegation
12    : Person(
13        initVal,
14        std::is_integral<std::remove_reference_t<T>>() // C++14
15    )                                         // only
16    {}
17 ...
18 private:
19    template<typename T>                      // private ctor for
20    Person(T&& n, std::true_type)            // integral args
21    : Person(nameFromIdx(std::forward<T>(n)))
22    {}
23    template<typename T>                      // private ctor for
24    Person(T&& n, std::false_type)           // non-integral args
25    : name(std::forward<T>(n))
26    {}
27    std::string name;                         // as in Item 28
28};
```

29 Our goal is to modify this class so that the universal reference constructor never
30 gets invoked if a `Person` is created from another `Person`.

31 Our approach is based on three observations. First, a class may contain variants of
32 the copy operations. In particular, classes may specify different copy constructors
33 for `const` and non-`const` objects. Second, `=default` can be used to get the "nor-
34 mal" implementation of any special member function (see Item 19), including all
35 copy operation variants. Third, as Item 28 explains, if a template can be instantiat-

1 ed with the same signature as a non-template function, calls to that function signa-
2 ture invoke the normal function, not the template instantiation.

3 So what we'll do is manually declare the copy and move operation signatures we
4 want to behave "normally," (i.e., to not use tag dispatch), and we'll implement them
5 using `=default`. The existence of these functions will prevent the universal refer-
6 ence constructor from being instantiated with their signatures. Hence:

```
7  class Person {  
8  public:  
9      template<typename T>                          // as before,  
10     explicit Person(T&& initVal);              // tag-dispatching ctor  
11     Person(const Person&) = default;            // copy-construct from  
12                                                        // const lvalue  
13     Person(Person&) = default;                    // copy-construct from  
14                                                        // non-const lvalue  
15     Person(Person&&) = default;                // move-construct  
16                                                        // from (non-const) rvalue  
17     ...  
18 };
```

19 You might wonder why there's only a single move constructor here, i.e., why
20 there's no "move constructor" taking a `const Person&&`. That's because the only
21 valid move constructor signature is the one above; there's no `const` variant.
22 Item 25 explains that it makes no sense to try to move from a `const` object, so C++
23 doesn't recognize a constructor taking a `const` rvalue reference as a special mem-
24 ber function.

25 Because this Item is already too long, I'll leave it to you to verify that when `Person`
26 is defined like this, creation of a `Person` from another `Person` will bypass the uni-
27 versal reference constructor and do the "normal" thing, regardless of whether the
28 source object is an lvalue or an rvalue and regardless of whether it's `const`.

29 Item 28 points out that a second problem with tag-dispatching constructors is that
30 they interact poorly with inheritance, and it gives this example:

```
31 class SpecialPerson: public Person {  
32 public:  
33     SpecialPerson(const SpecialPerson& rhs) // copy ctor; calls
```

```

1   : Person(rhs)           // base class
2   { ... }                 // forwarding ctor!
3   SpecialPerson(SpecialPerson&& rhs)    // move ctor; calls
4   : Person(std::move(rhs))      // base class
5   { ... }                 // forwarding ctor!
6 };

```

7 The good news is that if you can live with the compiler-generated copy and move
8 operations, this problem doesn't arise. Such operations will call their base class
9 counterparts, just as you'd hope.

10 If you need to implement these functions yourself, you have to make sure that you
11 pass base class arguments to your base class constructors. That's not what hap-
12 pens in the code above. There, both `SpecialPerson` constructors pass `rhs` as an
13 argument to the `Person` constructor, but `rhs`'s basic type is `SpecialPerson`, not
14 `Person`. That will cause the universal reference constructor in `Person` to instanti-
15 ate and "steal" what you'd like to be calls to the `Person` copy and move construc-
16 tors. The solution is to cast away the derived-ness of the argument you pass to the
17 base class by casting it to a base class reference:

```

18 class SpecialPerson: public Person {
19 public:
20   SpecialPerson(const SpecialPerson& rhs)           // now calls
21   : Person(static_cast<const Person&>(rhs))       // base copy
22   { ... }                                         // ctor
23   SpecialPerson(SpecialPerson&& rhs)             // now calls
24   : Person(static_cast<Person&&>(std::move(rhs))) // base move
25   { ... }                                         // ctor
26 };

```

27 Note how the copy constructor casts `rhs` to an lvalue reference, while the move
28 constructor casts it to an rvalue reference. Subtleties such as this are one of the
29 reasons you should really try to follow Item 19's advice to avoid writing your own
30 copy and move operations.

31 Trade-offs

32 The first three techniques considered in this Item—abandoning overloading, pass-
33 ing by `const T&`, and passing by value—specify a type for each parameter in the
34 function(s) to be called. Tag dispatch uses perfect forwarding, hence doesn't speci-

1 fy types for the parameters. This fundamental decision—to specify a type or not—
2 has important consequences.

3 As a rule, perfect forwarding is more efficient, because it avoids the creation of
4 temporary objects solely for the purpose of conforming to the type of a parameter
5 declaration. In the case of the `Person` constructor, perfect forwarding permits a
6 string literal such as "Nancy" to be forwarded to the constructor for the
7 `std::string` inside `Person`, whereas techniques not using perfect forwarding
8 must create a temporary `std::string` object from the string literal to satisfy the
9 parameter specification for the `Person` constructor.

10 But perfect forwarding has drawbacks. One is that some kinds of arguments can't
11 be perfect-forwarded, even though they can be passed to functions taking specific
12 types. Item 32 explores these perfect forwarding failure cases.

13 A second issue is the comprehensibility of error messages when clients pass invalid
14 arguments. Suppose, for example, a client creating a `Person` object passes a
15 string literal made up of `char16_ts` (a type introduced in C++11 to represent 16-bit
16 characters) instead of `chars` (which is what a `std::string` consists of):

```
17 Person p(u"Konrad Zuse");      // "Konrad Zuse" consists of
18                           // characters of type std::char16_t
```

19 With the first three approaches examined in this Item, compilers will see that the
20 available constructors take either `int` or `std::string`, and they'll produce a
21 more or less straightforward message explaining that there's no conversion from
22 `char16_t*` to `int` or `std::string`.

23 With an approach based on universal reference parameters, however, the
24 `char16_t*` pointer gets bound to the constructor's parameter without complaint.
25 From there it's forwarded to the constructor of `Person`'s `std::string` data
26 member, and it's only at that point that the mismatch between what the caller
27 passed in (essentially a `char16_t*` pointer) and what's required (any type acceptable
28 to the `std::string` constructor) is discovered. The resulting error message is likely to be,
29 er, impressive. With one of the compilers I use, it's 158 lines long. In this example, the universal reference is forwarded only once (from the
30 `Person` constructor to the `std::string` constructor), but the more complex the
31 constructor, the more likely it is that the universal reference will be forwarded multiple times.

1 system, the more likely that a universal reference is forwarded through several
2 layers of function calls before finally arriving at a site that knows what argument
3 type(s) are acceptable. The more times the universal reference is forwarded, the
4 more baffling the error message when something goes wrong. Many developers
5 find that this issue alone is grounds to reserve universal reference parameters for
6 interfaces where performance is a foremost concern.

7 **Things to Remember**

- 8 ♦ Alternatives to the combination of universal references and overloading in-
9 clude the use of distinct function names, passing parameters by `(lvalue)-`
10 reference-to-`const`, passing parameters by value, and using tag dispatch.
11 ♦ Universal reference parameters often have efficiency advantages, but they also
12 have usability disadvantages.

13 **Item 30: Understand reference collapsing.**

14 Item 25 remarks that when an argument is passed to a template function, the de-
15 duced type for the template parameter encodes whether that argument is an
16 lvalue or an rvalue. The Item fails to mention that this happens only when the ar-
17 gument is used to initialize a parameter that's a universal reference, but there's a
18 good reason for the omission: universal references aren't introduced until Item 26.
19 Together, these observations about universal references and lvalue/rvalue encod-
20 ing mean that for this template,

21 `template<typename T>`
22 `void func(T&& param);`

23 the deduced template parameter `T` will encode whether the argument passed to
24 `param` was an lvalue or an rvalue.

25 The encoding mechanism is simple. When an lvalue is passed as an argument, `T` is
26 deduced to be an lvalue reference. When an rvalue is passed, `T` is deduced to be a
27 non-reference. (Note the asymmetry: lvalues are encoded as lvalue references, but
28 rvalues are encoded as *non-references*.) Hence:

29 `Widget widgetFactory(); // function returning rvalue`

```
1 Widget w;                      // a variable (an lvalue)
2 func(w);                      // call func with lvalue; T deduced
3                                         // to be Widget&
4 func(widgetFactory());          // call func with rvalue; T deduced
5                                         // to be Widget
```

6 In both calls to `func`, a `Widget` is passed, yet because one `Widget` is an lvalue and
7 one is an rvalue, different types are deduced for the template parameter `T`. This, as
8 we shall soon see, is what determines whether universal references become rvalue
9 references or lvalue references, and it's also the underlying mechanism through
10 which `std::forward` does its work.

11 Before we can look more closely at `std::forward` and universal references, we
12 must observe that references to references are illegal in C++. Should you be so
13 bold as to try to declare one, your compilers will reprimand you:

```
14 int x;
15 ...
16 auto& & rx = x;    // error! can't declare reference to reference
17 But consider what happens when an lvalue is passed to a template function taking
18 a universal reference:
```

```
19 template<typename T>
20 void func(T&& param);    // as before
21 func(w);                  // invoke func with lvalue;
22                                         // T deduced as Widget&
```

23 If we take the type deduced for `T` (i.e., `Widget&`) and use it to instantiate the specific
24 function we're calling, we get this:

```
25 void func(Widget& && param);
```

26 A reference to a reference! And yet compilers issue no protest. We know from Item
27 26 that because the universal reference `param` is being initialized with an lvalue,
28 `param`'s type is supposed to be an lvalue reference, but how does the compiler get
29 from the result of taking the deduced type for `T` and substituting it into the tem-
30 plate to the following, which is the ultimate function signature?

```
31 void func(Widget& param);
```

1 The answer is *reference collapsing*. Yes, *you* are forbidden from declaring references to references, but *compilers* may do it in particular contexts, template instantiation being among them. When compilers generate references to references, reference collapsing dictates what happens next.

5 There are two kinds of references (lvalue and rvalue), so there are four possible
6 reference-reference combinations (lvalue to lvalue, lvalue to rvalue, rvalue to
7 lvalue, and rvalue to rvalue). If a reference to a reference arises in a context where
8 this is permitted (e.g., during template instantiation), the references *collapse* to a
9 single reference according to this simple rule:

- 10 • If either reference is an lvalue reference, the result is an lvalue reference. Otherwise the result is an rvalue reference.

12 In our example above, substitution of the deduced type `Widget&` into the template
13 `func` yields an rvalue reference to an lvalue reference, and the reference-
14 collapsing rule tells us that the result is an lvalue reference.

15 Reference collapsing is a key part of what makes `std::forward` work. Item 25
16 introduces the following conceptual implementation:

```
17 template<typename T>                      // conceptual impl.  
18 T&& forward(T&& param)                   // of std::forward  
19 {  
20     if (is_lvalue_reference<T>::value) { // in namespace std  
21         return param;                  // if T indicates lvalue  
22     } else {                         // return param as lvalue  
23         return move(param);          // else  
24     }  
25 }
```

26 A real `std::forward` implementation looks more like this:

```
27 template<typename T>                      // in namespace std  
28 T&& forward(T&& param)  
29 {  
30     return static_cast<T&&>(param);  
31 }
```

32 This isn't quite standards-conformant (I've omitted a few interface details), but it
33 makes clear that, as Item 25 claims, `std::forward` is simply a cast.

1 But how does this work? As explained in Item 27, `std::forward` is generally used
2 on universal reference parameters, so a typical use case could look like this:

```
3 template<typename T>
4 void f(T&& fParam)
5 {
6     ...                                // do some work
7     func(std::forward<T>(fParam));    // forward fParam to func
8 }
```

9 Because `fParam` is a universal reference, we know that the type parameter `T` will
10 encode whether the argument passed to `f` (i.e., the expression used to initialize
11 `fParam`) was an lvalue or an rvalue. `std::forward`'s job is to cast `fParam` (an
12 lvalue) to an rvalue if and only if `T` encodes that the argument passed to `f` was an
13 rvalue, i.e., if `T` is a non-reference type. Let's go through this step by step.

14 Suppose that the argument passed to `f` is an lvalue of type `Widget`. In that case, `T`
15 will be deduced as `Widget&`. The call to `std::forward` will instantiate as
16 `std::forward<Widget&>`. Plugging `Widget&` into the above implementation
17 yields

```
18 Widget& && forward(Widget& && param)
19 { return static_cast<Widget& &&>(param); }
```

20 which, after reference collapsing, becomes:

```
21 Widget& forward(Widget& param)
22 { return static_cast<Widget&>(param); }
```

23 So when an lvalue argument is passed to the function template `f`, `std::forward`
24 is instantiated to take and return an lvalue reference. Lvalue references are, by
25 definition, lvalues, so an lvalue argument passed to `f` will be forwarded to `func` as
26 an lvalue, just like it's supposed to be.

27 Now suppose that the argument passed to `f` is an rvalue of type `Widget`. In this
28 case, the deduced type for the `f`'s type parameter, `T`, will simply be `Widget`. The
29 call inside `f` to `std::forward` will thus be to `std::forward<Widget>`. Substitut-
30 ing `Widget` for `T` in `std::forward` gives this:

```
31 Widget&& forward(Widget&& fwdParam)
32 { return static_cast<Widget&&>(param); }
```

1 No references to references arise in this case, so no reference collapsing occurs.
2 Because rvalue references returned from functions are defined to be rvalues, this
3 means that `std::forward` will turn `f`'s parameter `fParm` (an lvalue) into an rvalue.
4 The end result is that an rvalue argument passed to `f` will be forwarded to
5 `func` as an rvalue, which is precisely what is supposed to happen.

6 Reference collapsing occurs in four contexts. The first and most common is tem-
7 plate instantiation. The second is type generation for `auto` variables. The details
8 are essentially the same as for templates, because type deduction for `auto` varia-
9 bles is essentially the same as type deduction for templates (see Item 2). Consider
10 again this example from earlier in the Item:

```
11 template<typename T>
12 void func(T&& param);
13 Widget widgetFactory();      // function returning rvalue
14 Widget w;                  // a variable (an lvalue)
15 func(w);                  // call func with lvalue; T deduced
16                         // to be Widget&
17 func(widgetFactory());     // call func with rvalue; T deduced
18                         // to be Widget
```

19 This example can be mimicked in `auto` form. The declaration

20 `auto&& w1 = w;`

21 initializes `w1` with an lvalue, thus deducing the type `Widget&` for `auto`. Plugging
22 `Widget&` in for `auto` in the declaration for `w1` yields this reference-to-reference
23 code,

24 `Widget& && w1 = w;`

25 which, after reference collapsing, becomes

26 `Widget& w1 = w;`

27 As a result, `w1` is an lvalue reference.

28 On the other hand, this declaration,

29 `auto&& w2 = widgetFactory();`

1 initializes w2 with an rvalue, causing the non-reference type `Widget` to be deduced
2 for `auto`. Substituting `Widget` for `auto` gives us this:

3 `Widget&& w2 = widgetFactory();`

4 There are no references to references here, so we're done: `w2` is an rvalue refer-
5 ence.

6 We're now in a position to truly understand the universal references introduced in
7 Item 26. A universal reference isn't a new kind of reference, it's simply an rvalue
8 reference in a reference collapsing context. The concept of universal references is
9 useful, because it frees you from having to recognize the existence of reference col-
10 lapsing contexts, to mentally deduce different types for lvalues and rvalues, and to
11 apply the reference collapsing rule after mentally substituting the deduced types
12 into the contexts in which they occur.

13 I said there were four such contexts, but we've discussed only two: template in-
14 stantiation and `auto` type generation. The third is the generation and use of
15 `typedefs` and alias declarations (see Item 9). If, during creation or evaluation of a
16 `typedef`, references to references arise, reference collapsing intervenes to elimi-
17 nate them. For example, suppose we have a `Widget` class template with an em-
18 bedded `typedef` for an rvalue reference type,

```
19 template<typename T>
20 class Widget {
21 public:
22     typedef T&& RvalueRefToT;
23     ...
24 };
```

25 and suppose we instantiate `Widget` with an lvalue reference type:

26 `Widget<int&> w;`

27 Substituting `int&` for `T` in the `Widget` template gives us the following `typedef`:

28 `typedef int& && RvalueRefToT;`

29 Reference collapsing reduces it to this,

30 `typedef int& RvalueRefToT;`

1 which makes clear that the name we chose for the `typedef` is perhaps not as de-
2 scriptive as we'd hoped: `RvalueRefToT` is a `typedef` for an *lvalue reference* when
3 `Widget` is instantiated with an lvalue reference type.

4 The final context in which reference collapsing takes place is uses of `decltype`. If,
5 during evaluation of an expression involving `decltype`, a reference to reference
6 arises, reference collapsing will kick in to eliminate it.

7 **Things to Remember**

- 8 ♦ Reference collapsing occurs in four contexts: template instantiation, `auto` type
9 generation, creation and use of `typedefs` and alias declarations, and
10 `decltype`.
- 11 ♦ When compilers generate a reference to a reference in a reference collapsing
12 context, the result becomes a single reference. If either of the original refer-
13 ences is an lvalue reference, the result is an lvalue reference. Otherwise it's an
14 rvalue reference.
- 15 ♦ Universal references are rvalue references in reference collapsing contexts.

16 **Item 31: Assume that move operations are not present, not
17 cheap, and not used.**

18 Move semantics is arguably *the* premier feature of C++11. “Moving containers is
19 now as cheap as copying pointers!,” you’re likely to hear, and “Copying temporary
20 objects is now so efficient, coding to avoid it is tantamount to premature optimiza-
21 tion!” Such sentiments are easy to understand. Move semantics is truly an im-
22 portant feature. It doesn’t just allow compilers to replace expensive copy opera-
23 tions with comparatively cheap moves, it actually *requires* that they do so (when
24 the proper conditions are fulfilled). Take your C++98 code base, recompile with a
25 C++11-conformant compiler and Standard Library, and—*shazam!*—your software
26 runs faster!

27 Move semantics can really pull that off, and that grants the feature an aura worthy
28 of legend. Legends, however, are generally the result of exaggeration. The purpose
29 of this Item is to keep your expectations grounded.

1 Let's begin with the observation that many types fail to support move semantics.
2 The entire C++98 Standard Library was overhauled for C++11 to add move opera-
3 tions for types where moving could be implemented faster than copying, and the
4 implementation of the library components was revised to take advantage of these
5 operations, but chances are that you're working with a code base that has not been
6 completely revised to take advantage of C++11. For types in your applications (or
7 in the libraries you use) where no modifications for C++11 have been made, the
8 existence of move support in your compilers is likely to do you little good. True,
9 C++11 is willing to generate move operations for classes that lack them, but that
10 happens only for classes declaring no copy operations, move operations, or de-
11 structors (see Item 19). Data members or base classes that can't be moved will also
12 suppress compiler-generated move operations. For types without explicit support
13 for moving and that don't qualify for compiler-generated move operations, there is
14 no reason to expect C++11 to deliver any kind of performance improvement over
15 C++98.

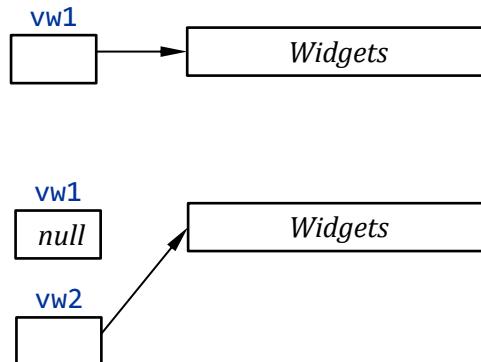
16 Even types with explicit move support may not benefit as much as you'd hope. All
17 containers in the standard C++11 library support moving, for example, but it
18 would be a mistake to assume that moving all containers is cheap. For some con-
19 tainers, this is because there's no truly cheap way to move their contents. For oth-
20 ers, it's because the truly cheap move operations the containers offer come with
21 caveats the container elements can't satisfy.

22 Consider `std::array`, a new container in C++11. `std::array` is essentially a
23 built-in array with an STL interface. This is fundamentally different from the other
24 standard containers, each of which store their contents on the heap. Objects of
25 such container types hold (as data members), conceptually, only a pointer to the
26 heap memory storing the contents of the container. (The reality is more complex,
27 but for purposes of this analysis, the differences are not important.) The existence
28 of this pointer makes it possible to move the contents of an entire container in
29 constant time: just move the pointer to the container's contents from the source
30 container to the target, and set the source's pointer to null:

```

std::vector<Widget> vw1;
// put data into vw1
...
// move vw1 into vw2. Runs in
// constant time. Only ptrs
// in vw1 and vw2 are modified
auto vw2 = std::move(vw1);
1

```

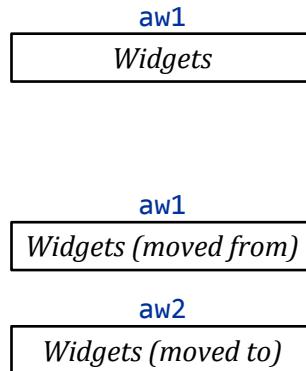


2 `std::array` objects lack such a pointer, because the data for a `std::array`'s contents are stored directly in the `std::array` object:

```

std::array<Widget, 10000> aw1;
// put data into aw1
...
// move aw1 into aw2. Runs in
// Linear time. All elements in
// aw1 are moved into aw2
auto aw2 = std::move(aw1);
4

```



5 Note that the elements in `aw1` are *moved* into `aw2`. Assuming that `Widget` is a type where moving is faster than copying, moving a `std::array` of `Widget` will be faster than copying the same `std::array`. So `std::array` certainly offers move support. Yet both moving and copying a `std::array` have linear-time computational complexity, because each element in the container must be copied or moved.

10 This is far from the “moving a container is now as cheap as assigning a couple of pointers” claim that one sometimes hears.

12 On the other hand, `std::string` offers constant-time moves and linear-time copies. That makes it sound like moving is faster than copying, but that may not be the case. Many string implementations employ the *small string optimization* (SSO). With the SSO, “small” strings (typically those with a capacity of no more than 15 characters) are stored in a buffer within the `std::string` object; no heap-

1 allocated storage is used. Moving small strings using an SSO-based implementation
2 is no faster than copying them, because the copy-only-a-pointer trick that general-
3 ly underlies the performance advantage of moves over copies isn't applicable.

4 The motivation for the SSO is extensive evidence that for many applications, short
5 strings are the norm. Using an internal buffer to store such strings' values elimi-
6 nates the need to dynamically allocate memory for them, and that's typically an
7 efficiency win. An implication of the win, however, is that moves are no faster than
8 copies, though one could just as well take a glass-half-full approach and say that
9 for such strings, copying is no slower than moving.

10 Even for types supporting speedy move operations, some seemingly sure-fire
11 move situations can still end up making copies. Item 16 explains that some con-
12 tainer operations in the Standard Library offer the strong exception safety guaran-
13 tee, and that to ensure that legacy C++98 code dependent on these exception safe-
14 ty guarantees isn't broken when upgrading to C++11, the underlying copy opera-
15 tions may be replaced with move operations only if the move operations are
16 known not to throw. A consequence is that even if a type offers move operations
17 that are more efficient than the corresponding copy operations, and even if, at a
18 particular point in the code, a move operation would generally be appropriate
19 (e.g., if the source object is an rvalue), compilers might still be forced to invoke a
20 copy operation, because the corresponding move operation isn't declared noex-
21 cept.

22 There are thus several scenarios in which C++11's move semantics do you no
23 good:

- 24 • **No move operations:** The object to be moved from fails to offer move opera-
25 tions. The move request therefore becomes a copy request.
- 26 • **Move not faster:** The object to be moved from has move operations that are
27 no faster than its copy operations.
- 28 • **Move not usable:** The context in which the moving would take place requires
29 a nothrow move operation, but that operation isn't declared noexcept.

1 It's worth mentioning, too, another scenario where move semantics offers no efficiency gain:
2

3 • **Source object is lvalue:** With very few exceptions (see e.g., Item 27) only
4 rvalues may be used as the source of a move operation.

5 But the title of this Item is to *assume* that move operations are not present, not
6 cheap, and not used. This is typically the case in generic code, e.g., when writing
7 templates, because you don't know all the types you're working with. In such cir-
8 cumstances, you must be as conservative about copying objects as you were in
9 C++98—before move semantics existed. This is also the case for “unstable” code,
10 i.e., code where the characteristics of the types being used are subject to relatively
11 frequent modification.

12 Often, however, you know the types your code uses, and you can rely on their
13 characteristics not changing (e.g., whether they support inexpensive move opera-
14 tions). When that's the case, you don't need to make assumptions. You can simply
15 look up the move support details for the types you're using. If those types offer
16 cheap move operations, and if you're using objects in contexts where those move
17 operations will be invoked, there's no need for assumptions: you can safely rely on
18 move semantics to replace copy operations with their less expensive move coun-
19 terparts.

20 **Things to remember**

- 21 • Assume that move operations are not present, not cheap, and not used.
22 • In code with known types or support for move semantics, there is no need for
23 assumptions.

24 **Item 32: Familiarize yourself with perfect forwarding fail-
25 ure cases.**

26 One of the features most prominently emblazoned on the C++11 box is perfect
27 forwarding. *Perfect* forwarding. It's *perfect!* Alas, tear the box open, and you'll find
28 that there's “perfect” (the ideal), and then there's “perfect” (the reality). C++11's
29 perfect forwarding is very good, but it achieves true perfection only if you're will-

1 ing to overlook an epsilon or two. This Item is devoted to familiarizing you with
2 the epsilons.

3 Before embarking on our epsilon exploration, it's worthwhile to review what's
4 meant by "perfect forwarding." "Forwarding" just means that one function pass-
5 es—*forwards*—its parameters to another function. The goal is for the second func-
6 tion (the one being forwarded to) to receive the same objects that the first function
7 (the one doing the forwarding) received. That rules out by-value parameters, be-
8 cause they're *copies* of what the original caller passed in; we want the forwarded-
9 to function to be able to work with the originally-passed-in objects. Pointer pa-
10 rameters are also ruled out, because we don't want to force callers to pass point-
11 ers. When it comes to general-purpose forwarding, we'll be dealing with parame-
12 ters that are references.

13 *Perfect* forwarding means we don't just forward objects, we also forward their sa-
14 lient characteristics: their types, whether they're lvalues or rvalues, and whether
15 they're `const` or `volatile`. In conjunction with the observation that we'll be deal-
16 ing with reference parameters, this implies that we'll be using universal references
17 (see Item 26), because only universal reference parameters encode information
18 about the lvalueness and rvalueness of the arguments that are passed to them.

19 Let's assume we have some function `f`, and we'd like to write a function (in truth, a
20 function template) that forwards to it. The core of what we need looks like this:

```
21 template<typename T>
22 void fwd(T&& param)           // accept any argument
23 {
24     f(std::forward<T>(param)); // forward it to f
25 }
```

26 Forwarding functions are, by their nature, generic. The `fwd` template, for example,
27 accepts any type of argument, and it forwards whatever it gets. A logical extension
28 of this genericity is for forwarding functions to be not just templates, but *variadic*
29 templates, thus accepting any number of arguments. The variadic form for `fwd`
30 looks like this:

```
31 template<typename... Ts>
32 void fwd(Ts&&... params)      // accept any arguments
33 {
```

```
1     f(std::forward<Ts>(params)...);    // forward them to f
2 }
```

3 This is the form you'll see in, among other places, the standard containers' em-
4 placement functions (see Item 18) and the smart pointer factory functions,
5 `std::make_shared` and `std::make_unique` (see Item 23).

6 Given our target function `f` and our forwarding function `fwd`, perfect forwarding
7 *fails* if calling `f` with a particular argument does one thing, but calling `fwd` with the
8 same argument does something different:

```
9 f( expression );      // if this does one thing,
10 fwd( expression );   // but this does something else, fwd fails
11                      // to perfectly forward expression to f
```

12 Several kinds of arguments lead to this kind of failure. Knowing what they are and
13 how to work around them is important, so let's tour the arguments that can't be
14 perfect-forwarded.

15 **Braced initializers**

16 Suppose `f` is declared like this:

```
17 void f(const std::vector<int>& v);
```

18 In that case, calling `f` with a braced initializer compiles,

```
19 f({ 1, 2, 3 });      // fine, "{1, 2, 3}" implicitly
20                      // converted to std::vector<int>
```

21 but passing the same braced initializer to `fwd` doesn't compile:

```
22 fwd({ 1, 2, 3 });   // error! doesn't compile
```

23 That's because the use of a braced initializer is a perfect forwarding failure cases.

24 All such failure cases have the same cause. In a direct call to `f` (such as
25 `f({ 1, 2, 3 })`), compilers see the types of arguments passed at the call site,
26 and they see the types of the parameters declared by `f`. They compare the argu-
27 ments at the call site to the parameter declarations to see if they're compatible,
28 and, if necessary, they perform implicit type conversions to make the call succeed.

- 1 In the example above, they generate a temporary `std::vector<int>` object from
2 `{ 1, 2, 3 }` so that `f`'s parameter `v` has a `std::vector<int>` object to bind to.
- 3 When calling `f` indirectly through the forwarding function template `fwd`, compilers
4 no longer compare the arguments passed at `fwd`'s call site to the parameter decla-
5 rations in `f`. Instead, they *deduce* the types of the arguments being passed to `fwd`,
6 and they compare the deduced types to `f`'s parameter declarations. Perfect for-
7 warding fails when either of the following occurs:
- 8 • **Compilers are unable to deduce a type** for one or more of `fwd`'s parameters.
9 In this case, the code fails to compile.
- 10 • **Compilers deduce the “wrong” type** for one or more of `fwd`'s parameters.
11 Here, “wrong” could mean that `fwd`'s instantiation won't compile with the
12 types that were deduced, but it could also mean that the call to `f` using `fwd`'s
13 deduced types behaves differently from a direct call to `f` with the arguments
14 that were passed to `fwd`. One source of such divergent behavior would be if `f`
15 were an overloaded function name, and, due to “incorrect” type deduction, the
16 overload of `f` called inside `fwd` were different from the overload that would be
17 invoked if `f` were called directly.
- 18 In the “`fwd({ 1, 2, 3 })`” call above, the problem is that passing a braced ini-
19 tializer to a function template parameter that's not declared to be a
20 `std::initializer_list` is decreed to be, as the Standard puts it, a “non-
21 deduced context.” In plain English, that means that compilers are forbidden from
22 deducing a type for the expression `{ 1, 2, 3 }` in the call to `fwd`, because `fwd`'s
23 parameter isn't declared to be a `std::initializer_list`. Being prevented from
24 deducing a type for `fwd`'s parameter, compilers must understandably reject the
25 call.
- 26 Interestingly, Item 2 explains that type deduction succeeds for `auto` variables ini-
27 tialized with a braced initializer. Such variables are deemed to be
28 `std::initializer_list` objects, and this affords a simple workaround for cases
29 where the type the forwarding function should deduce is a

1 `std::initializer_list`: declare a local variable using `auto`, then pass the local
2 variable to the forwarding function:

```
3 auto il = { 1, 2, 3 };      // il's type deduced to be
4                               // std::initializer_list<int>
5 fwd(il);                  // fine, perfect-forwards il to f
```

6 **0 or NULL as null pointers**

7 Item 8 explains that when you try to pass `0` or `NULL` as a null pointer to a template,
8 type deduction goes awry, deducing an integral type (typically `int`) instead of a
9 pointer type for the argument you pass. The result is that neither `0` nor `NULL` can
10 be perfect-forwarded as a null pointer. The fix is easy, however: pass `nullptr` in-
11 stead of `0` or `NULL`. For details, consult Item 8.

12 **Declaration-only integral static const data members**

13 As a general rule, there's no need to define integral `static const` data members
14 in classes; declarations alone suffice. That's because compilers perform *const-*
15 *propagation* on such members' values, thus eliminating the need to set aside
16 memory for them. For example, consider this code:

```
17 class Widget {
18 public:
19     static const std::size_t MinVals = 28; // MinVals' declaration
20 ...
21 };
22 ...                                // no defn. for MinVals
```



```
23 std::vector<int> widgetData;
24 widgetData.reserve(Widget::MinVals);    // use of MinVals
```

25 Here, we're using `Widget::MinVals` (henceforth simply `MinVals`) to specify
26 `widgetData`'s initial capacity, even though `MinVals` lacks a definition. Compilers
27 work around the missing definition (as they are required to do) by plopping the
28 value `28` into all places where `MinVals` is mentioned; the fact that no storage has
29 been set aside for `MinVals`' value is unproblematic. If `MinVals`' address were to
30 be taken (e.g., if somebody created a pointer to `MinVals`), then `MinVals` would
31 require storage (so that the pointer had something to point to), and the code

1 above, though it would compile, would fail at link-time until a definition for `MinVals`
2 was provided.

3 With that in mind, imagine that `f` (the function `fwd` forwards its argument to) is
4 declared like this:

5 `void f(std::size_t val);`

6 Calling `f` with `MinVals` is fine, because compilers will just replace `MinVals` with
7 its value:

8 `f(Widget::MinVals); // fine, treated as "f(28)"`

9 Alas, things may not go so smoothly if we try to call `f` through `fwd`:

10 `fwd(Widget::MinVals); // error! shouldn't link`

11 This code will compile, but it shouldn't link. If that reminds you of what happens if
12 we write code that takes `MinVals`' address, that's good, because the underlying
13 problem is the same.

14 Although nothing in the source code takes `MinVals`' address, `fwd`'s parameter is a
15 universal reference, and references, in the code generated by compilers, are usually
16 treated like pointers. In the program's underlying binary code (and on the
17 hardware), pointers and references are essentially the same thing. At this level,
18 there's truth to the adage that references are simply pointers that are automatically
19 dereferenced. That being the case, passing `MinVals` by reference is effectively
20 the same as passing it by pointer, and as such, there has to be some memory for
21 the pointer to point to. Passing integral `static const` data members by reference,
22 then, generally requires that they be defined, and that requirement can cause code
23 using perfect forwarding to fail where the equivalent code without perfect forward-
24 warding succeeds.

25 But perhaps you noticed the weasel words I sprinkled through the preceding dis-
26 cussion. The code "shouldn't" link. References are "usually" treated like pointers.
27 Passing integral `static const` data member by reference "generally" requires
28 that they be defined. It's almost like I know something I don't really want to tell
29 you...

1 That's because I do. According to the Standard, passing `MinVals` by reference re-
2 quires that it be defined. But not all build systems (i.e., compiler-linker combina-
3 tions) enforce this requirement. So, depending on your build system, you may find
4 that you can perfect-forward integral `static const` data members that haven't
5 been defined. If you do, congratulations, but there is no reason to expect such code
6 to port. To make it portable, simply provide a definition for the integral `static`
7 `const` data member in question. For `MinVals`, that'd look like this:

8 `const std::size_t Widget::MinVals; // in Widget's .cpp file`

9 Note that the definition doesn't repeat the initializer (28, in the case of `MinVals`).
10 Don't stress over this detail, however. If you forget and provide the initializer in
11 both places, your compilers will complain, thus reminding you to specify it only
12 once.

13 **Overloaded function names and template names**

14 Suppose our function `f` (the one we keep wanting to forward arguments to via
15 `fwd`) can have its behavior customized by passing it a function that does some of
16 its work. Assuming this function takes and returns `ints`, `f` could be declared like
17 this:

18 `void f(int (*pf)(int)); // pf = "processing function"`

19 It's worth noting that `f` could also be declared using a simpler non-pointer syntax.
20 Such a declaration would look like this, though it'd have the same meaning as the
21 declaration above:

22 `void f(int pf(int)); // declares same f as above`

23 Either way, now suppose we have an overloaded function, `processVal`:

24 `int processVal(int value);`
25 `int processVal(int value, int priority);`

26 We can pass `processVal` to `f`,

27 `f(processVal); // fine`

28 but it's something of a surprise that we can. `f` demands a pointer to a function as
29 its argument, but `processVal` isn't a function pointer or even a function, it's the

1 name of two different functions. However, compilers know which `processVal`
2 they need: the one matching `f`'s parameter type. They thus choose the `pro-`
3 `cessVal` taking one `int`, and they pass that function's address to `f`.

4 What makes this work is that `f`'s declaration lets compilers figure out which ver-
5 sion of `processVal` is required. `fwd`, however, being a function template, doesn't
6 have any information about what type it needs, and that makes it impossible for
7 compilers to determine which overload should be passed:

8 `fwd(processVal);` // error! which `processVal`?

9 `processVal` alone has no type. Without a type, there can be no type deduction,
10 and without type deduction, we're left with another perfect forwarding failure
11 case.

12 The same problem arises if we try to use a function template instead of (or in addi-
13 tion to) an overloaded function name. A function template doesn't represent one
14 function, it represents *many* functions:

15 `template<typename T>`
16 `void workOnVal(T param)` // template for processing values
17 `{ ... }`

18 `fwd(workOnVal);` // error! which `workOnVal`
19 // instantiation?

20 The way to get a perfect-forwarding function like `fwd` to accept an overloaded
21 function name or a template name is to manually specify the overload or instantia-
22 tion you want to have forwarded. For example, you can create a function pointer of
23 the same type as `f`'s parameter, initialize that pointer with `processVal` or `wor-`
24 `kOnVal` (thus causing the proper version of `processVal` to be selected or the
25 proper instantiation of `workOnVal` to be generated), and pass the pointer to `fwd`:

26 `using ProcessFuncType =` // make typedef;
27 `int (*)(int);` // see Item 9

28 `ProcessFuncType processValPtr = processVal;` // specify needed
29 // signature for
30 // `processVal`

31 `fwd(processValPtr);` // fine

```
1 fwd(static_cast<ProcessFuncType>(workOnVal)); // also fine
2 Of course, this requires that you know the type of function pointer that fwd is for-
3 warding to. It's not unreasonable to assume that a perfect-forwarding function will
4 document that. After all, perfect-forwarding functions are designed to accept any-
5 thing, so if there's no documentation telling you what to pass, how would you
6 know?
```

7 Bitfields

```
8 The final failure case for perfect forwarding is when a bitfield is used as a function
9 argument. To see what this means in practice, observe that an IPv4 header can be
10 modeled as follows:†
```

```
11 struct IPv4Header {
12     std::uint32_t version:4,
13         IHL:4,
14         DSCP:6,
15         ECN:2,
16         totalLength:16;
17     ...
18 };
```

```
19 If our long-suffering function f (the perennial target of our forwarding function
20 fwd) is declared to take a std::size_t parameter, calling it with, say, the total-
21 Length field of an IPv4Header object compiles without fuss:
```

```
22 void f(std::size_t sz);           // function to call
23 IPv4Header h;
24 ...
25 f(h.totalLength); // fine
```

```
26 Trying to forward h.totalLength to f via fwd, however, is a different story:
```

```
27 fwd(h.totalLength); // error!
```

[†] This assumes that bitfields are laid out lsb (least significant bit) to msb (most significant bit). C++ doesn't guarantee that, but compilers often provide a mechanism that allows programmers to control bitfield layout.

1 The problem is that `fwd`'s parameter is a reference, and `h.totalLength` is a non-
2 `const` bitfield. That may not sound so bad, but the C++ Standard condemns the
3 combination in unusually clear prose: “A non-`const` reference shall not be bound
4 to a bit-field.” There’s an excellent reason for the prohibition. Bitfields may consist
5 of arbitrary parts of machine words (e.g., bits 3-5 of a 32-bit `int`), but there’s no
6 way to directly address such things. I mentioned earlier that references and point-
7 ers are the same thing at the hardware level, and just as there’s no way to create a
8 pointer to arbitrary bits (C++ dictates that the smallest thing you can point to is a
9 `char`), there’s no way to bind a reference to arbitrary bits, either.

10 Working around the impossibility of perfect-forwarding a bitfield is easy, once you
11 realize that any function that accepts a bitfield as an argument will receive a *copy*
12 of the bitfield’s value. After all, no function can bind a reference to a bitfield, nor
13 can any function accept pointers to bitfields, because pointers to bitfields don’t
14 exist. The only kinds of parameters to which a bitfield can be passed are by-value
15 parameters and, counterintuitively, references-to-`const`. In the case of by-value
16 parameters, the called function obviously receives a copy of the value in the bit-
17 field, and it turns out that in the case of a reference-to-`const` parameter, the
18 Standard requires that the reference actually bind to a *copy* of the bitfield’s value
19 that’s stored in an object of some standard integral type (e.g., `int`). References-to-
20 `const` don’t bind to bitfields, they bind to “normal” objects into which the values
21 of the bitfields have been copied.

22 The key to passing a bitfield into a perfect-forwarding function, then, is to take ad-
23 vantage of the fact that the forwarded-to function will always receive a copy of the
24 bitfield’s value. You can thus make a copy yourself and call the forwarding function
25 with the copy. In the case of our example with `IPv4Header`, this code would do the
26 trick:

```
27 // copy bitfield value; see Item 6 for info on init. form
28 auto length = static_cast<std::uint16_t>{h.totalLength};
29 fwd(length);                                // forward the copy
```

1 **Upshot**

2 In most cases, perfect forwarding works exactly as advertised. You rarely have to
3 think about it. But when it doesn't work—when reasonable-looking code fails to
4 compile or, worse, compiles, but doesn't behave the way you anticipate, it's im-
5 portant to know about perfect forwarding's imperfections. Equally important is
6 knowing how to work around them. In most cases, this is straightforward.

7 **Things to remember**

- 8 ♦ Perfect forwarding fails when template type deduction fails or when it deduces
9 the wrong type.
- 10 ♦ The kinds of arguments that lead to perfect forwarding failure are braced ini-
11 tializers, null pointers expressed as `0` or `NULL`, declaration-only integral `const`
12 `static` data members, template and overloaded function names, and bitfields.

1 **Chapter 6 Lambda Expressions**

2 Lambda expressions—*lambdas*—are a game-changer in C++ programming. That's
3 somewhat surprising, because they bring no new expressive power to the lan-
4 guage. Everything a lambda can do is something you can do by hand with a bit
5 more typing. But lambdas are such a convenient way to create function objects, the
6 impact on day-to-day C++ software development is enormous. Without lambdas,
7 the STL “_if” algorithms (e.g., `std::find_if`, `std::remove_if`,
8 `std::count_if`, etc.) tend to be employed with only the most trivial predicates,
9 but when lambdas are available, use of these algorithms with nontrivial conditions
10 blossoms. The same is true of algorithms that can be customized with comparison
11 functions (e.g., `std::sort`, `std::nth_element`, `std::lower_bound`, etc.). Out-
12 side the STL, lambdas make it possible to quickly create custom deleters for
13 `std::unique_ptr` and `std::shared_ptr` (see Items 20 and 21), and they make
14 the specification of predicates for condition variables in the threading API equally
15 straightforward. Beyond the Standard Library, lambdas facilitate the on-the-fly
16 specification of callback functions, interface adaption functions, and context-
17 specific functions for one-off calls. Lambdas really make C++ a more pleasant pro-
18 gramming language.

19 The Items in this chapter cover important dos and don'ts for effective software
20 development with lambdas. The chapter begins with an admonition to steer clear
21 of what initially looks like an attractive feature: default capture modes. It then ad-
22 dresses how to accomplish move capture in C++14 (where it's more or less direct-
23 ly supported) and C++11 (where the way is more circuitous). That's followed by a
24 C++14-specific Item on implementing perfect-forwarding with generic lambdas,
25 and the chapter concludes with an Item explaining why programmers accustomed
26 to `std::bind` should switch to lambdas.

27 The vocabulary associated with lambdas can be confusing. Here's a brief refresher:

- 28 • A *lambda expression* is just that: an expression. It's part of the source code. In

```
29     std::find_if(container.begin(), container.end(),
30                   [] (auto val) { return 0 < val && val < 10; });
```

1 the highlighted expression is the lambda.

2 • A *closure* is the runtime object created by a lambda. Closures hold copies of all
3 captured data (although by-reference captures are typically optimized away).
4 In the call to `std::find_if` above, the closure is the object that's passed at
5 runtime as the third argument to `std::find_if`. (The first two arguments are
6 the begin and end iterators for *container*.)

7 • A *closure class* is a class from which a closure is instantiated. Each lambda
8 causes compilers to generate a unique closure class. The statements inside a
9 lambda become executable instructions in the member functions of its closure
10 class (modulo inlining and other optimizations).

11 A lambda is often used to create a closure that's used only as an argument to a
12 function. That's the case in the call to `std::find_if` above. However, closures
13 may generally be copied, so it's usually possible to have multiple closures of a clo-
14 sure type corresponding to a single lambda. For example, in the following code,

```
15 {                                     // begin a scope
16   int x;                           // x is local variable
17 ...
18   auto c1 =                         // c1 is copy of the
19     [x](int y) { return x * y = 5; }; // closure produced
20                               // by the lambda
21   auto c2 = c1;                     // c2 is copy of c1
22   auto c3 = c2;                     // c3 is copy of c2
23 ...
24 }                                     // end the scope
```

25 `c1, c2, and c3` are all copies of the closure produced by the lambda.

26 Informally, it's perfectly acceptable to blur the lines between lambdas, closures,
27 and closure classes. But in the Items that follow, it's often important to distinguish
28 what exists during compilation (lambdas and closure classes), what exists at
29 runtime (closures), and how they relate to one another.

1 **Item 33: Avoid default capture modes.**

2 There are two default capture modes in C++11: by-reference and by-value. Default
3 by-reference capture can lead to easy-to-overlook dangling references. Default by-
4 value capture lures you into thinking you're immune to that problem (you're not),
5 and it lulls you into thinking your closures are self-contained (they may not be).

6 That's the executive summary for this Item. If you're more engineer than execu-
7 tive, you'll want some meat on those bones, so let's start with the danger of default
8 by-reference capture.

9 A by-reference capture causes the code inside the lambda to refer to a local varia-
10 ble or a parameter that's available in the scope where the lambda is defined. If the
11 lifetime of a closure created from that lambda exceeds the lifetime of the local var-
12 iable or parameter, the reference in the code will dangle. For example, suppose we
13 have a container of filtering functions, each of which takes an `int` and returns a
14 `bool` indicating whether a passed-in value satisfies the filter:

```
15  using FilterContainer =           // typedef;
16  std::vector<std::function<bool(int)>>;    // see Item 9
17  FilterContainer filters;          // filtering funcs
```

18 We could add a filter for multiples of 5 like this:

```
19  filters.emplace_back(           // see Item 18 for
20  [](int value) { return value % 5 == 0; } // info on
21  );                                // emplace_back
```

22 However, it may be that we need to compute the divisor at runtime, i.e., we can't
23 just hard-code 5 into the lambda. So adding the filter might look more like this:

```
24  void addDivisorFilter()
25  {
26      auto calc1 = computeSomeValue1();
27      auto calc2 = computeSomeValue2();

28      auto divisor = computeDivisor(calc1, calc2);

29      filters.emplace_back(           // danger!
30          [&](int value) { return value % divisor == 0; } // divisor
31      );                                // might
32  }                                    // dangle!
```

1 This code is a problem waiting to happen. The lambda refers to the local variable
2 `divisor`, but that variable ceases to exist when `addDivisorFilter` returns.
3 That's immediately after `filters.emplace_back` returns, so the function that's
4 added to `filters` is essentially dead on arrival. Using that filter yields undefined
5 behavior from virtually the moment it's created.

6 Now, the same problem would exist if `divisor`'s by-reference capture were ex-
7 plicit,

```
8     filters.emplace_back(  
9         [&divisor](int value)           // danger! divisor can  
10        { return value % divisor == 0; }    // still dangle!  
11    );
```

12 but with an explicit capture, it's easier to see that the viability of the lambda is de-
13 pendent on `divisor`'s lifetime. Also, writing out the name, "divisor," reminds us to
14 ensure that `divisor` lives at least as long as the lambda's closures. That's a more
15 specific memory jog than the general "make sure nothing dangles" admonition that
16 "[&]" conveys.

17 If you know that the lambda will be used only once (e.g., in a call to an STL algo-
18 rithm), there is no risk that its closure will outlive the local variables and parame-
19 ters in the environment where the lambda is created. In that case, you might argue,
20 there's no risk of dangling references, hence no reason to avoid a default by-
21 reference capture mode. For example, our filtering lambda might be used only as
22 an argument to C++11's `std::all_of`, which returns whether all elements in a
23 range satisfy a condition:

```
24 template<typename C>  
25 void workWithContainer(const C& container)  
26 {  
27     auto calc1 = computeSomeValue1();           // as above  
28     auto calc2 = computeSomeValue2();           // as above  
29     auto divisor = computeDivisor(calc1, calc2); // as above  
30     using ContElemT = typename C::value_type;   // type of  
31                                         // elements in  
32                                         // container  
33     if (std::all_of(  
34         container.begin(), container.end(),
```

```
1      [&](const ContElemT& value)           // are multiples
2      { return value % divisor == 0; })       // of divisor...
3  ) {
4  ...
5 } else {
6 ...
7 }
8 }
```

9 It's true, this is safe, but its safety is somewhat precarious. If the lambda were
10 found to be useful in other contexts (e.g., as a function to be added to the `filters`
11 container) and was copy-and-pasted into a context where its closure could outlive
12 `divisor`, you'd be back in dangle-city, and there'd be nothing in the capture clause
13 to specifically remind you to perform lifetime analysis on `divisor`.

14 Long-term, it's simply better software engineering to explicitly list the local variables
15 and parameters that a lambda depends on.

16 By the way, the ability to use `auto` in C++14 lambda parameter specifications
17 means that the code above can be simplified in C++14. The `ContElemT` typedef can
18 be eliminated, and the `if` condition can be revised as follows:

```
19 if (std::all_of(container.begin(), container.end(),
20                   [&](const auto& value)           // C++14
21                   { return value % divisor == 0; })) // only
```

22 One way to solve our problem with `divisor` would be a default by-value capture
23 mode. That is, we could add the lambda to `filters` as follows:

```
24 filters.emplace_back(                  // now
25   [=](int value) { return value % divisor == 0; }    // divisor
26 );                                // can't
27                                     // dangle
```

28 This suffices for this example, but, in general, default by-value capture isn't the anti-dangling elixir you might imagine. The problem is that if you capture a pointer
29 by value, you copy the pointer into the closures arising from the lambda, but you
30 don't prevent code outside the lambda from `delete`ing the pointer and causing
31 your copies to dangle.
32

33 "That could never happen!" you protest. "Having read Chapter 4, I worship at the
34 house of smart pointers. Only loser C++98 programmers use raw pointers and de-

1 ~~lete~~." That may be true, but it's irrelevant, because you do, in fact, use raw pointers,
2 and they can, in fact, be deleted out from under you. It's just that in your
3 modern C++ programming style, there's often little sign of it in the source code.

4 Suppose one of the things `Widgets` can do is add entries to the container of filters:

```
5 class Widget {
6 public:
7     ...                                // ctors, etc.
8     void addFilter() const;           // add an entry to filters
9 private:
10    int divisor;                   // used in Widget's filter
11};
```

12 `Widget::addFilter` could be defined like this:

```
13 void Widget::addFilter() const
14 {
15     filters.emplace_back(
16         [=](int value) { return value % divisor == 0; }
17     );
18 }
```

19 To the blissfully uninitiated, this looks like safe code. The lambda is dependent on
20 `divisor`, but the default by-value capture mode ensures that `divisor` is copied
21 into any closures arising from the lambda, right?

22 Wrong. Completely wrong. Horribly wrong. Fatally wrong.

23 Captures apply only to non-`static` local variables (including parameters) visible
24 in the scope where the lambda is created. In the body of `Widget::addFilter`,
25 `divisor` is not a local variable, it's a data member of the `Widget` class. It can't be
26 captured. Yet if the default capture mode is eliminated, the code won't compile:

```
27 void Widget::addFilter() const
28 {
29     filters.emplace_back(                // error!
30         [](int value) { return value % divisor == 0; } // divisor
31     );                                // not
32 }                                    // available
```

1 Furthermore, if an attempt is made to explicitly capture `divisor` (either by value
2 or by reference—it doesn't matter), the capture won't compile, because `divisor`
3 isn't a local variable or a parameter:

```
4 void Widget::addFilter() const
5 {
6     filters.emplace_back(
7         [divisor](int value)           // error! no local
8             { return value % divisor == 0; } // divisor to capture
9     );
10 }
```

11 So if the default by-value capture clause isn't capturing `divisor`, yet without the
12 default by-value capture clause, the code won't compile, what's going on?

13 The explanation hinges on your implicit use of a raw pointer: `this`. Every non-
14 `static` member function has a `this` pointer, and you use that pointer every time
15 you mention a data member of the class. Inside any `Widget` member function, for
16 example, compilers internally replace uses of `divisor` with `this->divisor`. In
17 the version of `Widget::addFilter` with a default by-value capture,

```
18 void Widget::addFilter() const
19 {
20     filters.emplace_back(
21         [=](int value) { return value % divisor == 0; }
22     );
23 }
```

24 what's being captured is the `Widget`'s `this` pointer, not `divisor`. Compilers treat
25 the code as if it had been written as follows:

```
26 void Widget::addFilter() const
27 {
28     auto currentObjectPtr = this;
29
30     filters.emplace_back(
31         [currentObjectPtr](int value)
32             { return value % currentObjectPtr->divisor == 0; }
33     );
34 }
```

34 Understanding this is tantamount to understanding that the viability of the clo-
35 sures arising from this lambda are tied to the lifetime of the `Widget` whose `this`

1 pointer they contain a copy of. In particular, consider this code, which, in accord
2 with Chapter 4, uses pointers of only the smart variety:

```
3 using FilterContainer = // as before
4     std::vector<std::function<bool(int)>>;
5
6 FilterContainer filters; // as before
7
8 void doSomeWork()
9 {
10     auto pw = // create Widget; see
11         std::make_unique<Widget>(); // Item 23 for
12         // std::make_unique
13
14     pw->addFilter(); // add filter that uses
15         // Widget::divisor
16     ...
17 }
18
19     // destroy Widget; filter
20     // now holds dangling pointer!
```

21 When a call is made to `doSomeWork`, a filter is created that depends on the `Widget`
22 object produced by `std::make_unique`, i.e., a filter that contains a copy of a
23 pointer to that `Widget`—the `Widget`'s `this` pointer. This filter is added to `filters`, but when `doSomeWork` finishes, the `Widget` is destroyed by the
24 `std::unique_ptr` managing its lifetime (see Item 20). From that point on, `filters`
25 contains an entry with a dangling pointer.

26 This particular problem can be solved by making a local copy of the data member
27 you want to capture and then capturing the copy:

```
28 void Widget::addFilter() const
29 {
30     auto divisorCopy = divisor; // copy data member
31
32     filters.emplace_back(
33         [divisorCopy](int value) // capture the copy
34         { return value % divisorCopy == 0; } // use the copy
35     );
36 }
```

37 To be honest, if you take this approach, default by-value capture will work, too,

```
38 void Widget::addFilter() const
39 {
40     auto divisorCopy = divisor; // copy data member
```

```
1   filters.emplace_back(
2     [=](int value) // capture the copy
3     { return value % divisorCopy == 0; } // use the copy
4   );
5 }
```

6 but why tempt fate? A default capture mode is what made it possible to accidentally
7 capture `this` when you thought you were capturing `divisor` in the first place.

8 In C++14, a better way to capture a data member is to use generalized lambda cap-
9 ture (see Item 34):

```
10 void Widget::addFilter() const
11 {
12   filters.emplace_back( // C++14 only:
13     [divisor = divisor](int value) // copy divisor to closure
14     { return value % divisor == 0; } // use the copy
15   );
16 }
```

17 There's no such thing as a default capture mode for a generalized lambda capture,
18 however, so even in C++14, the advice of this Item—to avoid default capture
19 modes—stands.

20 An additional drawback to default by-value captures is that they can suggest that
21 the corresponding closures are self-contained and insulated from changes to data
22 outside the closures. In general, that's not true, because lambdas may be depend-
23 ent not just on local variables and parameters (which may be captured), but also
24 on objects with *static storage duration*. Such objects are defined at global or
25 namespace scope or are declared `static` inside classes, functions, or files. These
26 objects can be used inside lambdas, but they can't be captured. Yet specification of
27 a default by-value capture mode can lend the impression that they are. Consider
28 this revised version of the `addDivisorFilter` template we saw earlier:

```
29 void addDivisorFilter()
30 {
31   static auto calc1 = computeSomeValue1(); // now static
32   static auto calc2 = computeSomeValue2(); // now static
33   static auto divisor = // now static
34     computeDivisor(calc1, calc2);
```

```

1   filters.emplace_back(
2     [=](int value) // captures nothing!
3     { return value % divisor == 0; } // refers to above static
4   );
5
6   ++divisor; // modify divisor

```

7 A casual reader of this code could be forgiven for seeing “[=]” and thinking, “Okay,
 8 the lambda makes a copy of all the objects it uses and is therefore self-contained.”
 9 But it’s not self-contained. This lambda doesn’t use any non-static local variables,
 10 so nothing is captured. Rather, the code for the lambda refers to the **static** varia-
 11 ble **divisor**. When, at the end of each invocation of **addDivisorFilter**, **divisor**
 12 is incremented, any lambdas that have been added to **filters** via this function
 13 will exhibit new behavior (corresponding to the new value of **divisor**). Practically
 14 speaking, this lambda captures **divisor** by reference, a direct contradiction to
 15 what the default by-value capture clause seems to imply. If you stay away from
 16 default by-value capture clauses, you eliminate the risk of your code being misread
 17 in this way.

18 Things to Remember

- 19 ♦ Default by-reference capture can lead to hidden dangling references.
- 20 ♦ Default by-value capture is susceptible to hidden dangling pointers (especially
21 **this**), and it misleadingly suggests that lambdas are self-contained.

22 Item 34: Use init capture to move objects into closures.

23 Sometimes neither by-value capture nor by-reference capture is what you want. If
 24 you have a move-only object (e.g., a **std::unique_ptr** or a **std::future**) that
 25 you want to get into a closure, C++11 offers no way to do it. If you have an object
 26 that’s expensive to copy but cheap to move (e.g., most containers in the Standard
 27 Library), and you’d like to get that object into a closure, you’d much rather move it
 28 than copy it. Again, however, C++11 gives you no way to accomplish that.

29 But that’s C++11. C++14 is a different story. It offers direct support for moving ob-
 30 jects into closures. If your compilers are C++14-compliant, rejoice and read on. If

1 you're still working with C++11 compilers, you should rejoice and read on, too,
2 because there are ways to approximate move capture in C++11.

3 The absence of move capture was recognized as a shortcoming even as C++11 was
4 adopted. The straightforward remedy would have been to add it in C++14, but the
5 Standardization Committee chose a different path. They introduced a new capture
6 mechanism that's so flexible, capture-by-move is only one of the tricks it can per-
7 form. The new capability is called *init capture*. It can do virtually everything the
8 C++11 capture forms can do, plus more. The one thing you can't express with an
9 init capture is a default capture mode, but Item 33 explains that you should stay
10 away from those, anyway. (For situations covered by C++11 captures, init cap-
11 ture's syntax is a bit wordier, so in cases where a C++11 capture gets the job done,
12 it's perfectly reasonable to use it.)

13 Using an init capture makes it possible for you to specify

- 14 1. **the name of a data member** in the closure class generated from the lambda
15 and
16 2. **an expression** initializing that data member.

17 Here's how you can use init capture to move a `std::unique_ptr` into a closure:

```
18 class Widget {                                     // some useful type
19 public:
20 ...
21     bool isValidated() const;
22     bool isProcessed() const;
23     bool isArchived() const;
24 private:
25 ...
26 };
27
28 auto pw = std::make_unique<Widget>();           // create Widget; see
29                                         // Item 23 for info on
30                                         // std::make_unique
31 ...                                              // configure *pw
```

```
1 auto func = [pw = std::move(pw)]           // init data mbr
2     { return pw->isValidated()           // in closure w/
3         && pw->isArchived(); };      // std::move(pw)
```

4 The highlighted text comprises the init capture. To the left of the “=” is the name of
5 the data member in the closure class you’re specifying, and to the right is the ini-
6 tializing expression. Interestingly, the scope on the left of the “=” is different from
7 the scope on the right. The scope on the left is that of the closure class. The scope
8 on the right is the same as where the lambda is being defined. In the example
9 above, the name `pw` on the left of the “=” refers to a data member in the closure
10 class, while the name `pw` on the right refers to the object declared above the lamb-
11 da, i.e., the variable initialized by the call to `std::make_unique`. So “`pw =`
12 `std::move(pw)`” means “create a data member `pw` in the closure, and initialize
13 that data member with the result of applying `std::move` to the local variable `pw`.”

14 As usual, code in the body of the lambda is in the scope of the closure class, so uses
15 of `pw` there refer to the closure class data member.

16 The comment “configure *`pw`” in this example indicates that after the `Widget` is
17 created by `std::make_unique` and before the `std::unique_ptr` to that `Widget`
18 is captured by the lambda, the `Widget` is modified in some way. If no such configu-
19 ration is necessary, i.e., if the `Widget` created by `std::make_unique` is in a state
20 suitable to be captured by the lambda, the local variable `pw` is unnecessary, be-
21 cause the closure class’s data member can be directly initialized by
22 `std::make_unique`:

```
23 auto func = [pw = std::make_unique<Widget>()] // init data mbr
24     { return pw->isValidated()           // in closure w/
25         && pw->isArchived(); };      // result of call
26                                         // to make_unique
```

27 This should make clear that the C++14 notion of “capture” is considerably general-
28 ized from C++11, because in C++11, it’s not possible to capture the result of an ex-
29 pression. As a result, another name for init capture is *generalized lambda capture*.

30 But what if one or more of the compilers you use lacks support for C++14’s init
31 capture? How can you accomplish move capture in a language lacking support for
32 move capture?

1 Remember that a lambda expression is simply a way to cause a class to be generated
2 and an object of that type to be created. There is nothing you can do with a
3 lambda that you can't do by hand. The example C++14 code we just saw, for example, can be written in C++11 like this:

```
5 class IsValAndArch {                                // "is validated
6 public:                                         // and archived"
7     using DataType = std::unique_ptr<Widget>;
8     explicit IsValAndArch(DataType&& ptr)      // Item 27 explains
9     : pw(std::move(ptr)) {}                      // use of std::move
10    bool operator()() const
11    { return pw->isValidated() && pw->isArchived(); }
12 private:
13     DataType pw;
14 };
15 auto func = IsValAndArch(std::make_unique<Widget>());
```

16 That's more work than writing the lambda, but it doesn't change the fact that if you
17 want a class in C++11 that supports move-initialization of its data members, the
18 only thing between you and your desire is a bit of time with your keyboard.

19 If you want to stick with lambdas (and given their convenience, you probably do),
20 move capture can be emulated in C++11 by

21 1. **moving the object to be captured into a function object produced by**
22 **std::bind** and

23 2. **giving the lambda a reference to the “captured” object.**

24 If you're familiar with `std::bind`, the code is pretty straightforward. If you're not
25 familiar with `std::bind`, the code takes a little getting used to, but it's worth the
26 trouble.

27 Suppose you'd like to create a local `std::vector`, put an appropriate set of values
28 into it, then move it into a closure. In C++14, this is easy:

```
29 std::vector<double> data;                         // object to be moved
30                                         // into closure
31 ...                                              // populate data
```

```
1 auto func = [data = std::move(data)]           // C++14 init capture
2     { /* uses of data */ };
```

3 I've highlighted key parts of this code: the type of object you want to move
4 (`std::vector<double>`), the name of that object (`data`), and the initializing ex-
5 pression for the init capture (`std::move(data)`). The C++11 equivalent is as fol-
6 lows, where I've highlighted the same key things:

```
7 std::vector<double> data;                      // as above
8 ...
9 auto func =
10    std::bind(
11        [](std::vector<double>& data)           // C++11 emulation
12            { /* uses of data */ },
13            std::move(data)
14    );
```

15 Like lambda expressions, `std::bind` produces function objects. I call function ob-
16 jects returned by `std::bind` *bind objects*. The first argument to `std::bind` is al-
17 ways something callable (e.g., a callable entity). Subsequent arguments represent
18 values to be passed to the first argument.

19 A bind object contains copies of all the arguments passed to `std::bind`. For each
20 lvalue argument, the corresponding object in the bind object is copy constructed.
21 For each rvalue, it's move constructed. In this example, the second argument is an
22 rvalue (the result of `std::move`—see Item 25), so `data` is move-constructed into
23 the bind object. This move construction is the crux of move capture emulation, be-
24 cause moving an rvalue into a bind object is how we work around the inability to
25 move an rvalue into a C++11 closure.

26 When a bind object is “called” (i.e., its function call operator is invoked) the objects
27 it stores are passed to the callable entity originally passed to `std::bind`. In this
28 example, that means that when `func` (the bind object) is called, the move-
29 constructed copy of `data` inside `func` is passed as an argument to the lambda that
30 was passed to `std::bind`.

31 This lambda is the same as the lambda we'd use in C++14, except a parameter, `da-`
32 `ta`, has been added to correspond to our pseudo-move-captured object. This pa-

1 parameter is an lvalue reference to the copy of `data` in the bind object. (It's not an
2 rvalue reference, because although the expression used to initialize the copy of
3 `data` ("`std::move(data)`") is an rvalue, the copy of `data` itself is an lvalue.) Uses
4 of `data` inside the lambda will thus operate on the move-constructed copy of `data`
5 inside the bind object.

6 Because a bind object stores copies of all the arguments passed to `std::bind`, the
7 bind object in our example contains a copy of the closure produced by the lambda
8 that is its first argument. The lifetime of the closure is therefore the same as the
9 lifetime of the bind object. That's important, because it means that as long as the
10 closure exists, the bind object containing the pseudo-move-captured object exists,
11 too.

12 If this is your first exposure to `std::bind`, you may need to consult your favorite
13 C++11 reference before all the details of the foregoing discussion fall into place.
14 Even if that's the case, these fundamental points should be clear:

- 15 • It's not possible to move-construct an object into a C++11 closure, but it is pos-
16 sible to move-construct an object into a C++11 bind object.
- 17 • Emulating move-capture in C++11 consists of move-constructing an object into
18 a bind object, then passing the move-constructed object to the lambda by ref-
19 erence.
- 20 • Because the lifetime of the bind object is the same as that of the closure, it's
21 possible to treat objects in the bind object as if they were in the closure.

22 As a second example of using `std::bind` to emulate move capture, here's the
23 C++14 code we saw earlier to create a `std::unique_ptr` in a closure:

```
24 auto func = [pw = std::make_unique<Widget>()] // as before,  
25 { return pw->isValidated() // create pw  
26     && pw->isArchived(); }; // in closure
```

27 And here's the C++11 emulation:

```
28 auto func = std::bind(  
29     [](std::unique_ptr<Widget>& pw)  
30     { return pw->isValidated()
```

```
1           && pw->isArchived(); },
2           std::make_unique<Widget>()
3       );
```

4 It's ironic that I'm showing how to use `std::bind` to work around limitations in
5 C++11 lambdas, because in Item 36, I advocate the use of lambdas over
6 `std::bind`. However, that Item explains that there are some cases in C++11
7 where `std::bind` can be useful, and this is one of them. (In C++14, features such
8 as init capture and `auto` parameters eliminate those cases.)

9 Things to Remember

- 10 • Use C++14's init capture to move objects into closures.
- 11 • In C++11, emulate init capture via hand-written classes or `std::bind`.

12 Item 35: Use `decltype` on `auto&&` parameters to 13 `std::forward` them.

14 One of the most exciting features of C++14 is generic lambdas—lambdas that use
15 `auto` in their parameter specifications. The implementation of this feature is
16 straightforward: the `operator()` in the closure class arising from the lambda is a
17 template. Given this lambda, for example,

```
18 auto f = [](auto x){ return func(normalize(x)); };
```

19 the closure class's function call operator looks like this:

```
20 class SomeCompilerGeneratedClassName {
21 public:
22     template<typename T> // see Item 3 for
23         auto operator()(T x) const // auto return type
24             { return func(normalize(x)); }
```

25 ... // other closure class
26 };// functionality

27 In this example, the only thing the lambda does with its parameter `x` is forward it
28 to `normalize`. If `normalize` treats lvalues differently from rvalues, this lambda
29 isn't really written properly, because it always passes an lvalue (the parameter `x`)
30 to `normalize`, even if the argument that was passed to the lambda was an rvalue.

1 The correct way to write the lambda is to have it perfect-forward `x` to `normalize`.
2 Doing that requires two changes to the code. First, `x` has to become a universal ref-
3 erence (see Item 26), and second, it has to be passed to `normalize` via
4 `std::forward` (see Item 27). In concept, these are trivial modifications:

5 `auto f = [](auto&& x)`
6 `{ return func(normalize(std::forward<??>(x))); }`;

7 Between concept and realization, however, is the question of what type to pass to
8 `std::forward`, ie., to determine what should go where I've written `???` above.

9 Normally, when you employ perfect forwarding, you're in a template function tak-
10 ing some type parameter `T`, so you just write `std::forward<T>`. In the lambda,
11 though, there's no type parameter `T` available to you. There is a `T` in the tem-
12 platized `operator()` inside the closure class generated by the lambda, but it's not
13 possible to refer to it from the lambda, so it does you no good.

14 Item 30 explains that if an lvalue argument is passed to a universal reference pa-
15 rameter, the type of that parameter becomes an lvalue reference. If an rvalue is
16 passed, the parameter becomes an rvalue reference. This means that in our lamb-
17 da, we can determine whether the argument passed was an lvalue or an rvalue by
18 inspecting the type of the parameter, `x decltype` gives us a way to do that (see
19 Item 3). If an lvalue was passed in, `decltype(x)` will produce a type that's an
20 lvalue reference. If an rvalue was passed, `decltype(x)` will produce an rvalue
21 reference type.

22 Item 30 also explains that when calling `std::forward`, the type argument should
23 be an lvalue reference for lvalues, and it should be a non-reference for rvalues. In
24 our lambda, if `x` is bound to an lvalue, `decltype(x)` will yield an lvalue reference,
25 which is exactly what `std::forward` wants. However, if `x` is bound to an rvalue,
26 `decltype(x)` will yield an rvalue reference—not the non-reference that
27 `std::forward` expects.

28 But look at the sample implementation for `std::forward` from Item 30:

29 `template<typename T>` // from Item 30
30 `T&& forward(T&& param)`
31 {

```
1     return static_cast<T&&>(param);
2 }
```

3 If client code wants to perfect-forward an rvalue of type `Widget`, it'll call
4 `std::forward` with the type `Widget` (i.e, a non-reference type), and the
5 `std::forward` template will be instantiated to yield this function:

```
6 Widget&& forward(Widget&& param)           // instantiation of
7 {                                              // std::forward when
8     return static_castWidget&&>(param);      // T is Widget
9 }
```

10 But consider what would happen if the client code wanted to perfect-forward the
11 same rvalue of type `Widget`, but instead of following the convention of specifying `T`
12 to be a non-reference type, it specified it to be an rvalue reference. That is, consider
13 what would happen if `T` were specified to be `Widget&&`. After instantiation of
14 `std::forward` but before reference collapsing (once again, see Item 30),
15 `std::forward` would look like this:

```
16 Widget&& && forward(Widget&& && param)           // instantiation of
17 {                                              // std::forward when
18     return static_castWidget&& &&>(param);      // T is Widget&&
19 }                                              // (before reference-
20                                         // collapsing)
```

21 Applying the reference-collapsing rule that an rvalue reference to an rvalue refer-
22 ence becomes a single rvalue reference, this instantiation emerges:

```
23 Widget&& forward(Widget&& param)           // instantiation of
24 {                                              // std::forward when
25     return static_castWidget&&>(param);      // T is Widget&&
26 }                                              // (after reference-
27                                         // collapsing)
```

28 If you compare this instantiation with the one that results when `std::forward` is
29 called with `T` set to `Widget`, you'll see that they're identical. That means that call-
30 ing `std::forward` with a non-reference type yields the same result as calling it
31 with an rvalue reference type.

32 That's wonderful news, because `decltype(x)` yields an rvalue reference type
33 when an rvalue is passed as an argument to our lambda's parameter `x`. We estab-
34 lished above that when an lvalue is passed to our lambda, `decltype(x)` yields the

1 proper type to pass to `std::forward`, and now we realize that for rvalues,
2 `decltype(x)` yields a type to pass to `std::forward` that's not conventional, but
3 that nevertheless yields the same outcome as the conventional type. So for both
4 lvalues and rvalues, passing `decltype(x)` to `std::forward` gives us the result
5 we want. Our perfect-forwarding lambda can therefore be written like this:

```
6 auto f =
7   [](auto&& x)
8   { return func(normalize(std::forward<decltype(x)>(x))); };
```

9 From there, it's just a hop, skip, and three dots to a perfect-forwarding lambda that
10 accepts not just a single parameter, but any number of parameters, because C++14
11 lambdas can also be variadic:

```
12 auto f =
13   [](auto&&... x)
14   { return func(normalize(std::forward<decltype(x)>(x)...)); };
```

15 Things to Remember

- 16 • Use `decltype` on `auto&&` parameters to `std::forward` them.

17 Item 36: Prefer lambdas to `std::bind`.

18 `std::bind` is the C++11 successor to C++98's `std::bind1st` and `std::bind2nd`,
19 but, informally, it's been part of the Standard Library since 2005. That's when the
20 Standardization Committee adopted a document known as TR1, which included
21 `bind`'s specification. (In TR1, `bind` was in a different namespace, so it was
22 `std::tr1::bind`, not `std::bind`, and a few interface details were different.) This
23 history means that some programmers have a decade or more of experience using
24 `std::bind`. If you're one of them, you may be reluctant to abandon a tool that's
25 served you well. That's understandable, but in this case, change is good, because in
26 C++11, lambdas are almost always a better choice than `std::bind`. As of C++14,
27 the case for lambdas isn't just stronger, it's downright iron-clad.

28 This Item assumes that you're familiar with `std::bind`. If you're not, you'll want
29 to acquire a basic understanding before continuing. Such an understanding is
30 worthwhile in any case, because you never know when you might encounter uses
31 of `std::bind` in a code base you have to read or maintain.

1 The most important reason to prefer lambdas over `std::bind` is that lambdas are
2 more readable. Suppose, for example, we have a function to set up an audible
3 alarm:

```
4 // typedef for a point in time (see Item 9 for syntax)
5 using Time = std::chrono::time_point<std::chrono::steady_clock>;
6 // see Item 10 for "enum class"
7 enum class Sound { Beep, Siren, Whistle };
8 // typedef for a length of time
9 using Duration = std::chrono::steady_clock::duration;
10 // at time t, make sound s for duration d
11 void setAlarm(Time t, Sound s, Duration d);
```

12 Further suppose that at some point in the program, we've determined we'll want
13 an alarm that will go off an hour after it's set and that will stay on for 30 seconds.
14 The alarm sound, however, remains undecided. We can write a lambda that revises
15 `setAlarm`'s interface so that only a sound needs to be specified:

```
16 // setSoundL ("L" for "lambda") is a function object allowing a
17 // sound to be specified for a 30-sec alarm to go off an hour
18 // after it's set
19 auto setSoundL =
20     [] (Sound s)
21     {
22         // make std::chrono components available w/o qualification
23         using namespace std::chrono;
24
25         setAlarm(steady_clock::now() + hours(1),    // alarm to go off
26                  s,                                // in an hour for
27                  seconds(30));                    // 30 seconds
28     };
```

28 I've highlighted the call to `setAlarm` inside the lambda. This is a normal-looking
29 function call, and even a reader with little lambda experience can see that the pa-
30 rameter `s` passed to the lambda is passed as an argument to `setAlarm`.

31 The corresponding `std::bind` call looks like this:

```
32 using namespace std::chrono;           // as above
33 using namespace std::placeholders;      // needed for use of "_1"
34 auto setSoundB =                      // "B" for "bind"
35     std::bind(setAlarm,
```

```
1     steady_clock::now() + hours(1),
2     _1,
3     seconds(30));
```

4 I'd like to highlight the call to `setAlarm` here as I did above, but there's no call to
5 highlight. Readers of this code simply have to know that calling `setSoundB` in-
6 volves `setAlarm` with the time and duration specified in the call to `std::bind`. To
7 the uninitiated, the placeholder "`_1`" is essentially magic, but even readers in the
8 know have to mentally map from the number in that placeholder to its position in
9 the `std::bind` parameter list in order to understand that the first argument in a
10 call to `setSoundB` is passed as the second argument to `setAlarm`. The type of this
11 argument is not identified in the call to `std::bind`, so readers have to consult the
12 `setAlarm` declaration to determine what kind of argument to pass to `setSoundB`.

13 If `setAlarm` is overloaded, the situation becomes more interesting. Suppose
14 there's an overload taking a fourth parameter specifying the alarm volume:

```
15 enum class Volume { Normal, Loud, LoudPlusPlus };
```

```
16 void setAlarm(Time t, Sound s, Duration d, Volume v);
```

17 The lambda continues to work as before, because overload resolution chooses the
18 three-argument version of `setAlarm`:

```
19 auto setSoundL =                                     // same as before
20     [](Sound s)
21 {
22     using namespace std::chrono;
23     setAlarm(steady_clock::now() + hours(1),      // fine, calls
24             s,                                     // 3-arg version
25             seconds(30));                      // of setAlarm
26 }
```

27 The `std::bind` call, on the other hand, now fails to compile:

```
28 auto setSoundB =
29     std::bind(setAlarm,                                // error! which
30             steady_clock::now() + hours(1),          // setAlarm?
31             _1,
32             seconds(30));
```

1 The problem is that compilers have no way to determine which of the two
2 `setAlarm` functions they should pass to `std::bind`. All they have is a function
3 name, and the name alone is ambiguous.

4 To get the `std::bind` call to compile, `setAlarm` must be cast to the proper func-
5 tion pointer type:

```
6 using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);
7 auto setSoundB =
8     std::bind(static_cast<SetAlarm3ParamType>(setAlarm), // now
9             steady_clock::now() + hours(1),
10            _1,
11            seconds(30));
```

12 But this brings up another difference between lambdas and `std::bind`. Inside the
13 function call operator for `setSoundL` (i.e., the function call operator of the lamb-
14 da's closure class), the call to `setAlarm` is a normal function invocation that can be
15 inlined by compilers in the usual fashion. Furthermore, C++ specifies that the func-
16 tion call operator on the closure class resulting from the lambda is `inline`, so in a
17 call to the lambda, it's entirely possible that the body of `setAlarm` is inlined at the
18 call site:

```
19 setSoundL(Sound::Siren);           // body of setAlarm may
20                           // well be inlined here
```

21 The call to `std::bind`, however, passes a function pointer to `setAlarm`, and that
22 means that inside the function call operator for `setSoundB` (i.e., the function call
23 operator for the class of the object returned by `std::bind`), the call to `setAlarm`
24 takes place through a function pointer. Compilers are much less likely to inline
25 function calls through function pointers, and that means that calls to `setAlarm`
26 through `setSoundB` are less likely to be fully inlined than those through
27 `setSoundL`:

```
28 setSoundB(Sound::Siren);           // body of setAlarm is less
29                           // likely to be inlined here
```

30 It's thus probable that using lambdas generates faster code than using `std::bind`.

31 The `setAlarm` example involves only a simple function call. If you want to do any-
32 thing more complicated, the scales tip even further in favor of lambdas. For exam-

1 ple, consider this C++14 lambda, which returns whether its argument is between a
2 minimum value (`lowVal`) and a maximal value (`highVal`), where `lowVal` and
3 `highVal` are local variables:

```
4 auto betweenL =
5     [lowVal, highVal]
6     (const auto& val) // C++14 only
7     { return lowVal <= val && val <= highVal; };
```

8 `std::bind` can express the same thing, but the construct is an example of job se-
9 curity through code obscurity:

```
10 using namespace std::placeholders; // as above
11 auto betweenB =
12     std::bind(std::logical_and<>(),
13                 std::bind(std::less_equal<>(), _1, lowVal),
14                 std::bind(std::less_equal<>(), _1, highVal));
```

15 Within this use of `std::bind`, I'm taking advantage of the C++14 feature of not
16 having to specify a template type argument for the standard operator templates
17 (e.g., `std::logical_and`, `std::less_equal`, etc.). In C++11, we'd have to specify
18 the types we wanted to compare, and the `std::bind` call would look like this:

```
19 using namespace std::placeholders;
20 auto betweenB = // C++11 version
21     std::bind(std::logical_and<bool>(),
22                 std::bind(std::greater_equal<int>(), _1, lowVal),
23                 std::bind(std::less_equal<int>(), _1, highVal));
```

24 Of course, in C++11, the lambda couldn't take an `auto` parameter, so it'd have to
25 commit to a type, too:

```
26 auto betweenL = // C++11 version
27     [lowVal, highVal]
28     (int x)
29     { return x >= lowVal && x <= highVal; };
```

30 Either way, I hope we can agree that the lambda version is not just shorter, but
31 also more comprehensible and maintainable.

32 Earlier, I remarked that for those with little `std::bind` experience, its placehold-
33 ers (e.g., `_1`, `_2`, etc.) are essentially magic. But it's not just the behavior of the

1 placeholders that's opaque. Suppose we have a function to create compressed cop-
2 ies of `Widget`,

```
3 enum class CompLevel { Low, Normal, High }; // compression
4                                         // level
5 Widget compress(const Widget& w,           // make compressed
6                  CompLevel lev);        // copy of w
```

7 and we want to create a function object that allows us to specify how much a par-
8 ticular `Widget` `w` should be compressed. This use of `std::bind` will create such an
9 object:

```
10 Widget w;
11 using namespace std::placeholders;
12 auto compressRateB = std::bind(compress, w, _1);
```

13 Now, when we pass `w` to `std::bind`, it has to be stored for the later call to `com-`
14 `press`. It's stored inside the object `compressRate`, but how is it stored—by value
15 or by reference? It makes a difference, because if `w` is modified between the call to
16 `std::bind` and a call to `compressRate`, storing `w` by reference will reflect the
17 changes, while storing it by value won't.

18 The answer is that it's stored by value, but the only way to know that is to memo-
19 rize how `std::bind` works; there's no sign of it in the call to `std::bind`.[†] Con-
20 trast that with a lambda approach, where whether `w` is captured by value or by ref-
21 erence is explicit:

```
22 auto compressRateL =                               // w is captured by
23   [w](CompLevel lev)                           // value; lev is
24   { return compress(w, lev); };                // passed by value
```

[†] `std::bind` always copies its arguments, but callers can achieve the effect of having an argument stored by reference by applying `std::ref` to it. The result of

```
auto compressRateB = std::bind(compress, std::ref(w), _1);
```

is that `compressRateB` acts as if it holds a reference to `w`, rather than a copy. A lambda, of course, would simply use an explicit by-reference capture mode to achieve this effect.

1 Equally explicit is how parameters are passed to the lambda. Here, it's clear that
2 the parameter `lev` is passed by value. Hence:

3 `compressRateL(CompLevel::High);` // arg is passed
4 // by value

5 But in the call to the object resulting from `std::bind`, how is the argument
6 passed?

7 `compressRateB(CompLevel::High);` // how is arg
8 // passed?

9 Again, the only way to know is to memorize how `std::bind` works. (The answer
10 is that all arguments passed to objects created by `std::bind` are passed by refer-
11 ence, because the function call operator for such objects uses perfect forwarding.)

12 Compared to lambdas, then, code using `std::bind` is less readable, less expres-
13 sive, and less efficient. In C++14, there are no reasonable use cases for `std::bind`.

14 In C++11, however, `std::bind` can be justified in two constrained situations:

15 • **Move capture.** C++11 lambdas don't offer move capture, but it can be emulat-
16 ed through a combination of a lambda and `std::bind`. For details, consult
17 Item 34, which also explains that in C++14, lambdas' support for init capture
18 eliminates the need for the emulation.

19 • **Polymorphic function objects.** Because the function call operator on the ob-
20 ject returned from `std::bind` uses perfect forwarding, it can accept argu-
21 ments of any type (modulo the restrictions on perfect forwarding described in
22 Item 32). This can be useful when you want to bind an object with a tem-
23 platized function call operator. For example, given this class,

```
24 class PolyWidget {  
25 public:  
26     template<typename T>  
27     void operator()(const T& param);  
28     ...  
29 };
```

30 `std::bind` can bind a `PolyWidget` as follows:

31 `PolyWidget pw;`

```
1 auto boundPW = std::bind(pw, _1);
2 boundPW can then be called with different types of arguments:
3 boundPW(1930);           // pass int to
4                                // PolyWidget::operator()
5 boundPW(nullptr);         // pass nullptr to
6                                // PolyWidget::operator()
7 boundPW("Rosebud");       // pass string literal to
8                                // PolyWidget::operator()
```

9 There is no way to do this with a C++11 lambda. In C++14, however, it's easily
10 achieved via a lambda with an `auto` parameter:

```
11 auto boundPW = [pw](const auto& param)    // C++14 only
12     { pw(param); };
```

13 These are edge cases, of course, and they're transient edge cases at that, because
14 compilers supporting C++14 lambdas are increasingly common.

15 When `bind` was unofficially added to C++ in 2005, it was a big improvement over
16 its 1998 predecessors. The addition of lambda support to C++11 rendered
17 `std::bind` all but obsolete, however, and as of C++14, there are just no use cases
18 for it.

19 Things to Remember

- 20 • Lambdas are more readable, more expressive, and more efficient than using
21 `std::bind`.
- 22 • In C++11 only, `std::bind` may be useful for implementing move capture or
23 for binding objects with templated function call operators.

1 **Chapter 7 The Concurrency API**

2 One of C++11's great triumphs is the incorporation of concurrency into the lan-
3 guage and library. Programmers familiar with other threading APIs (e.g., pthreads
4 or Windows' Threads) are sometimes surprised at the comparatively Spartan fea-
5 ture set that C++ offers, but that's because a great deal of C++'s support for concur-
6 rency is in the form of constraints on compiler-writers. The resulting language as-
7 surances mean that for the first time in C++'s history, programmers can write mul-
8 tithreaded programs with standard behavior across all computing platforms. This
9 establishes a solid foundation on which expressive libraries can be built, and the
10 concurrency elements of the Standard Library (tasks, futures, threads, mutexes,
11 condition variables, atomic objects, and more) are merely the beginning of what is
12 sure to become an increasingly rich set of tools for the development of concurrent
13 C++ software.

14 The existing features are impressive in their own right, of course, and this chapter
15 focuses on the questions that inform their effective application. What are the dif-
16 ferences between tasks and threads, and which should be used when? What be-
17 havioral guarantees does `std::async` offer, and how can they be controlled?
18 What are the implications of the varying behaviors of thread handle destructors?
19 What are the pros and cons of different inter-thread event communication strate-
20 gies? How do `std::atomics` differ from `volatiles`, and what is the proper appli-
21 cation of each? The coming pages address these issue, and more.

22 In the Items that follow, bear in mind that the Standard Library has two templates
23 for futures: `std::future` and `std::shared_future`. In many cases, the distinc-
24 tion is not important, so I often simply talk about *futures*, by which I mean both
25 kinds.

26 **Item 37: Prefer task-based programming to thread-based.**

27 If you want to run a function `doAsyncWork` asynchronously, you have two basic
28 choices. You can create a `std::thread` and run `doAsyncWork` on it, thus employ-
29 ing a *thread-based* approach:

```
1 int doAsyncWork();  
2 std::thread t(doAsyncWork);  
3 Or you can pass doAsyncWork to std::async, a strategy known as task-based:  
4 auto fut = std::async(doAsyncWork);           // "fut" for "future"  
5 In such calls, the callable entity passed to std::async (e.g., doAsyncWork) is con-  
6 sidered a task.  
7 The task-based approach is typically superior to its thread-based counterpart, and  
8 the tiny amount of code we've seen already demonstrates some reasons why. Here,  
9 doAsyncWork produces a return value, which we can reasonably assume the code  
10 invoking doAsyncWork is interested in. With the thread-based invocation, there's  
11 no straightforward way to get access to it. With the task-based approach, it's easy,  
12 because the future returned from std::async offers the get function. The get  
13 function is even more important if doAsyncWork emits an exception, because get  
14 provides access to that, too. With the thread-based approach, if doAsyncWork  
15 throws, the program dies (via a call to std::terminate or, if there was one, the  
16 function most recently specified via std::set_terminate).  
17 A more fundamental difference between thread-based and task-based program-  
18 ming is the higher level of abstraction that task-based embodies. It frees you from  
19 the details of thread management, an observation that reminds me that I need to  
20 summarize the three meanings of "thread" in concurrent C++ software:  
21 • Hardware threads are the threads that actually perform computation. Contem-  
22 porary machine architectures offer one or more hardware threads per CPU  
23 core.  
24 • Software threads (also known as OS threads or system threads) are the threads  
25 that the operating system† manages across all processes and schedules for ex-  
26 ecution on hardware threads. Typically, it's possible to create many more  
27 software threads than hardware threads, because when a software thread is
```

[†] Assuming you have one. Some embedded systems don't.

1 blocked (e.g., on I/O or waiting for a mutex or condition variable), throughput
2 can be improved by executing other, unblocked, threads.

3 • `std::threads` are objects in a C++ process that act as handles to underlying
4 software threads. Some `std::thread` objects represent “null” handles, i.e.,
5 correspond to no software thread, because they’re in a default-constructed
6 state (hence have no function to execute), have been moved from (the moved-
7 to `std::thread` then acts as the handle to the underlying software thread),
8 have been joined (the function they were to run has finished), or detached
9 (the connection between them and their underlying software thread has been
10 severed).

11 Software threads are a limited resource. If you try to create more than the system
12 can offer, a `std::system_error` exception is thrown. This is true even if the func-
13 tion you want to run can’t throw. For example, even if `doAsyncWork` is `noexcept`,

14

```
int doAsyncWork() noexcept; // see Item 16 for noexcept
```

15 this statement could result in an exception:

16

```
std::thread t(doAsyncWork); // throws if no more  
17                             // threads are available
```

18 Well-written software must somehow deal with this possibility, but how? One ap-
19 proach is to run `doAsyncWork` on the current thread, but that could lead to un-
20 balanced loads and, if the current thread is a GUI thread, responsiveness issues.
21 Another option is to wait for some existing software threads to complete and then
22 try to create a new `std::thread` again, but it’s possible that the existing threads
23 are waiting for an action that `doAsyncWork` is supposed to perform (e.g., produce
24 a result or notify a condition variable).

25 Even if you don’t run out of threads, you can have trouble with *oversubscription*.
26 That’s when there are more ready-to-run (i.e., unblocked) software threads than
27 hardware threads. When that happens, the thread scheduler (typically part of the
28 OS) time-slices the software threads on the hardware. When one thread’s time-
29 slice is finished and another’s begins, a context switch is performed. Such context
30 switches increase the overall thread management overhead of the system, and
31 they can be particularly costly when the hardware thread on which a software

1 thread is scheduled is on a different core than the software thread was for its last
2 time-slice. In that case, (1) the CPU caches are typically cold for that software
3 thread and (2) the running of the “new” software thread on that core “pollutes” the
4 CPU caches for “old” threads that had been running on that core and are likely to
5 be scheduled to run there again.

6 Avoiding oversubscription is difficult, because the optimal ratio of software to
7 hardware threads depends on how often the software threads are runnable, and
8 that can change dynamically, e.g., when a program goes from an I/O-heavy region
9 to a computation-heavy region. The best ratio of software to hardware threads is
10 also dependent on the cost of context switches and how effectively the software
11 threads use the CPU caches. Furthermore the number of hardware threads and the
12 details of the CPU caches (e.g., how large they are and their relative speeds) de-
13 pend on the machine architecture, so even if you tune your application to avoid
14 oversubscription (while still keeping the hardware busy) on one platform, there's
15 no guarantee that your solution will work well on other kinds of machines.

16 Your life will be easier if you dump these problems on somebody else, and using
17 `std::async` does exactly that.

```
18 auto fut = std::async(doAsyncWork); // onus of thread mgmt is
19 // on implementer of
20 // the Standard Library
```

21 This call shifts all the thread management responsibility to the implementers of
22 the C++ Standard Library. For example, you don't need to worry about an out-of-
23 threads exception arising, because this call will never yield one. “How can that
24 be?”, you might wonder. “If I ask for more software threads than the system can
25 provide, why does it matter whether I do it by creating `std::threads` or by call-
26 ing `std::async`?” It matters, because `std::async`, when called in this form (i.e.,
27 with the default launch policy—see Item 38), doesn't guarantee that it will create a
28 new software thread. Rather, it permits the scheduler to arrange for the specified
29 function (in this example, `doAsyncWork`) to be run on the current thread, and the
30 scheduler will take advantage of that freedom if the system is oversubscribed or is
31 out of threads.

1 If you pulled this “run it on the current thread” trick yourself, of course, I remarked
2 that it could lead to load-balancing or responsiveness issues, and those issues
3 don’t go away simply because it’s `std::async` and the runtime scheduler that
4 confront them instead of you. When it comes to load-balancing, however, the
5 runtime scheduler is likely to have a more comprehensive picture of what’s hap-
6 pening on the machine than you do, because it manages the threads from all pro-
7 cesses, not just the one your code is running in.

8 With `std::async`, responsiveness on a GUI thread can still be problematic, be-
9 cause the scheduler has no way of knowing which of your threads has tight re-
10 sponsiveness requirements. In that case, you’ll want to pass the
11 `std::launch::async` launch policy to `std::async`. That will ensure that the
12 function you want to run really executes on a different thread (see Item 38). How-
13 ever, you’ll have to be prepared for the possibility of an exception, because
14 `std::async` called with `std::launch::async` can run out of threads and emit a
15 `std::system_error` exception, just like the `std::thread` constructor can.

16 State-of-the-art thread schedulers employ system-wide thread pools to avoid
17 oversubscription, and they improve load balancing across hardware cores through
18 work-stealing algorithms. The C++ Standard does not require the use of thread
19 pools or work-stealing, but it’s deliberately written to permit them, and some ven-
20 dors already take advantage of this technology in their Standard Library imple-
21 ments. More are expected to follow. If you take a task-based approach to your
22 concurrent programming, you automatically reap the benefits of such technology
23 as it becomes more widespread. If, on the other hand, you program directly with
24 `std::threads`, you assume the burden of dealing with thread exhaustion, over-
25 subscription, and load balancing yourself, not to mention how your solutions to
26 these problems mesh with the solutions implemented in programs running in oth-
27 er processes on the same machine.

28 Compared to thread-based programming, a task-based design spares you the trav-
29 ails of manual thread management, and it provides a natural way to examine the
30 results of asynchronously executed functions (i.e., return values or exceptions).
31 Nevertheless, there are some situations where using threads directly may be ap-
32 propiate. They include:

- You need access to the API of the underlying threading implementation.

The C++ concurrency API is typically implemented using a lower-level platform-specific API, usually pthreads or Windows' Threads. Those APIs are currently richer than what C++ offers. (For example, C++ has no notion of thread priorities or affinities.) To provide access to the API of the underlying threading implementation, `std::thread` objects offer the `native_handle` member function. There is no counterpart to this functionality for `std::futures` (i.e., what `std::async` returns).

- You need to and are able to optimize thread usage for your application.

This could be the case, for example, if you're developing server software with a known execution profile that will be deployed as the only significant process on a machine with fixed hardware characteristics.

- You need to implement threading technology beyond the C++ concurrency API, e.g., thread pools on platforms where your C++ implementations don't offer them.

These are uncommon cases, however. Most of the time, you should choose task-based designs instead of programming with threads.

Things to Remember

- ◆ The `std::thread` API offer no direct way to get return values from asynchronously-run functions, and if those functions throw, the program is terminated.
 - ◆ Thread-based programming calls for manual management of thread exhaustion, oversubscription, load balancing, and adaptation to new platforms.
 - ◆ Task-based programming via `std::async` with the default launch policy suffers from none of these drawbacks.

Item 38: Specify `std::launch::async` if asynchronicity is essential.

When you call `std::async` to execute a function (or other callable entity), you're generally intending to run the function asynchronously. But that's not necessarily what you're asking `std::async` to do. You're really requesting that the function

1 be run in accord with a `std::async` *launch policy*. There are two standard policies, each represented by an enumerator in the `std::launch` scoped enum. (See
2 Item 10 for information on scoped enums.) Assuming a function `f` is passed to
3 `std::async` for execution,

- 5 • The `std::launch::async` launch policy means that `f` must be run asynchronously, i.e., on a different thread.
6
- 7 • The `std::launch::deferred` launch policy means that `f` may run only when
8 `get` or `wait` is called on the future returned by `std::async`.[†] That is, `f`'s exec-
9 `ution` is *deferred* until such a call is made. When `get` or `wait` is invoked, `f` will
10 execute synchronously, i.e., on the thread that made the invocation. (The caller
11 will block until `f` finishes running.) If neither `get` nor `wait` is ever called, `f` will
12 never run.

13 Perhaps surprisingly, `std::async`'s default launch policy—the one it uses if you
14 don't expressly specify one—is neither of these. Rather, it's these or-ed together.
15 The following two calls have exactly the same meaning:

```
16 auto fut1 = std::async(f);           // run f using
17                           // default launch
18                           // policy
19 auto fut2 = std::async(std::launch::async |    // run f either
20                       std::launch::deferred, // async or
21                       f);           // deferred
```

22 The default policy thus permits `f` to be run either asynchronously or synchronously. As Item 37 points out, this flexibility permits `std::async` and the thread-

[†]This is not rigorously true. What matters isn't the future on which `get` or `wait` is invoked, it's the shared state to which the future refers. (Item 40 discusses the relationship between futures and shared states.) Because `std::futures` support moving and can also be used to construct `std::shared_futures`, and because `std::shared_futures` can be copied, the future object referring to the shared state arising from the call to `std::async` to which `f` was passed is likely to be different from the one returned by `std::async`. That's a mouthful, however, so it's common to fudge the truth and simply talk about invoking `get` or `wait` on the future returned from `std::async`.

1 management components of the Standard Library to assume full responsibility for
2 thread creation and destruction, avoidance of oversubscription, and load balanc-
3 ing. That's among the things that make concurrent programming with
4 `std::async` so convenient.

5 But using `std::async` with the default launch policy has some noteworthy impli-
6 cations. Given a thread `t` executing this statement,

```
7 auto fut = std::async(f);           // run f using default
8                                         // launch policy
```

- 9 • It's not possible to predict whether `f` will run concurrently with `t`, because `f`
10 might be scheduled to run deferred.
- 11 • It's not possible to predict whether `f` runs on a thread different from `t`.
- 12 • It may not be possible to predict whether `f` runs at all, because it may not be
13 possible to guarantee that `get` or `wait` will be called on the future correspond-
14 ing to `f` along every path through the program.

15 The default launch policy's scheduling flexibility often mixes poorly with the use of
16 `thread_local` variables, because it means that if `f` reads or writes such *thread-*
17 *local storage* (TLS), it's not possible to predict which thread's variables will be ac-
18 cessed:

```
19 auto fut = std::async(f);           // TLS for f possibly for
20                                         // different thread, but
21                                         // possibly for current thread
```

22 It also complicates `wait`-based loops using timeouts, because calling `wait_for` or
23 `wait_until` on a task (see Item 37) that's deferred yields the value
24 `std::launch::deferred`. This means that the following loop, which looks like it
25 should eventually terminate, may, in reality, run forever:

```
26 void f()                         // f sleeps for
27 {                                 // 1 second,
28     std::this_thread::sleep_for(   // then returns
29         std::chrono::seconds(1)
30     );
31 }
```

```
1 auto fut = std::async(f); // run f
2 // asynchronously
3 // (conceptually)
4 while (fut.wait_for(
5     std::chrono::milliseconds(100)) != // loop until f
6     std::future_status::ready) // has finished
7 {
8     ... // running...
9 }
```

10 If `f` runs concurrently with the thread calling `std::async` (i.e., if the launch policy
11 chosen for `f` is `std::launch::async`), there's no problem here, but if `f` is de-
12 ferred, `fut.wait_for` will always return `std::future_status::deferred`.
13 That will never be equal to `std::future_status::ready`, so the loop will never
14 terminate.

15 This kind of bug is easy to overlook during development and unit testing, because
16 it's likely to manifest itself only under heavy loads. Those are the conditions that
17 push the machine towards oversubscription or thread exhaustion, and that's when
18 a task is most likely to be deferred. After all, if the hardware isn't threatened by
19 oversubscription or thread exhaustion, there's no reason for the runtime system
20 not to schedule the task for concurrent execution.

21 The fix is simple: just check the future corresponding to the `std::async` call to
22 see whether the task is deferred, and, if so, avoid entering the timeout-based loop.
23 Unfortunately, there's no direct way to ask a future whether its task is deferred.
24 Instead, you have to call a timeout-based function—a function such as `wait_for`.
25 Of course, you don't really want to wait for anything, you just want to see if the re-
26 turn value is `std::future_status::deferred`, so stifle your mild disbelief at
27 the necessary circumlocution and call `wait_for` with a zero timeout:

```
28 auto fut = std::async(f); // as above
29 if (fut.wait_for(std::chrono::seconds(0)) != // if task
30     std::future_status::deferred) // isn't
31 { // deferred...
32     while (fut.wait_for( // infinite loop
33         std::chrono::milliseconds(100)) != // no longer
34         std::future_status::ready) { // possible
```

```

1      ...                                // fut is neither deferred nor ready, so
2      ...                                // do concurrent work until it's ready
3  }
4      ...                                // fut is ready
5 } else {
6      ...                                // task is deferred, so use wait or
7      ...                                // get to call it synchronously
8 }

```

9 C++14 offers nothing to improve testing a future to see if it corresponds to a de-
10 ferred task, but it does make the specification of time durations more palatable,
11 because it takes advantage of C++11's support for user-defined literals to offer suf-
12 fixes for seconds (s), milliseconds (ms), hours (h), etc. These suffixes are imple-
13 mented in the `std::literals` namespace, so the above code can be rewritten as
14 follows:

```

15 using namespace std::literals;           // for duration suffixes
16 if (fut.wait_for(0s) != std::future_status::deferred) { // C++14
17     while (fut.wait_for(100ms) !=                         // only
18             std::future_status::ready) {
19     ...
20 }
21 ...                                // as before

```

22 The upshot of these various considerations is that using `std::async` with the de-
23 fault launch policy for a task is fine as long as the following conditions are fulfilled:

- 24 • Execution of the task need not run concurrently with the calling thread.
- 25 • It doesn't matter which thread's `thread_local` variables are read or written.
- 26 • There's either (1) a guarantee that `get` or `wait` will always be called on the
27 future returned by `std::async` or (2) it's acceptable that the task may never
28 be executed.
- 29 • Code using `wait_for` or `wait_until` always checks for deferred status.

1 If any of these conditions fails to hold, you probably want to guarantee that
2 `std::async` will schedule the task for truly asynchronous execution. The way to
3 do that is to pass `std::launch::async` as the first argument when you make the
4 call:

```
5 auto fut = std::async(std::launch::async, f); // launch f
6 // asynchronously
```

7 In fact, having a function that acts like `std::async`, but that automatically uses
8 `std::launch::async` as the launch policy, is a convenient tool to have around, so
9 it's nice that it's easy to write:

```
10 template<typename F, typename... Args>
11 inline
12 std::future<typename std::result_of<F(Args...)>::type>
13 reallyAsync(F&& f, Args&&... args) // return future
14 { // for asynchronous
15     return std::async(std::launch::async, // call to f(args...)
16                     std::forward<F>(f),
17                     std::forward<Args>(args)...);
18 }
```

19 This function takes an executable entity `f` and zero or more arguments `args` and
20 perfect-forwards them (see Item 27) to `std::async`, passing
21 `std::launch::async` as the launch policy. Like `std::async`, it returns a
22 `std::future` for the result of invoking `f` on `args`. Determining the type of that
23 result is easy, because the type trait `std::result_of` gives it to you. (See Item 9
24 for information on type traits.)

25 `reallyAsync` is used just like `std::sync`, except that, as Item 37 explains, it may
26 emit an exception indicating that it's not possible to create a new thread:

```
27 auto fut = reallyAsync(f); // run f asynchronously or
28 // throw std::system_error
```

29 In C++14, the ability to deduce `reallyAsync`'s return type streamlines the func-
30 tion declaration:

```
31 template<typename F, typename... Args>
32 inline
33 auto // C++14 only
34 reallyAsync(F&& f, Args&&... args)
35 {
```

```
1     return std::async(std::launch::async,
2                         std::forward<F>(f),
3                         std::forward<Args>(args)...);
4 }
```

5 This version makes it even clearer that `reallyAsync` does nothing but invoke
6 `std::async` with the `std::launch::async` launch policy.

7 **Things to Remember**

- 8 • The default launch policy for `std::async` permits both asynchronous and
9 synchronous task execution.
- 10 • This flexibility leads to uncertainty when accessing `thread_locals`, implies
11 that the task may never execute, and complicates program logic for timeout-
12 based `wait` calls.
- 13 • Specify `std::launch::async` if asynchronous task execution is essential.

14 **Item 39: Make `std::threads` unjoinable on all paths.**

15 Every `std::thread` object is in one of two states: *joinable* or *unjoinable*. A joina-
16 ble `std::thread` corresponds to an underlying asynchronous thread of execution
17 that is or could be running. A `std::thread` corresponding to an underlying
18 thread that's blocked or waiting to be scheduled is joinable, for example.

19 An unjoinable `std::thread` is what you'd expect: a `std::thread` that's not join-
20 able. Unjoinable `std::thread` objects include:

- 21 • **Default-constructed `std::threads`.** Such `std::threads` have no function to
22 execute, hence don't correspond to an underlying thread of execution.
- 23 • **`std::thread` objects that have been moved from.** The result of a move is
24 that the underlying thread of execution a `std::thread` used to correspond to
25 (if any) now corresponds to a different `std::thread`.
- 26 • **`std::threads` that have been joined.** After a `join`, a `std::thread`'s under-
27 lying thread of execution has finished running.

- 1 • **std::threads that have been detached.** A `detach` severs the connection
 2 between a `std::thread` object and the underlying thread of execution it cor-
 3 responds to.
- 4 One reason a `std::thread`'s joinability is important is that if the destructor for a
 5 joinable thread is invoked, execution of the program is terminated.
- 6 For example, suppose we have a function `dowork` that takes a filtering function,
 7 `filter`, and a maximum value, `maxVal`, as parameters. `dowork` checks to make
 8 sure that all conditions necessary for its computation are satisfied, then performs
 9 the computation with all the values between 0 and `maxVal` that pass the filter. If
 10 it's time-consuming to do the filtering and it's also time-consuming to determine
 11 whether `dowork`'s conditions are satisfied, it would be reasonable to do those two
 12 things concurrently. We might come up with code like this:

```

13  constexpr int tenMillion = 10000000;           // see Item 14
14                                // for constexpr
15  bool dowork(std::function<bool(int)> filter,   // returns whether
16              int maxVal = tenMillion)            // computation was
17  {                                              // performed
18      std::vector<int> vals;                     // values that
19                                // satisfy filter
20      std::thread t([&filter, maxVal, &vals]        // compute vals'
21                  {                           // content
22                      for (auto i = 0; i <= maxVal; ++i)
23                          { if (filter(i)) vals.push_back(i); }
24                  });
25
26      if (conditionsAreSatisfied()) {
27          t.join();                            // let t finish
28          performComputation(vals);          // computation was
29          return true;                      // performed
30      }
31      return false;                         // computation was
32  };

```

- 32 Before I explain why this code is problematic, I'll remark that `tenMillion`'s ini-
 33 tializing value can be made more readable in C++14 by taking advantage of
 34 C++14's ability to use an apostrophe as a digit separator:

1 `constexpr auto tenMillion = 10'000'000; // C++ 14 only`

2 But back to `doWork`. If `conditionsAreSatisfied()` returns `true`, all is well, but
3 if it returns `false` or throws an exception, the `std::thread` object `t` will be join-
4 able when its destructor is called at the end of `doWork`. That would cause program
5 execution to be terminated.

6 You might wonder why the `std::thread` destructor behaves this way. It's be-
7 cause the two other obvious options are arguably worse. They are:

8 • **An implicit join.** In this case, a `std::thread`'s destructor would wait for its
9 underlying asynchronous thread of execution to complete. That sounds rea-
10 sonable, but it could lead to performance anomalies that would be difficult to
11 track down. For example, it would be counterintuitive that `doWork` would wait
12 for its filter to be applied to all values if `conditionsAreSatisfied()` had al-
13 ready returned `false`.

14 • **An implicit detach.** In this case, a `std::thread`'s destructor would sever the
15 connection between the `std::thread` object and its underlying thread of exe-
16 cution. The underlying thread would continue to run. This sounds no less rea-
17 sonable than the `join` approach, but the debugging problems it can lead to are
18 actually worse. In `doWork`, for example, `vals` is a local stack variable that is
19 captured by reference. It's also modified inside the lambda (via the call to
20 `push_back`). Suppose, then, that while the lambda is running asynchronously,
21 `conditionsAreSatisfied()` returns `false`. In that case, `doWork` would re-
22 turn, and its local variables (including `vals`) would be destroyed. Its stack
23 frame would be popped, and execution of its thread would continue at
24 `doWork`'s call site.

25 Statements following that call site would, at some point, make additional func-
26 tion calls, and at least one such call would probably end up using some or all of
27 the memory that had once been occupied by the `doWork` stack frame. Let's call
28 such a function `f`. While `f` was running, the lambda that `doWork` initiated
29 would still be running asynchronously. That lambda could call `push_back` on
30 the stack memory that used to be `vals`, but that is now somewhere inside `f`'s
31 stack frame. Such a call would modify the memory that used to be `vals`, and

1 that means that from `f`'s perspective, the content of memory in its stack frame
2 could spontaneously change! Imagine the fun you'd have debugging *that*.

3 The Standardization Committee decided that the consequences of destroying a
4 joinable thread were sufficiently dire that they essentially banned it (by specifying
5 that destruction of a joinable thread causes program termination).

6 This puts the onus on you to ensure that if you use a `std::thread` object, it's
7 made unjoinable on every path out of the scope in which it's defined. But covering
8 every path can be complicated. It includes flowing off the end of the scope as well
9 as jumping out via a `return`, `continue`, `break`, `goto` or exception. That can be a
10 lot of paths.

11 Any time you want to perform some action along every path out of a scope, the
12 normal approach is to put that action in the destructor of a local object. Such ob-
13 jects are known as *RAII objects*, and the classes they come from are known as *RAII*
14 *classes*. (*RAII* itself stands for "resource acquisition is initialization," although the
15 crux of the technique is destruction, not initialization). RAII classes are common in
16 the Standard Library. Examples include the STL containers (each container's de-
17 structor destroy the container's contents and releases its memory), the standard
18 smart pointers (Items 20-22 explain that `std::unique_ptr`'s destructor invokes
19 its deleter on the object it points to, and the destructors in `std::shared_ptr` and
20 `std::weak_ptr` decrement reference counts), `std::fstream` objects (their de-
21 structors close the files they correspond to), and many more. And yet there is no
22 standard RAII class for `std::thread` objects—perhaps because the Standardiza-
23 tion Committee, having rejected both `join` and `detach` as default options, simply
24 didn't know what such a class should do.

25 Fortunately, it's not difficult to write one yourself. For example, the following class
26 allows callers to specify a `std::thread` member function (e.g., `join` or `detach`)
27 that should be called when a `ThreadRAII` object (an RAII object for a
28 `std::thread`) is destroyed:

```
29 class ThreadRAII {  
30 public:  
31     using RAIIAction =           // type of mbr func to  
32         void (std::thread::*())(); // invoke in ThreadRAII dtor
```

```

1   ThreadRAII(std::thread&& t, RAIIAction a) // in dtor,
2     : action(a), t(std::move(t)) {}           // invoke a on t
3
4   ~ThreadRAII()                           // see below for
5     { if (t.joinable()) (t.*action)(); }    // joinability test
6
7   std::thread& get() { return t; }         // see below
8
9 };

```

10 With any luck, this code is largely self-explanatory, but the following points may be
11 helpful:

- 12 • The constructor accepts only `std::thread` rvalues, because we want to move
13 the passed-in `std::thread` into the `ThreadRAII` object.
- 14 • The parameter order in the constructor is designed to be intuitive to callers
15 (specifying the `std::thread` first and the destructor action second makes
16 more sense than vice-versa), but the member initialization list is designed to
17 match the order of the data members' declarations. That order puts the
18 `std::thread` object last. In this class, the order makes no difference, but in
19 general, it's possible for the initialization of one data member to depend on
20 another, and because `std::thread` objects may start running a function im-
21 mediately after they are initialized, it's a good habit to declare them last in a
22 class. That guarantees that at the time they are constructed, all the data mem-
23 bers that precede them have already been initialized and can therefore be safe-
24 ly accessed by the asynchronously running thread that corresponds to the
25 `std::thread` data member.
- 26 • `ThreadRAII` offers a `get` function to provide access to the underlying
27 `std::thread` object. This is analogous to the `get` functions offered by the
28 standard smart pointer classes that give access to their underlying raw point-
29 ers. Providing `get` avoids the need for `ThreadRAII` to replicate the full
30 `std::thread` interface, and it also means that `ThreadRAII` objects can be
31 used in contexts where `std::thread` objects are required, e.g., because an in-
32 terface requires that a reference to a `std::thread` be passed.

- 1 • Before the `ThreadRAII` destructor invokes the action `a` on the `std::thread`
2 object `t`, it checks to make sure that `t` is joinable. This is necessary, because in-
3 voking `join` or `detach` on an unjoinable thread yields undefined behavior,
4 and it's possible that a client constructed a `std::thread`, created a `Thread-`
5 `RAII` object from it, used `get` to acquire access to `t`, and then did a move from
6 `t` or called `join` or `detach` on it. Each of those actions would render `t` unjoin-
7 able.

8 If you're worried that in the statement,

```
9   if (t.joinable()) (t.*action)();
```

10 a race condition exists, because between execution of `t.joinable()` and
11 `(t.*action)()`, another thread could render `t` unjoinable, your intuition is
12 commendable, but your fears are unfounded. A `std::thread` object can
13 change state from joinable to unjoinable only through a member function call,
14 e.g., `join`, `detach`, or a move operation. At the time a `ThreadRAII` object's de-
15 structor is invoked, no other thread should be making member function calls
16 on that object. If there are simultaneous calls, there is certainly a race condi-
17 tion, but the race isn't inside the destructor, it's in the client code that is trying
18 to invoke two member functions (the destructor and something else) on one
19 object at the same time. In general, simultaneous member function calls on a
20 single object are safe only if all are to `const` member functions (see Item 15).

21 Employing `ThreadRAII` in our `doWork` example would look like this:

```
22   bool doWork(std::function<bool(int)> filter, // as before  
23                int maxVal = tenMillion)  
24   {  
25     std::vector<int> vals;                          // as before  
26     ThreadRAII t(  
27       std::thread([&filter, maxVal, &vals]      // use RAII object  
28        {  
29           for (auto i = 0; i <= maxVal; ++i)  
30            { if (filter(i)) vals.push_back(i); }  
31        }),  
32       &std::thread::join                                // RAII action  
33     );
```

```

1  if (conditionsAreSatisfied()) {
2      t.get().join();                                // let t finish
3      performComputation(vals);
4      return true;
5  }
6
7  return false;                                    // join thread
8};                                              // running lambda

```

9 In this case, we've chosen to do a `join` on the asynchronously running thread in
10 the `ThreadRAII` destructor, because, as we saw earlier, doing a `detach` could lead
11 to some truly nightmarish debugging. We also saw earlier that doing a `join` could
12 lead to performance anomalies (that, to be frank, could also be unpleasant to de-
13 bug), but given a choice between undefined behavior (which `detach` would get
14 us), program termination (which use of a raw `std::thread` would yield), or per-
15 formance anomalies, performance anomalies seems like the best of a bad lot.

16 Alas, Item 41 demonstrates that using `ThreadRAII` to perform a `join` on
17 `std::thread` destruction can sometimes lead not just to a performance anomaly,
18 but to a hung program. The “proper” solution to these kinds of problems would be
19 to communicate to the asynchronously running lambda that we no longer need its
20 work and that it should return early, but there's no support in C++11 or C++14 for
21 *interruptible threads*. They can be implemented by hand, but that's a topic beyond
22 the scope of this book.[†]

23 Item 19 explains that because `ThreadRAII` declares a destructor, there will be no
24 compiler-generated move operations, but there is no reason `ThreadRAII` objects
25 shouldn't be movable. If compilers were to generate these functions, the functions
26 would do the right thing, so explicitly requesting their creation is appropriate:

```

27 class ThreadRAII {
28 public:
29     using RAIIAction = void (std::thread::*())();           // as before

```

[†] A nice treatment of this topic is in Anthony Williams' *C++ Concurrency in Action* (Manning Publications, 2012), section 9.2.

```

1   ThreadRAII(std::thread&& thread, RAIIAction a)      // as before
2   : action(a), t(std::move(thread)) {}
3
4   ~ThreadRAII()                                         // as before
5   { if (t.joinable()) (t.*action)(); }
6
7   ThreadRAII(ThreadRAII&&) = default;                // support
8   ThreadRAII& operator=(ThreadRAII&&);               // moving
9
10  std::thread& get() { return t; }
11
12  private:                                              // as before
13    RAIIAction action;
14    std::thread t;
15
16 A class like ThreadRAII helps avoid early program termination due to destruction
17 of joinable std::threads, but the drawbacks associated with join-on-
18 destruction and detach-on-destruction may convince you that a better solution is
19 to follow the advice of Item 37 and stay away from std::threads in the first
20 place. Doing that would allow you to deal with futures instead of std::threads,
21 and futures don't invoke std::terminate in their destructors. What they do in
22 their destructors is sufficiently interesting, however, that it's worth an Item of its
23 own. That Items is Item 40. It's coming right up.

```

20 Things to Remember

- 21 • Make std::threads unjoinable on all paths.
- 22 • join-on-destruction can lead to difficult-to-debug performance anomalies.
- 23 • detach-on-destruction can lead to difficult-to-debug undefined behavior.

24 Item 40: Be aware of varying thread handle destructor behavior.

25 Item 39 explains that a joinable std::thread corresponds to an underlying system thread of execution. A future for a non-deferred task (see Item 38) has a similar relationship to a system thread. As such, both std::thread objects and future objects can be thought of as *handles* to system threads.

30 From this perspective, it's interesting that std::threads and futures have such different behaviors in their destructors. As noted in Item 39, destruction of a join-

1 able `std::thread` terminates your program, because the two obvious alterna-
2 tives—an implicit `join` and an implicit `detach`—were considered worse choices.
3 Yet destruction of a future sometimes performs an implicit `join`, sometimes it per-
4 forms an implicit `detach`, and sometimes it does neither. It never causes program
5 termination.

6 This thread handle behavioral bouillabaisse deserves closer examination.

7 We'll begin with the observation that a future is one end of a communications
8 channel through which a callee transmits a result to a caller.[†] The callee (running
9 asynchronously, except in the case of deferred tasks) writes the result of its com-
10 putation into the communications channel (typically via a `std::promise` object),
11 and the caller reads that result using a future. You can think of it as follows, where
12 the dashed arrow shows the flow of information from callee to caller:



13

14 But where is the callee's result stored? The callee could finish before the caller in-
15 vokes `get` on a corresponding future, so the result can't be stored in the callee's
16 `std::promise`. That object, being local to the callee, would be destroyed when the
17 callee finished.

18 The result can't be stored in the caller's future, either, because (among other rea-
19 sons) a `std::future` may be used to create a `std::shared_future` (thus trans-
20 ferring ownership of the callee's result from the `std::future` to the
21 `std::shared_future`), which may then be copied many times after the original
22 `std::future` is destroyed. Given that not all result types can be copied (e.g.,
23 `std::unique_ptr`—see Item 20) and that the result must live at least as long as
24 the last future referring to it, which of the potentially many futures corresponding
25 to the callee would be the one to contain its result?

[†] Item 41 explains that the kind of communications channel associated with a future can be employed for other purposes. For this Item, however, we'll consider only its use as a mechanism for a callee to convey its result to a caller.

1 Because neither objects associated with the callee nor objects associated with the
2 caller are suitable places to store the callee's result, it's stored in a location outside
3 both. This location is known as the *shared state*. The shared state is typically rep-
4 resented by a heap-based object, but its type, interface, and implementation are
5 not specified by the Standard. Standard Library authors are free to implement
6 shared states in any way they like.

7 We can envision the relationship among the callee, the caller, and the shared state
8 like this, where dashed arrows once again represent the flow of information:



9

10 The existence of the shared state is important, because the behavior of a future's
11 destructor—the topic of this Item—is determined by the shared state associated
12 with the future. In particular,

- 13 • **The destructor for the last future referring to a shared state for a non-**
14 **deferred task launched via std::async blocks** until the task completes. In
15 essence, the destructor for such a future does an implicit join on the thread
16 on which the asynchronously-executing task is running.
- 17 • **The destructor for all other futures simply destroys the future object.** For
18 asynchronously running tasks, this amounts to an implicit detach on the un-
19 derlying thread. For deferred tasks for which this is the final future, it means
20 that the deferred task will never run.

21 These rules sound more complicated than they are. What we're really dealing with
22 is a simple “normal” behavior and one lone exception to it. The “normal” behavior
23 is that a future's destructor destroys the future object. That's it. It doesn't join
24 with anything, it doesn't detach from anything, it doesn't run anything. It just de-
25 stroys the future's data members. (Well, okay, it does one more thing. It also de-
26 crements the reference count inside the shared state that's manipulated by both the
27 futures referring to it and the callee's std::promise. This reference count makes

1 it possible for the library to know when the shared state can be destroyed. For in-
2 formation about reference counting, see Item 21.)

3 The exception to this normal behavior arises only for a future for which all of the
4 following apply:

- 5 • **It refers to a shared state that was created due to a call to `std::async`.**
- 6 • **The task's launch policy is `std::launch::async`** (see Item 38), either be-
7 cause that was chosen by the runtime system or because it was specified in the
8 call to `std::async`.
- 9 • **The future is the last future referring to the shared state.** For
10 `std::futures`, this will always be the case. For `std::shared_futures`, how-
11 ever, if other `std::shared_futures` refer to the same shared state as the fu-
12 ture being destroyed, the future being destroyed follows the “normal” behavior
13 (i.e., it simply destroys its data members).

14 Only when all of these conditions are fulfilled does a future’s destructor exhibit
15 special behavior, and that behavior is to block until the asynchronously running
16 task completes. Practically speaking, this amounts to an implicit `join` on the
17 thread running the `std::async`-created task.

18 It’s not uncommon to hear this exception to “normal” future destructor behavior
19 summarized as “Futures from `std::async` block in their destructors.” To a first
20 approximation, that’s correct, but sometimes you need more than a first approxi-
21 mation. Now you know the truth in all its glory and wonder.

22 What you may wonder is why there’s a special rule for shared states for non-
23 deferred tasks that are launched by `std::async`. It’s a reasonable question. From
24 what I can tell, the Standardization Committee wanted to avoid the problems asso-
25 ciated with an implicit `detach` (see Item 39), but they didn’t want to adopt as radi-
26 cal a policy as mandatory program termination (as they did for joinable
27 `std::threads`—again, see Item 39), so they compromised on an implicit `join`.
28 The decision was not without controversy, and there was serious talk about aban-

1 doing this behavior for C++14. In the end, however, no change was made, so the
2 behavior of destructors for futures is consistent in C++11 and C++14.

3 The API for futures offers no way to determine whether a future refers to a shared
4 state arising from a call to `std::async`, so given a random future object, it's not
5 possible to know whether it will block in its destructor waiting for an asynchro-
6 nously running task to finish. This has some interesting implications:

```
7 // this container might block in its dtor, because one or more
8 // contained futures could refer to shared state for a non-
9 // deferred task launched via std::async
10 std::vector<std::future<void>> futs; // see Item 41 for info
11 // on std::future<void>
12 class Widget { // Widget objects might
13 public: // block in their dtors
14 ...
15 private:
16     std::shared_future<double> fut;
17 };
18 void doWork(std::future<int> fut); // fut might block
19 // in its dtor
20 // (see Item 17 for info
21 // on by-value params)
```

22 Of course, if you have a way of knowing that a given future *does not* satisfy the
23 conditions that trigger the special destructor behavior (e.g., due to program logic),
24 you're assured that that future won't block in its destructor. For example, only
25 shared states arising from calls to `std::async` qualify for the special behavior, but
26 there are other ways that shared states get created. One is the use of
27 `std::packaged_task`. A `std::packaged_task` object prepares a function (or
28 other callable entity) for asynchronous execution by wrapping it such that its re-
29 sult is put into a shared state. A future referring to that shared state can then be
30 obtained via `std::packaged_task`'s `get_future` function:

```
31 int calcValue(); // func to run
32 std::packaged_task<int()> pt(calcValue()); // wrap calcValue so it
33 // can run asynchronously
34 auto ptFut = pt.get_future(); // get future for pt
```

1 Once created, the `std::packaged_task` `pt` can be run on a thread. (It could be run
2 via a call to `std::async`, too, but if you want to run a task using `std::async`,
3 there's little reason to create a `std::packaged_task`, because `std::async` does
4 everything `std::packaged_task` does before it schedules the task for execution.)

5 `std::packaged_tasks` aren't copyable, so when `pt` is passed to the
6 `std::thread` constructor, it must be cast to an rvalue (via `std::move`—see
7 Item 25), thus ensuring that it will be moved into the data area associated with the
8 thread, not copied:

9 `std::thread t(std::move(pt));` // run pt on t

10 At this point, we know that the future `ptFut` doesn't refer to a shared state created
11 by a call to `std::async`, so its destructor will behave normally. We can therefore
12 call the `doWork` function (declared above) without fear that when `doWork`'s by-
13 value parameter `fut` is destroyed, it will block waiting for `t` to finish.

14 `std::packaged_task::get_future` returns a `std::future` object, which is a
15 move-only type, so for the call to `doWork` to compile, we must apply `std::move` to
16 `ptFut`, just as we did when passing `pt` to the `std::thread` constructor:

17 `doWork(std::move(ptFut));` // doWork won't block on t

18 This example actually lends some insight into the normal behavior for future de-
19 structors, but it's easier to see if the statements are put together inside a scope:

20 { // begin scope
21 `std::packaged_task<int()>` // as above
22 `pt(calcValue);`
23 `auto ptFut = pt.get_future();` // as above
24 `std::thread t(std::move(pt));` // as above
25 `doWork(std::move(ptFut));` // as above
26 ... // see below
27 }

1 The most interesting code here is the “...” that follows the call to `doWork` and pre-
2 cedes the end of the scope. What makes it interesting is what can happen to the
3 `std::thread` object `t` inside the “...” region. There are three basic possibilities:

- **Nothing happens to t.** In this case, `t` will be joinable at the end of the scope.
That will cause the program to be terminated (see Item 39).

- **A join is done on t.** In this case, there would be no need for `ptFut` (or the
parameter `fut`) to block in its destructor, because the `join` call is already pre-
sent in the calling code.

- **A detach is done on t.** In this case, there would be no need for `ptFut` (or
`fut`) to detach in its destructor, because the calling code already does that.

11 In other words, when you have a future corresponding to a shared state that arose
12 due to a `std::packaged_task`, there's usually no need to adopt a special destruc-
13 tion policy, because the decision among termination, joining, or detaching will be
14 made in the code that manipulates the `std::thread` on which the
15 `std::packaged_task` is typically run.

16 **Things to Remember**

- Future destructors normally just destroy the future's data members.
- The final future referring to a shared state for a non-deferred task launched via
`std::async` blocks until the task completes.

20 **Item 41: Consider void futures for one-shot event commu- 21 nication.**

22 Sometimes it's useful for a task to tell a second (asynchronously-running) task that
23 a particular event has occurred, because the second task can't proceed until the
24 event has taken place. Perhaps a data structure has been initialized, a stage of
25 computation has been completed, or a significant sensor value has been detected.
26 When that's the case, what's the best way for this kind of inter-thread communica-
27 tion to take place?

1 An obvious approach is to use a condition variable. If we call the task that detects
2 the condition the *detecting task* and the task reacting to the condition the *reacting*
3 *task*, the strategy is simple: the reacting task waits on a condition variable
4 (*condvar*), and the detecting thread notifies that condvar when the event occurs.

5 Given

6 `std::condition_variable cv;` // condvar for event
7 `std::mutex m;` // mutex for use with cv

8 the code in the detecting task is as simple as simple can be:

9 `...` // detect event
10 `cv.notify_one();` // tell reacting task

11 If there were multiple reacting tasks to be notified, it would be appropriate to re-
12 place `notify_one` with `notify_all`, but for now, we'll assume there's only one
13 reacting task.

14 The code for the reacting task is a bit more complicated, because before calling
15 `wait` on the condvar, it must lock the mutex through a `std::unique_lock` object.
16 (Locking a mutex before waiting on a condition variable is typical for threading
17 libraries. The need to lock the mutex through a `std::unique_lock` object is
18 simply part of the C++11 API.) Here's the conceptual approach, although by this
19 point in this book, you probably suspect that when I call code "conceptual," it's not
20 really correct. Your suspicion is well-founded, but set that aside for a moment:

21 `...` // prepare to react
22 `{` // open critical section
23 `std::unique_lock<std::mutex> lk(m);` // lock mutex
24 `cv.wait(lk);` // wait for notify;
25 // this isn't correct!
26 `...` // react to event
27 // (mutex is locked)
28 `}` // close crit. section;
29 // unlock mutex via
30 // lk's dtor

```
1   ...                                // continue reacting
2                               // (mutex now unlocked)
```

3 The first issue with this code is what's sometimes termed a *code smell*: even if the
4 code works, something doesn't seem quite right. In this case, the odor emanates
5 from the need to use a mutex. Mutexes are used to control access to shared data,
6 but it's entirely possible that the detecting and reacting tasks have no need for
7 such mediation. For example, the detecting task might be responsible for initializing
8 a global data structure, then turning it over to the reacting task for use. If the
9 detecting task never accesses the data structure after initializing it, and if the re-
10 acting task never accesses it before the detecting task indicates that it's ready, the
11 two tasks will stay out of each other's way through program logic. There will be no
12 need for a mutex. The fact that the condvar approach requires one leaves behind
13 the unsettling aroma of suspect design.

14 Even if you look past that, there are two other problems you should definitely pay
15 attention to:

- 16 • **If the detecting task notifies the condvar before the reacting task waits, the reacting task will hang.** In order for notification of a condvar to wake another task, the other task must be waiting on that condvar. If the detecting task happens to execute the notification before the reacting task executes the `wait`, the reacting task will miss the notification, and it will wait forever.
- 21 • **The `wait` statement fails to account for spurious wakeups.** A fact of life in threading APIs (in many languages—not just C++) is that code waiting on a condition variable may be awakened even if the condvar wasn't notified. Such awakenings are known as *spurious wakeups*. Proper code deals with them by confirming that the condition being waited for has actually occurred, and it does this as its first action after waking. The C++ condvar API makes this exceptionally easy, because it permits a lambda (or other callable entity) that tests for the waited-for condition to be passed to `wait`. That is, the `wait` call in the reacting task could be written like this:

```
30   cv.wait(lk,
31           []{ return whether the event has occurred; });
```

1 Taking advantage of this capability requires that the reacting task be able to
2 determine whether the condition it's waiting for is true. But in the scenario
3 we've been considering, the condition it's waiting for is the occurrence of an
4 event that the detecting thread is responsible for recognizing. The reacting
5 thread may have no way of determining whether the event it's waiting for has
6 taken place. That's why it's waiting on a condition variable!

7 There are many situations where having tasks communicate using a condvar is an
8 good fit for the problem at hand, but this doesn't seem to be one of them.

9 For many developers, the next trick in their bag is a shared boolean flag. The flag is
10 initially `false`. When the detecting thread recognizes the event of interest, it sets
11 the flag:

```
12 std::atomic<bool> flag(false);      // shared flag; see
13                               // Item 42 for std::atomic
14
15 ...                           // detect event
16 flag = true;                  // tell reacting task
```

17 For its part, the reacting thread simply polls the flag. When it sees that the flag is
18 set, it knows that the event it's been waiting for has occurred:

```
19 ...                           // prepare to react
20 while (!flag);                // wait for event
21 ...                           // react to event
```

22 This approach suffers from none of the drawbacks of the condvar-based design.
23 There's no need for a mutex, no problem if the detecting task sets the flag before
24 the reacting task starts polling, and nothing akin to a spurious wakeup. Good, good,
25 good.

26 Less good is the cost of polling in the reacting task. During the time the task is
27 waiting for the flag to be set, the task is essentially blocked, yet it's still running. As
28 such, it occupies a hardware thread that another task might be able to make use of,
29 it incurs the cost of a context switch each time it starts or completes its timeslice,
30 and it could keep a core running that might otherwise be shut down to save power.

1 A truly blocked task would do none of these things. That's an advantage of the
2 condvar-based approach, because a task in a `wait` call is truly blocked.

3 But there's another way to block a task—one that doesn't suffer from the problems
4 associated with condition variables. It's to have the reacting task `wait` on a future
5 that's set by the detecting task. This may seem like an odd idea. After all, Item 40
6 explains that a future represents the receiving end of a communications channel
7 from a callee to a (typically asynchronous) caller, and here there's no callee-caller
8 relationship between the detecting and reacting tasks. However, Item 40 also
9 notes that a communications channel whose transmitting end is a `std::promise`
10 and whose receiving end is a future can be used for more than just callee-caller
11 communication. Such a communications channel can be used in any situation
12 where you need to transmit information from one place in your program to another.
13 In this case, we'll use it to transmit information from the detecting task to the
14 reacting task, and the information we'll convey will be that the event of interest
15 has taken place.

16 The design is simple. The detecting task has a `std::promise` object (i.e., the writing
17 end of the communications channel), and the reacting task has a corresponding
18 future. When the detecting task sees that the event of interest has occurred, it sets
19 the `std::promise` (i.e., write into the communications channel). Meanwhile, the
20 reacting task `waits` on its future. That `wait` blocks the reacting task until the
21 `std::promise` has been set.

22 Now, both `std::promise` and futures (i.e., `std::future` and
23 `std::shared_future`) are templates that require a type parameter. That parameter
24 indicates the type of data to be transmitted through the communications
25 channel. In our case, however, there's no data to be conveyed. The only thing of
26 interest to the reacting task is that its future has been set (via the corresponding
27 `std::promise`). What we need for the `std::promise` and future templates is a
28 type that indicates that no data is to be conveyed across the communications
29 channel. That type is `void`. The detecting task will thus use a
30 `std::promise<void>`, and the reacting task a `std::future<void>` or
31 `std::shared_future<void>`. The detecting task will set its
32 `std::promise<void>` when the event of interest occurs, and the reacting task

1 will `wait` on its future. Even though the reacting task won't receive any data from
2 the detecting task, the communications channel will permit the reacting task to
3 know when the detecting task has "written" its `void` data by calling `set_value` on
4 its `std::promise`.

5 So given

6 `std::promise<void> p;` // promise for
7 // communications channel

8 the detecting task's code is trivial,

9 `...` // detect event

10 `p.set_value();` // tell reacting task

11 and the reacting task's code is equally simple:

12 `...` // prepare to react

13 `p.get_future().wait();` // wait on future
14 // corresponding to p

15 `...` // react to event

16 Like the approach using a flag, this design requires no mutex, works regardless of
17 whether the detecting task sets its `std::promise` before the reacting task `waits`,
18 and is immune to spurious wakeups. (Only condition variables are susceptible to
19 that problem.) Like the condvar-based approach, the reacting task is truly blocked
20 after making the `wait` call, so it consumes no system resources while waiting. Per-
21 fect, right?

22 Not exactly. Sure, a future-based approach skirts those shoals, but there are other
23 hazards to worry about. For example, Item 40 explains that between a
24 `std::promise` and a future is a shared state, and shared states are typically dy-
25 namically allocated. You should therefore assume that this design incurs the cost
26 of heap-based memory allocation and deallocation.

27 Perhaps more importantly, a `std::promise` may be set only once. The communi-
28 cations channel between a `std::promise` and a future is a *one-shot* mechanism: it
29 can't be used repeatedly. This is a notable difference from the condvar- and flag-

1 based designs, both of which can be used to communicate multiple times. (A
2 condvar can be repeatedly notified, and a flag can always be cleared and set again.)

3 The one-shot restriction isn't as limiting as you might think. Suppose you'd like to
4 create a system thread in a suspended state. That is, you'd like to get all the over-
5 head associated with thread creation out of the way, so that when you're ready to
6 execute something on the thread, the normal thread-creation latency will be
7 avoided. Or you might want to create a suspended thread so that you could config-
8 ure it before letting it run. Such configuration might include things like setting its
9 priority or core affinity. The C++ concurrency API offers no way to do those things,
10 but `std::thread` objects offer the `native_handle` member function, the result
11 of which is intended to give you access to the platform's underlying threading API
12 (usually POSIX threads or Windows threads). The lower-level API often does make
13 it possible to configure thread characteristics such as priority and affinity.

14 Assuming you want to suspend a thread only once (after creation, but before it's
15 running its thread function), a design using a `void` future is a reasonable choice.
16 Here's the essence of the technique:

```
17 std::promise<void> p;  
18 void react(); // func for reacting task  
19 void detect(); // func for detecting task  
20 {  
21     std::thread t([]) // create thread  
22     {  
23         p.get_future().wait(); // suspend t until  
24         react(); // future is set  
25     };  
26     ... // here, t is suspended  
27     // prior to call to react  
28     p.set_value(); // unsuspend t (and thus  
29     // call reset)  
30     ... // do additional work  
31     t.join(); // make t unjoinable  
32 };
```

1 Because it's important that `t` become unjoinable on all paths out of `detect`, use of
2 an RAII class like Item 39's `ThreadRAII` seems like it would be advisable. Code
3 like this comes to mind:

```
4 void detect()
5 {
6     ThreadRAII tr(                                     // use RAII object
7         std::thread([])
8         {
9             p.get_future().wait();
10            react();
11        },
12        &std::thread::join                                // risky! (see below)
13    );
14 ...
15 p.set_value();                                    // unsuspend t
16 ...
17 };
```

18 This looks safer than it is. The problem is that if in the first "..." region (the one with
19 the "t's suspended here" comment), an exception is emitted, `set_value` will never
20 be called on `p`. That means that the call to `wait` inside the lambda that's running on
21 `t` will never return. That, in turn, means that `t` will never finish running, and that's
22 a problem, because the RAII object `tr` has been configured to perform a `join` on `t`
23 in `tr`'s destructor. In other words, if an exception is emitted from the first "..." re-
24 gion of code, this function will hang, because `tr`'s destructor will never complete.

25 There are ways to address this problem, but I'll leave them in the form of the hal-
26 lowed exercise for the reader.[†] Here, I'd like to show how the original code (i.e., not
27 using `ThreadRAII`) can be extended to suspend and then unsuspend not just one
28 reacting task, but many. It's a simple generalization, because the key is to use
29 `std::shared_futures` instead of a `std::future` in the `react` code. Once you

[†] A reasonable place to begin researching the matter is the 24 December 2013 blog post,
["ThreadRAII + Thread Suspension = Trouble?"](#)

1 know that the `std::future`'s `share` member function transfers ownership of its
2 shared state to the `std::shared_future` object produced by `share`, the code
3 nearly writes itself. The only subtlety is that each reacting thread needs its own
4 copy of the `std::shared_future` that refers to the shared state, so the
5 `std::shared_future` obtained from `share` is captured by value by the lambdas
6 running on the reacting threads:

```
7 std::promise<void> p;                                // as before
8 void detect()                                         // now for multiple
9 {                                                       // reacting tasks
10    auto sf = p.get_future().share(); // sf's type is
11    // std::shared_future<void>
12    std::vector<std::thread> vt;           // container for
13    // reacting threads
14    for (int i = 0; i < threadsToRun; ++i) {
15        vt.emplace_back([sf]{ sf.wait();          // wait on local
16                               react();}); // copy of sf; see
17    }                                                 // Item 18 for info
18                                              // on emplace_back
19    ...                                            // detect hangs if
20    // this "..." code throws!
21    p.set_value();                                // unsuspend all threads
22    ...
23    for (auto& t : vt) t.join(); // make all threads
24 };                                                 // unjoinable; see Item 2
25                                              // for info on "auto&"
```

26 The fact that design using futures can achieve this effect is noteworthy, and that's
27 why you should consider it for one-shot event communication. Other approaches
28 also work, however. Use of a boolean flag yields source code that's just as simple,
29 but its tradeoffs are different: reacting threads don't block (bad), but there's no
30 need for use of the heap (good). A condvar-based design could also be made to
31 work here, but because there's no shared data to protect by the mutex a condvar
32 requires, that approach isn't really appropriate.

1 **Things to Remember**

- 2 ♦ For simple event communication, condvar-based designs require a superfluous
3 mutex, impose constraints on the relative progress of detecting and reacting
4 tasks, and require reacting tasks to verify that the event has taken place.
- 5 ♦ Designs employing a boolean flag avoid those problems, but are based on poll-
6 ing, not blocking.
- 7 ♦ Using `std::promises` and futures dodges these issue, but it uses heap
8 memory for shared states, and it's limited to one-shot communication.

9 **Item 42: Use `std::atomic` for concurrency, `volatile` for
10 special memory.**

11 Poor `volatile`. So misunderstood. It shouldn't even be in this chapter, because it
12 has nothing to do with concurrent programming. But in other programming lan-
13 guages (e.g., Java and C#), it is useful for such programming, and even in C++, some
14 compilers have imbued `volatile` with semantics that render it helpful in concur-
15 rent software (but only when compiled with those compilers). It's thus worthwhile
16 to discuss `volatile` in a chapter on currency if for no other reason than to dispel
17 the confusion surrounding it.

18 The C++ feature that programmers sometimes confuse `volatile` with—the fea-
19 ture that definitely does belong in this chapter—is the `std::atomic` template.
20 Instantiations of this template (e.g., `std::atomic<int>`, `std::atomic<bool>`,
21 `std::atomic<Widget*>`, etc.) offer operations that are guaranteed to be seen as
22 atomic by other threads. Operations on `std::atomic` objects behave as if they
23 were inside a mutex-protected critical section, but the operations are generally
24 implemented using special machine instructions that are more efficient than they
25 would be if a mutex were employed.

26 Consider this code using `std::atomic`:

```
27 std::atomic<int> ai(0);      // atomically initialize ai to 0
28 ai = 10;                  // atomically set ai to 10
29 std::cout << ai;          // atomically read ai's value
```

```
1  ++ai;                      // atomically increment ai to 11
2  ai--;                      // atomically decrement ai to 10
3  During execution of these statements, other threads reading ai may see only val-
4  ues of 0, 10, or 11. No other values are possible (assuming, of course, that this is
5  the only thread modifying ai).
```

6 Two aspects of this example are worth noting. First, in the “`std::cout << ai;`”
7 statement, the fact that `ai` is a `std::atomic` guarantees only that the read of `ai` is
8 atomic. There is no guarantee that the entire statement proceeds atomically. Be-
9 tween the time `ai`’s value is read and `operator<<` is invoked to write it to the
10 standard output, another thread—possibly several threads—may have modified
11 `ai`’s value. That has no effect on the behavior of the statement, because `opera-`
12 `tor<<` for `ints` uses a by-value parameter for the `int` to output (the outputted
13 value will therefore be the one that was read from `ai`), but it’s important to under-
14 stand that what’s atomic in that statement is nothing more than the read of `ai`.

15 The second noteworthy aspect of the example is the behavior of the last two
16 statements—the increment and decrement of `ai`. These are each read-modify-
17 write (RMW) operations, yet, they execute atomically. This is one of the nicest
18 characteristics of the `std::atomic` types: all the member functions, including
19 those comprising RMW operations, are guaranteed to be seen by other threads as
20 atomic.

21 In contrast, the corresponding code using `volatile` guarantees virtually nothing
22 in a multithreaded context:

```
23 volatile int vi(0);          // initialize vi to 0
24 vi = 10;                     // set vi to 10
25 std::cout << vi;             // read vi's value
26 ++vi;                        // increment vi to 11
27 vi--;                         // decrement vi to 10
```

28 During execution of this code, if other threads are reading the value of `vi`, they
29 may see anything (e.g., -12, 68, 4090727—anything!). Such code would have unde-
30 fined behavior, because these statements modify `vi`, so if other threads are read-

1 ing `vi` at the same time, there are simultaneous readers and writers of memory
2 that's neither `std::atomic` nor protected by a mutex, and that's the definition of a
3 data race.

4 As a concrete example of how the behavior of `std::atomics` and `volatiles` can
5 differ in a multithreaded program, consider a simple counter of each type that's
6 incremented by multiple threads. We'll initialize each to 0:

```
7 std::atomic<int> ac(0);      // "atomic counter"  
8 volatile int vc(0);         // "volatile counter"
```

9 We'll then increment each counter one time in two simultaneously-running
10 threads:

```
11 /*----- Thread 1 ----- */      /*----- Thread 2 ----- */  
12     ac++;                      ac++;  
13     vc++;                      vc++;
```

14 When both threads have finished, `ac`'s value (i.e., the value of the `std::atomic`)
15 must be 2, because each increment occurs as an indivisible operation. `vc`'s value,
16 on the other hand, need not be 2, because its increments may not occur atomically.
17 Each increment consists of reading `vc`'s value, incrementing the value that was
18 read, and writing the result back into `vc`. But these three operations are not guar-
19 anteed to proceed atomically for `volatile` objects, so it's possible that the com-
20 ponent parts of the two increments of `vc` are interleaved as follows:

- 21 1. Thread 1 reads `vc`'s value, which is 0.
- 22 2. Thread 2 reads `vc`'s value, which is still 0.
- 23 3. Thread 1 increments the 0 it read to 1, then writes that value into `vc`.
- 24 4. Thread 2 increments the 0 it read to 1, then writes that value into `vc`.

25 `vc`'s final value is therefore 1, even though it was incremented twice.

26 This is not the only possible outcome. `vc`'s final value is, in general, not predicta-
27 ble, because `vc` is involved in a data race, and the Standard's decree that data races
28 cause undefined behavior means that compilers may generate code to do literally

1 anything. Compilers don't use this leeway to be malicious, of course. Rather, they
2 perform optimizations that would be valid in programs without data races, and
3 these optimizations yield unexpected and unpredictable behavior in programs
4 where races are present.

5 The use of RMW operations isn't the only situation where `std::atomics` com-
6 prise a concurrency success story and `volatiles` suffer ignominious failure. Sup-
7 pose one task computes an important value needed by a second task. When the
8 first task has computed the value, it must communicate this to the second task.
9 Item 41 explains that one way for the first task to communicate the availability of
10 the desired value to the second task is by using a `std::atomic<bool>`. Code in
11 the task computing the value would look something like this:

```
12 std::atomic<bool> valAvailable(false);  
13 auto imptValue = computeImportantValue(); // compute value  
14 valAvailable = true; // tell other task  
15 // it's available
```

16 As humans reading this code, we know it's crucial that the assignment to `impt-`
17 `Value` take place before the assignment to `valAvailable`, but all compilers see is
18 a pair of assignments to independent variables. As a general rule, compilers are
19 permitted to reorder such unrelated assignments. That is, given this sequence of
20 assignments (where `a`, `b`, `x`, and `y` correspond to independent variables),

```
21 a = b;  
22 x = y;
```

23 compilers may generally reorder them as follows:

```
24 x = y;  
25 a = b;
```

26 Even if compilers don't reorder them, the underlying hardware might do it (or
27 might make it seem to other cores as if it had), because that can sometimes make
28 the code run faster.

29 However, the use of `std::atomics` imposes restrictions on how code can be reor-
30 dered, and one such restriction is that no code that, in the source code, precedes a

1 write of a `std::atomic` variable may take place (or appear to other cores to take
2 place) afterwards. That means that in our code,

```
3 auto imptValue = computeImportantValue(); // compute value
4 valAvailable = true; // tell other task
5 // it's available
```

6 not only must compilers retain the order of the assignments to `imptValue` and
7 `valAvailable`, they must generate code that ensures that the underlying hard-
8 ware does, too. As a result, declaring `valAvailable` as `std::atomic` ensures that
9 our critical ordering requirement—`imptValue` must be seen by all threads to
10 change no later than `valAvailable` does—is maintained.

11 Declaring `valAvailable` as `volatile` doesn't impose the same code reordering
12 restrictions:

```
13 volatile bool valAvailable(false);
14 auto imptValue = computeImportantValue();
15 valAvailable = true; // other threads might see this assignment
16 // before the one to imptValue!
```

17 Here, compilers might flip the order of the assignments to `imptValue` and
18 `valAvailable`, and even if they don't, they might fail to generate machine code
19 that would prevent the underlying hardware from making it possible for code on
20 other cores to see `valAvailable` change before `imptValue`.

21 These two issues—no guarantee of operation atomicity and insufficient re-
22 strictions on code reordering—explain why `volatile`'s not useful for concurrent
23 programming, but it doesn't explain what it is useful for. In a nutshell, it's for tell-
24 ing compilers that they're dealing with memory that doesn't behave normally.

25 “Normal” memory has the characteristic that if you write a value to a memory loca-
26 tion, the value remains there until something overwrites it. So if I have a normal
27 `int`,

```
28 int x;
```

29 and a compiler sees the following sequence of operations on it,

```
1 auto y = x;           // read x
2 y = x;                // read x again
3 the compiler can optimize the generated code by eliminating the assignment to y,
4 because it's redundant with y's initialization.
```

```
5 Normal memory also has the characteristic that if you write a value to a memory
6 location, never read it, and then write to that memory location again, the first write
7 can be eliminated, because it was never used. So given these two adjacent state-
8 ments,
```

```
9 x = 10;               // write x
10 x = 20;              // write x again
```

```
11 compilers can eliminate the first one. That means that if we have this in the source
12 code,
```

```
13 auto y = x;           // read x
14 y = x;                // read x again
15 x = 10;               // write x
16 x = 20;              // write x again
```

```
17 compilers can treat it as if it had been written like this:
```

```
18 auto y = x;           // read x
19 x = 20;               // write x
```

```
20 Lest you wonder who'd write code that performs these kinds of redundant reads
21 and superfluous writes (technically known as redundant loads and dead stores),
22 the answer is that humans don't write it directly—at least we hope they don't.
23 However, after compilers take reasonable-looking source code and perform tem-
24 plate instantiation, inlining, and various common kinds of reordering optimiza-
25 tions, it's not uncommon for the result to have redundant loads and dead stores
26 that compilers can get rid of.
```

```
27 Such optimizations are valid only if memory behaves normally. "Special" memory
28 doesn't. Probably the most common kind of special memory is memory used for
29 memory-mapped I/O. Locations in such memory actually communicate with pe-
30 ripherals, e.g., external sensors or displays, printers, network ports, etc. rather
```

1 than reading or writing normal memory (e.g., RAM). In such a context, consider
2 again the code with seemingly redundant reads:

```
3 auto y = x;           // read x
4 y = x;                // read x again
```

5 If x corresponds to, say, the temperature reported by a temperature sensor, the
6 second read of x is not redundant, because the temperature may have changed be-
7 tween the first and second reads.

8 It's a similar situation for seemingly superfluous writes. In this code, for example,

```
9 x = 10;              // write x
10 x = 20;             // write x again
```

11 if x corresponds to the control port for a radio transmitter, it could be that the
12 code is issuing commands to the radio, and the value 10 corresponds to a different
13 command from the value 20. Optimizing out the first assignment would change the
14 sequence of commands sent to the radio.

15 **volatile** is the way we tell compilers that we're dealing with special memory. Its
16 meaning to compilers is "Don't perform any optimizations on operations on this
17 memory." So if x corresponds to special memory, it'd be declared **volatile**:

```
18 volatile int x;
```

19 Consider the effect that has on our original code sequence:

```
20 auto y = x;           // read x
21 y = x;                // read x again (can't be optimized away)
22 x = 10;              // write x
23 x = 20;                // write x again (can't be optimized away)
```

24 This is precisely what we want if x is memory-mapped (or has been mapped to a
25 memory location shared across processes, etc.).

1 Pop quiz! In that last piece of code, what is `y`'s type: `int` or `volatile int`?[†]

2 The fact that seemingly redundant loads and dead stores must be preserved when
3 dealing with special memory explains, by the way, why `std::atomics` are unsuit-
4 able for this kind of work. Compilers are permitted to eliminate such redundant
5 operations on `std::atomics`. The code isn't written quite the same way it is for
6 `volatiles`, but if we overlook that for a moment and focus on what compilers are
7 permitted to do, we can say that, conceptually, compilers may take this,

```
8 std::atomic<int> x;  
9 auto y = x;           // conceptually read x (see below)  
10 y = x;              // conceptually read x again (see below)  
11 x = 10;             // write x  
12 x = 20;             // write x again
```

13 and optimize it to this:

```
14 auto y = x;           // conceptually read x (see below)  
15 x = 20;              // write x
```

16 For special memory, this is clearly unacceptable behavior.

17 Now, as it happens, neither of these two statements will compile when `x` is
18 `std::atomic`:

```
19 auto y = x;           // error!  
20 y = x;              // error!
```

21 That's because the copy operations for `std::atomic` are deleted (see Item 11).
22 And with good reason. Consider what would happen if the initialization of `y` with `x`
23 compiled. Because `x` is `std::atomic`, `y`'s type would be deduced to be

[†] `y`'s type is `auto`-deduced, so it uses the rules described in Item 2. Those rules dictate that for the declaration of non-reference non-pointer types (which is the case for `y`), `const` and `volatile` qualifiers are dropped. `y`'s type is therefore simply `int`. This means that redundant reads of and writes to `y` can be eliminated. In the example, compilers must perform both the initialization of and the assignment to `y`, because `x` is `volatile`, so the second read of `x` might yield a different value from the first one.

1 `std::atomic`, too (see Item 2.) I remarked earlier that one of the best things
2 about `std::atomics` is that all their operations are atomic, but in order for the
3 copy construction of `y` from `x` to be atomic, compilers would have to generate code
4 to read `x` and write `y` in a single atomic operation. Hardware generally can't do
5 that, so copy construction isn't supported for `std::atomic` types. Copy assign-
6 ment is deleted for the same reason, which is why the assignment from `x` to `y`
7 won't compile. (The move operations aren't explicitly declared in `std::atomic`,
8 so, per the rules for compiler-generated special functions described in Item 19,
9 `std::atomic` offers neither move construction nor move assignment.)

10 It's possible, of course, to get the value of `x` into `y`, but it requires use of
11 `std::atomic`'s member functions `load` and `store`. The `load` member function
12 reads a `std::atomic`'s value atomically, while the `store` member function writes
13 it atomically. To initialize `y` with `x`, then, followed by putting `x`'s value in `y`, the
14 code must be written like this:

```
15 std::atomic<int> y(x.load());      // read x
16 y.store(x.load());                // read x again
```

17 This compiles, but the fact that reading `x` (via `x.load()`) is a separate function call
18 from initializing or storing to `y` makes clear that there is no reason to expect either
19 statement as a whole to execute as a single atomic operation.

20 Given that code, compilers could "optimize" it by storing `x`'s value in a register in-
21 stead of reading it twice:

```
22 register = x.load();           // read x into register
23 std::atomic<int> y(register); // init y with register value
24 y.store(register);          // store register value into y
```

25 The result, as you can see, reads from `x` only once, and that's the kind of optimiza-
26 tion that must be avoided when dealing with special memory. (The optimization
27 isn't permitted for `volatile` variables.)

28 The situation should thus be clear:

1 • `std::atomic` is useful for concurrent programming, but not for accessing spe-
2 cial memory.

3 • `volatile` is useful for accessing special memory, but not for concurrent pro-
4 gramming.

5 Because `std::atomic` and `volatile` serve different purposes, they can even be
6 used together:

```
7     volatile std::atomic<int> vai;     // operations on vai are  
8                                            // atomic and can't be  
9                                            // optimized away
```

10 This could be useful if `vai` corresponded to a memory-mapped I/O location that
11 was concurrently accessed by multiple threads.

12 As a final note, some developers prefer to use `std::atomic`'s `load` and `store`
13 member functions even when they're not syntactically required, because it makes
14 explicit in the source code that the variables involved aren't "normal." Emphasiz-
15 ing that fact isn't unreasonable. Accessing a `std::atomic` is typically much slower
16 than accessing a non-`std::atomic`, and we've already seen that the use of
17 `std::atomics` prevents compilers from performing certain kinds of statement
18 reorderings that would otherwise be permitted. Calling out loads and stores of
19 `std::atomics` can therefore help identify potential scalability chokepoints. From
20 a correctness perspective, *not* seeing a call to `store` on a variable meant to com-
21 municate information to other threads (e.g., a flag indicating the availability of da-
22 ta) could mean that the variable wasn't declared `std::atomic` when it should
23 have been.

24 This is more a stylistic issue, however, and as such is quite different from the
25 choice between `std::atomic` and `volatile`. `std::atomic` is a tool for writing
26 concurrent software. `volatile` is a tool for working with special memory.

27 **Things to Remember**

28 • `std::atomic` is for data accessed from multiple threads without using mutex-
29 es.

- 1 ♦ **volatile** is for memory where reads and writes should never be optimized
- 2 away.