



# Effective Modern C++

---

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

Scott Meyers

## 1 **Contents**

2	<b>Introduction</b>	<b>4</b>
3	<b>Chapter 1 Deducing Types</b>	<b>11</b>
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	<b>Chapter 2 auto</b>	<b>41</b>
9	Item 5: Prefer <code>auto</code> to explicit type declarations.	41
10	Item 6: Use the explicitly typed initializer idiom when <code>auto</code> deduces undesired types.	47
12	<b>Chapter 3 Moving to Modern C++</b>	<b>54</b>
13	Item 7: Distinguish between <code>()</code> and <code>{}</code> when creating objects.	54
14	Item 8: Prefer <code>nullptr</code> to <code>0</code> and <code>NULL</code> .	64
15	Item 9: Prefer alias declarations to <code>typedefs</code> .	68
16	Item 10: Prefer scoped <code>enums</code> to unscoped <code>enums</code> .	73
17	Item 11: Prefer deleted functions to private undefined ones.	80
18	Item 12: Declare overriding functions <code>override</code> .	85
19	Item 13: Prefer <code>const_iterators</code> to <code>iterators</code> .	93
20	Item 14: Declare functions <code>noexcept</code> if they won't emit exceptions.	97
21	Item 15: Use <code>constexpr</code> whenever possible.	106
22	Item 16: Make <code>const</code> member functions thread-safe.	112
23	Item 17: Understand special member function generation.	118
24	<b>Chapter 4 Smart Pointers</b>	<b>127</b>
25	Item 18: Use <code>std::unique_ptr</code> for exclusive-ownership resource management.	129

1	Item 19: Use <code>std::shared_ptr</code> for shared-ownership resource management.	136
3	Item 20: Use <code>std::weak_ptr</code> for <code>std::shared_ptr</code> -like pointers that can dangle.	147
5	Item 21: Prefer <code>std::make_unique</code> and <code>std::make_shared</code> to direct use of <code>new</code> .	152
7	Item 22: When using the Pimpl Idiom, define special member functions in the implementation file.	162
9	<b>Chapter 5 Rvalue References, Move Semantics, and Perfect Forwarding</b>	<b>172</b>
11	Item 23: Understand <code>std::move</code> and <code>std::forward</code> .	173
12	Item 24: Distinguish universal references from rvalue references.	179
13	Item 25: Use <code>std::move</code> on rvalue references, <code>std::forward</code> on universal references.	185
15	Item 26: Avoid overloading on universal references.	194
16	Item 27: Familiarize yourself with alternatives to overloading on universal references.	202
18	Item 28: Understand reference collapsing.	217
19	Item 29: Assume that move operations are not present, not cheap, and not used.	224
21	Item 30: Familiarize yourself with perfect forwarding failure cases.	228
22	<b>Chapter 6 Lambda Expressions</b>	<b>239</b>
23	Item 31: Avoid default capture modes.	241
24	Item 32: Use init capture to move objects into closures.	248
25	Item 33: Use <code>decltype</code> on <code>auto&amp;&amp;</code> parameters to <code>std::forward</code> them.	254
26	Item 34: Prefer lambdas to <code>std::bind</code> .	258
27	<b>Chapter 7 The Concurrency API</b>	<b>267</b>
28	Item 35: Prefer task-based programming to thread-based.	267

1	Item 36: Specify <code>std::launch::async</code> if asynchronicity is essential.	273
2	Item 37: Make <code>std::threads</code> unjoinable on all paths.	278
3	Item 38: Be aware of varying thread handle destructor behavior.	286
4	Item 39: Consider <code>void</code> futures for one-shot event communication.	292
5	Item 40: Use <code>std::atomic</code> for concurrency, <code>volatile</code> for special memory.	301
7	<b>Chapter 8 Tweaks</b>	<b>312</b>
8	Item 41: Consider pass by value for copyable parameters that are cheap to move and always copied.	312
10	Item 42: Consider emplacement instead of insertion.	321

## 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 you  
4    learned more, you were surprised by the scope of the changes. `auto` declarations,  
5    range-based `for` loops, lambda expressions, and rvalue references change the face  
6    of C++, to say nothing of the new concurrency features. And then there are the idi-  
7    omatic changes. `0` and `typedefs` are out, `nullptr` and alias declarations are in.  
8    Enums should now be scoped. Smart pointers are now preferable to built-in ones.  
9    Moving objects is normally better than copying them.

10   There's a lot to learn about C++11, not to mention C++14.

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

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

25   The Items in this book are guidelines, not rules, because guidelines have excep-  
26   tions. The most important part of each Item is not the advice it offers, but the ra-  
27   tionale behind the advice. Once you've read that, you'll be in a position to deter-  
28   mine whether the circumstances of your project justify a violation of the Item's  
29   guidance. The true goal of this book isn't to tell you what to do or what to avoid

1 doing, but to convey a deeper understanding of how things work in C++11 and  
2 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 official versions of  
6 C++, each named after the year in which the corresponding ISO Standard was  
7 adopted: *C++98*, *C++03*, *C++11*, and *C++14*. C++98 and C++03 differ only in tech-  
8 nical details, so in this book, I refer to both as C++98. When I refer to C++11, I  
9 mean both C++11 and C++14, because C++14 is effectively a superset of C++11.  
10 When I write C++14, I mean specifically C++14. And if I simply mention C++, I'm  
11 making a broad statement that pertains to all language versions.

Term I Use	Language Versions I Mean
C++	All
C++98	C++98 and C++03
C++11	C++11 and C++14
C++14	C++14

12 As a result, I might say that C++ places a premium on efficiency (true for all ver-  
13 sions), that C++98 lacks support for concurrency (true only for C++98 and C++03),  
14 that C++11 supports lambda expressions (true for C++11 and C++14), and that  
15 C++14 offers generalized function return type deduction (true for C++14 only).

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

23 A useful heuristic to determine whether an expression is an lvalue is to ask if you  
24 can take its address. If you can, it typically is. If you can't, it's usually an rvalue. A  
25 nice feature of this heuristic is that it helps you remember that the type of an ex-  
26 pression is independent of whether the expression is an lvalue or an rvalue. That

1 is, given a type T, you can have lvalues of type T as well as rvalues of type T. It's  
2 especially important to remember this when dealing with a parameter of rvalue  
3 reference type, because the parameter itself is an lvalue:

```
4 class Widget {  
5     public:  
6         Widget(Widget&& rhs);    // rhs is an Lvalue, though it has  
7         ...                      // an rvalue reference type  
8     };
```

9 Here, it'd be perfectly valid to take rhs's address inside Widget's move construc-  
10 tor, so rhs is an lvalue, even though its type is an rvalue reference. (By similar rea-  
11 soning, all parameters are lvalues.)

12 That code snippet demonstrates several conventions I normally follow:

13 • The class name is `Widget`. I use `Widget` whenever I want to refer to an arbi-  
14 trary user-defined type. Unless I need to show specific details of the class, I use  
15 `Widget` without declaring it.

16 • I use the parameter name `rhs` ("right-hand side"). It's my preferred parameter  
17 name for the *move operations* (i.e., move constructor and move assignment op-  
18 erator) and the *copy operations* (i.e., copy constructor and copy assignment  
19 operator). I also employ it for the right-hand parameter of binary operators:

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

21 It's no surprise, I hope, that `Lhs` stands for "left-hand side."

22 • I apply special formatting to parts of code or parts of comments to draw your  
23 attention to them. In the `Widget` move constructor above, I've highlighted the  
24 declaration of `rhs` and the part of the comment noting that `rhs` is an lvalue.  
25 Highlighted code is neither inherently good nor inherently bad. It's simply  
26 code you should pay particular attention to.

27 • I use "..." to indicate "other code could go here." This narrow ellipsis is different  
28 from the wide ellipsis ("...") that's used in the source code for C++11's vari-  
29 adic templates. That sounds confusing, but it's not. For example:

```
1 template<typename... Ts> // these are C++
2 void processVals(const Ts&... params) // source code
3 { // ellipses
4     ... // this means "some
5         // code goes here"
6 }
```

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

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

```
17 void someFunc(Widget w); // someFunc's parameter w
18 // is passed by value
19
20 Widget wid; // wid is some Widget
21 someFunc(wid); // in this call to someFunc,
22 // w is a copy of wid that's
23 // created via copy construction
24 someFunc(std::move(wid)); // in this call to SomeFunc,
25 // w is a copy of wid that's
26 // created via move construction
```

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

1 In a function call, the expressions passed at the call site are the function's *arguments*.  
2 The arguments are used to initialize the function's *parameters*. In the first  
3 call to `someFunc` above, the argument is `wid`. In the second call, the argument is  
4 `std::move(wid)`. In both calls, the parameter is `w`. The distinction between argu-  
5 ments and parameters is important, because parameters are lvalues, but the ar-  
6 guments with which they are initialized may be rvalues or lvalues. This is especial-  
7 ly relevant during the process of *perfect forwarding*, whereby an argument passed  
8 to a function is passed to a second function such that the original argument's rval-  
9 ueness or lvalueness is preserved. (Perfect forwarding is discussed in detail in  
10 Item 30.)

11 Well-designed functions are *exception-safe*, meaning they offer at least the basic  
12 exception safety guarantee (i.e., the *basic guarantee*). Such functions assure callers  
13 that even if an exception is thrown, program invariants remain intact (i.e., no data  
14 structures are corrupted) and no resources are leaked. Functions offering the  
15 strong exception safety guarantee (i.e., the *strong guarantee*) assure callers that if  
16 an exception is arises, the state of the program remains as it was prior to the call.

17 When I refer to a *function object*, I usually mean an object of a type supporting an  
18 `operator()` member function. In other words, an object that acts like a function.  
19 Occasionally I use the term in a slightly more general sense to mean anything that  
20 can be invoked using the syntax of a non-member function call (i.e., "*function-*  
21 *Name(arguments)*"). This broader definition covers not just objects supporting  
22 `operator()`, but also functions and C-like function pointers. (The narrower defi-  
23 nition comes from C++98, the broader one from C++11.) Generalizing further by  
24 adding member function pointers yields what are known as *callable objects*. You  
25 can generally ignore the fine distinctions and simply think of function objects and  
26 callable objects as things in C++ that can be invoked using some kind of function-  
27 calling syntax.

28 Function objects created through lambda expressions are known as *closures*. It's  
29 seldom necessary to distinguish between lambda expressions and the closures  
30 they create, so I often refer to both as *lambdas*. Similarly, I rarely distinguish be-  
31 tween *function templates* (i.e., templates that generate functions) and *template*

1    *functions* (i.e., the functions generated from function templates). Ditto for *class*  
2    *templates* and *template classes*.

3    Many things in C++ can be both declared and defined. *Declarations* introduce  
4    names and types without giving details, such as where storage is located or how  
5    things are implemented:

```
6  extern int x;                      // object declaration
7  class Widget;                     // class declaration
8  bool func(const Widget& w);      // function declaration
9  enum class Color;                 // scoped enum declaration
10                         // (see Item 10)
```

11 *Definitions* provide the storage locations or implementation details:

```
12 int x;                           // object definition
13 class Widget {                   // class definition
14 ...
15 };
16 bool func(const Widget& w)      // function definition
17 { return w.size() < 10; }
18 enum class Color                // scoped enum definition
19 { Yellow, Red, Blue };
```

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

22 I define a function's *signature* to be the part of its declaration that specifies parameter and return types. Function and parameter names are not part of the signature.  
23 In the example above, `func`'s signature is `bool(const Widget&)`. Elements of a function's declaration other than its parameter and return types (e.g., `noexcept` or  
24 `constexpr`, if present), are excluded. (`noexcept` and `constexpr` are described in  
25 Items 14 and 15.) The official definition of "signature" is slightly different from  
26 mine, but for this book, my definition is more useful. (The official definition sometimes  
27 omits return types.)

30 New C++ Standards generally preserve the validity of code written under older  
31 ones, but occasionally the Standardization Committee *deprecates* features. Such

1 features are on standardization death row and may be removed from future  
2 Standards. Compilers may or may not warn about the use of deprecated features,  
3 but you should do your best to avoid them. Not only can they lead to future porting  
4 headaches, they're generally inferior to the features that replace them. For exam-  
5 ple, `std::auto_ptr` is deprecated in C++11, because `std::unique_ptr` does the  
6 same job, only better.

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

14 I call built-in pointers, such as those returned from `new`, *raw pointers*. The oppo-  
15 site of a raw pointer is a *smart pointer*. Smart pointers normally overload the  
16 pointer-deferencing operators (`operator->` and `operator*`), though Item 20 ex-  
17 plains that `std::weak_ptr` is an exception.

18 In source code comments, I sometimes abbreviate "constructor" as *ctor* and "de-  
19 structor" as *dtor*.

## 20 Reporting Bugs and Suggesting Improvements

21 I've done my best to fill this book with clear, accurate, useful information, but sure-  
22 ly there are ways to make it better. If you find errors of any kind (technical, exposi-  
23 tory, grammatical, typographical, etc.), or if you have suggestions for how the book  
24 could otherwise be improved, please email me at [emc++@aristeia.com](mailto:emc++@aristeia.com). New  
25 printings give me the opportunity to revise *Effective Modern C++*, and I can't ad-  
26 dress issues I don't know about!

27 To view the list of the issues I do know about, consult the book's errata page,  
28 <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 that ruleset 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 you from the tyranny of spelling out types that are obvious or redundant. It makes  
7 C++ software more adaptable, because changing a type at one point in the source  
8 code automatically propagates through type deduction to other locations. Howev-  
9 er, it can render code more difficult to reason about, because the types deduced by  
10 compilers may not be as apparent as you'd like.

11 Without a solid understanding of how type deduction operates, effective pro-  
12 gramming in modern C++ is all but impossible. There are just too many contexts  
13 where type deduction takes place: in calls to function templates, in most situations  
14 where `auto` appears, in `decltype` expressions, and, as of C++14, where the enig-  
15 matic `decltype(auto)` construct is employed.

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

## 21 Item 1: Understand template type deduction.

22 When users of a complex system are ignorant of how it works, yet happy with  
23 what it does, that says a lot about the design of the system. By this measure, tem-  
24 plate type deduction in C++ is a tremendous success. Millions of programmers  
25 have passed arguments to template functions with completely satisfactory results,  
26 even though many of those programmers would be hard-pressed to give more  
27 than the haziest description of how the types used by those functions were de-  
28 duced.

1 If that group includes you, I have good news and bad news. The good news is that  
2 type deduction for templates is the basis for one of modern C++'s most compelling  
3 features: `auto`. If you were happy with how C++98 deduced types for templates,  
4 you're set up to be happy with how C++11 deduces types for `auto`. The bad news  
5 is that when the template type deduction rules are applied in the context of `auto`,  
6 they sometimes seem less intuitive than when they're applied to templates. For  
7 that reason, it's important to truly understand the aspects of template type deduc-  
8 tion that `auto` builds on. This Item covers what you need to know.

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

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

13 A call can look like this:

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

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

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

21 and we have this call,

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

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

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

1     • *ParamType* is a pointer or reference type, but not a universal reference. (Universal references are described in Item 24. At this point, all you need to know is that they exist and that they're not the same as lvalue references or rvalue references.)

5     • *ParamType* is a universal reference.

6     • *ParamType* is neither a pointer nor a reference.

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

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

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

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

15     1. If *expr*'s type is a reference, ignore the reference part.

16     2. Then pattern-match *expr*'s type against *ParamType* to determine T.

17     For example, if this is our template,

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

20     and we have these variable declarations,

```
21 int x = 27;           // x is an int
22 const int cx = x;     // cx is a const int
23 const int& rx = x;   // rx is a reference to x as a const int
```

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

```
25 f(x);                // T is int, param's type is int&
```

```
26 f(cx);                // T is const int,
27                                // param's type is const int&
```

```
1  f(rx);          // T is const int,  
2                  // param's type is const int&  
  
3  In the second and third calls, notice that because cx and rx designate const val-  
4  ues, T is deduced to be const int, thus yielding a parameter type of const int&. That's important to callers. When they pass a const object to a reference parameter, they expect that object to remain unmodifiable, i.e., for the parameter to be a reference-to-const. That's why passing a const object to a template taking a T& parameter is safe: the constness of the object becomes part of the type deduced for T.  
  
10 In the third example, note that even though rx's type is a reference, T is deduced to  
11 be a non-reference. That's because rx's reference-ness is ignored during type de-  
12 duction.  
  
13 These examples all show lvalue reference parameters, but type deduction works  
14 exactly the same way for rvalue reference parameters. Of course, only rvalue ar-  
15 guments may be passed to rvalue reference parameters, but that restriction has  
16 nothing to do with type deduction.  
  
17 If we change the type of f's parameter from T& to const T&, things change a little,  
18 but not in any really surprising ways. The constness of cx and rx continues to be  
19 respected, but because we're now assuming that param is a reference-to-const,  
20 there's no longer a need for const to be deduced as part of T:  
  
21 template<typename T>  
22 void f(const T& param); // param is now a ref-to-const  
  
23 int x = 27;           // as before  
24 const int cx = x;    // as before  
25 const int& rx = x;  // as before  
  
26 f(x);               // T is int, param's type is const int&  
27 f(cx);               // T is int, param's type is const int&  
28 f(rx);               // T is int, param's type is const int&
```

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

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

```

1 template<typename T>
2 void f(T* param);           // param is now a pointer
3 int x = 27;                 // as before
4 const int *px = &x;          // px is a ptr to x as a const int
5
6 f(&x);                    // T is int, param's type is int*
7 f(px);                     // T is const int,
8                         // param's type is const int*

```

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

### 13 Case 2: *ParamType* is a Universal Reference

14 Things are less obvious for templates taking universal reference parameters. Such  
 15 parameters are declared like rvalue references (i.e., in a function template taking a  
 16 type parameter T, a universal reference's declared type is T&&), but they behave  
 17 differently when lvalue arguments are passed in. The complete story is told in  
 18 Item 24, but here's the headline version:

- 19 • If *expr* is an lvalue, both T and *ParamType* are deduced to be lvalue references. That's doubly unusual. First, it's the only situation in template type deduction where T is deduced to be a reference. Second, although *ParamType* is declared using the syntax for an rvalue reference, its deduced type is an lvalue reference.
- 24 • If *expr* is an rvalue, the "normal" (i.e., Case 1) rules apply.

25 For example:

```

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

```

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

7 Item 24 explains exactly why these examples play out the way they do. The key  
8 point here is that the type deduction rules for universal reference parameters are  
9 different from those for parameters that are lvalue references or rvalue references.  
10 In particular, when universal references are in use, type deduction distinguishes  
11 between lvalue arguments and rvalue arguments. That never happens for non-  
12 universal references.

### 13 Case 3: *ParamType* is Neither a Pointer nor a Reference

14 When *ParamType* is neither a pointer nor a reference, we're dealing with pass-by-  
15 value:

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

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

- 21 1. As before, if *expr*'s type is a reference, ignore the reference part.
- 22 2. If, after ignoring *expr*'s reference-ness, *expr* is *const*, ignore that, too. If it's  
23 *volatile*, also ignore that. (*volatile* objects are uncommon. They're gener-  
24 ally used only for implementing device drivers. For details, see Item 40.)

25 Hence:

```
26 int x = 27;          // as before
27 const int cx = x;    // as before
28 const int& rx = x;  // as before
29 f(x);                // T's and param's types are both int
30 f(cx);               // T's and param's types are again both int
```

```
1 f(rx); // T's and param's types are still both int
2 Note that even though cx and rx represent const values, param isn't const. That
3 makes sense. param is an object that's completely independent of cx and rx—a
4 copy of cx or rx. The fact that cx and rx can't be modified says nothing about
5 whether param can be. That's why expr's constness (and volatileness, if any)
6 is ignored when deducing a type for param: just because expr can't be modified
7 doesn't mean that a copy of it can't be.
```

```
8 It's important to recognize that const (and volatile) is ignored only for by-
9 value parameters. As we've seen, for parameters that are references-to- or point-
10 ers-to-const, the constness of expr is preserved during type deduction. But con-
11 sider the case where expr is a const pointer to a const object, and expr is passed
12 to a by-value param:
```

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

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

## 28 Array Arguments

```
29 That pretty much covers it for mainstream template type deduction, but there's a
30 niche case that's worth knowing about. It's that array types are different from
31 pointer types, even though they sometimes seem to be interchangeable. A primary
```

1 contributor to this illusion is that, in many contexts, an array *decays* into a pointer  
2 to its first element. This decay is what permits code like this to compile:

```
3 const char name[] = "J. P. Briggs"; // name's type is
4 // const char[13]
```

```
5 const char * ptrToName = name; // array decays to pointer
```

6 Here, the `const char*` pointer `ptrToName` is being initialized with `name`, which is  
7 a `const char[13]`. These types (`const char*` and `const char[13]`) are not the  
8 same, but because of the array-to-pointer decay rule, the code compiles.

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

```
11 template<typename T>
12 void f(T param); // template with by-value parameter
13 f(name); // what types are deduced for T and param?
```

14 We begin with the observation that there is no such thing as a function parameter  
15 that's an array. Yes, yes, the syntax is legal,

```
16 void myFunc(int param[]);
```

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

```
19 void myFunc(int* param); // same function as above
```

20 This equivalence of array and pointer parameters is a bit of foliage springing from  
21 the C roots at the base of C++, and it fosters the illusion that array and pointer  
22 types are the same.

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

```
27 f(name); // name is array, but T deduced as const char*
```

1 But now comes a curve ball. Although functions can't declare parameters that are  
2 truly arrays, they *can* declare parameters that are *references* to arrays! So if we  
3 modify the template `f` to take its argument by reference,

4 `template<typename T>`  
5 `void f(T& param);` // template with by-reference parameter

6 and we pass an array to it,

7 `f(name);` // pass array to `f`

8 the type deduced for `T` is the actual type of the array! That type includes the size of  
9 the array, so in this example, `T` is deduced to be `const char [13]`, and the type of  
10 `f`'s parameter (a reference to this array) is `const char (&)[13]`. Yes, the syntax  
11 looks toxic, but knowing it will score you mondo points with those few souls who  
12 care.

13 Interestingly, the ability to declare references to arrays enables creation of a tem-  
14 plate that deduces the number of elements that an array contains:

15 // return size of an array as a compile-time constant. (The  
16 // array parameter has no name, because we care only about  
17 // the number of elements it contains.)  
18 `template<typename T, std::size_t N>` // info  
19 `constexpr std::size_t arraySize(T (&)[N]) noexcept` // below on  
20 { // constexpr  
21 `return N;` // and  
22 } // noexcept

23 As Item 15 explains, declaring this function `constexpr` makes its result available  
24 during compilation. That makes it possible to declare, say, an array with the same  
25 number of elements as a second array whose size is computed from a braced ini-  
26 tializer:

27 `int keyVals[] = { 1, 3, 7, 9, 11, 22, 35 };` // keyVals has  
28 // 7 elements  
29 `int mappedVals[arraySize(keyVals)];` // so does  
30 // mappedVals

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

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

3 As for `arraySize` being declared `noexcept`, that's to help compilers generate better code. For details, see Item 14.

## 5 Function Arguments

6 Arrays aren't the only things in C++ that can decay into pointers. Function types  
7 can decay into function pointers, and everything we've discussed regarding type  
8 deduction for arrays applies to type deduction for functions and their decay into  
9 function pointers. As a result:

```
10 void someFunc(int, double); // someFunc is a function;  
11 // type is void(int, double)  
  
12 template<typename T>  
13 void f1(T param); // in f1, param passed by value  
  
14 template<typename T>  
15 void f2(T& param); // in f2, param passed by ref  
  
16 f1(someFunc); // param deduced as ptr-to-func;  
17 // type is void (*)(int, double)  
  
18 f2(someFunc); // param deduced as ref-to-func;  
19 // type is void (&)(int, double)
```

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

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

## 29 Things to Remember

- 30 • During template type deduction, arguments that are references are treated as  
31 non-references, i.e., their reference-ness is ignored.

- When deducing types for universal reference parameters, lvalue arguments get special treatment.
- When deducing types for by-value parameters, `const` and/or `volatile` arguments are treated as non-`const` and non-`volatile`.
- During template type deduction, arguments that are array or function names decay to pointers, unless they're used to initialize references.

## 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.

In Item 1, template type deduction is explained using this general function template

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

and this general call:

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

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

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

```
auto x = 27;
```

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

```
1 const auto cx = x;
2 the type specifier is const auto. And here,
3 const auto& rx = x;
4 the type specifier is const auto&. To deduce types for x, cx, and rx in these ex-
5 amples, compilers act as if there were a template for each declaration as well as a
6 call to that template with the corresponding initializing expression:
```

```
7 template<typename T> // conceptual template for
8 void func_for_x(T param); // deducing x's type
9 func_for_x(27); // conceptual call: param's
10 // deduced type is x's type
11 template<typename T> // conceptual template for
12 void func_for_cx(const T param); // deducing cx's type
13 func_for_cx(x); // conceptual call: param's
14 // deduced type is cx's type
15 template<typename T> // conceptual template for
16 void func_for_rx(const T& param); // deducing rx's type
17 func_for_rx(x); // conceptual call: param's
18 // deduced type is rx's type
```

19 As I said, deducing types for **auto** is, with only one exception (which we'll discuss  
20 soon), the same as deducing types for templates.

21 Item 1 divides template type deduction into three cases, based on the characteris-
22 tics of *ParamType*, the type specifier for **param** in the general function template. In
23 a variable declaration using **auto**, the type specifier takes the place of *ParamType*,
24 so there are three cases for that, too:

- 25 • Case 1: The type specifier is a pointer or reference, but not a universal refer-
26 ence.
- 27 • Case 2: The type specifier is a universal reference.
- 28 • Case 3: The type specifier is neither a pointer nor a reference.

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

```
30 auto x = 27; // case 3 (x is neither ptr nor reference)
```

```
1 const auto cx = x;      // case 3 (cx isn't either)
2 const auto& rx = x;    // case 1 (rx is a non-universal ref.)
```

3 Case 2 works as you'd expect:

```
4 auto&& uref1 = x;      // x is int and lvalue,
5                                // so uref1's type is int&
6 auto&& uref2 = cx;     // cx is const int and lvalue,
7                                // so uref2's type is const int&
8 auto&& uref3 = 27;      // 27 is int and rvalue,
9                                // so uref3's type is int&&
```

10 Item 1 concludes with a discussion of how array and function names decay into  
11 pointers for non-reference type specifiers. That happens in **auto** type deduction,  
12 too:

```
13 const char name[] =           // name's type is const char[13]
14     "R. N. Briggs";
15 auto arr1 = name;           // arr1's type is const char*
16 auto& arr2 = name;           // arr2's type is
17                                // const char (&)[13]
18
19 void someFunc(int, double); // someFunc is a function;
20                                // type is void(int, double)
21 auto func1 = someFunc;       // func1's type is
22                                // void (*)(int, double)
23 auto& func2 = someFunc;       // func2's type is
24                                // void (&)(int, double)
```

25 As you can see, **auto** type deduction works like template type deduction. They're  
26 essentially two sides of the same coin.

27 Except for the one way they differ. We'll start with the observation that if you want  
28 to declare an **int** with an initial value of 27, C++98 gives you two syntactic choices:

```
30 int x1 = 27;
31 int x2(27);
```

32 C++11, through its support for uniform initialization, adds these:

```
1 int x3 = { 27 };
2 int x4{ 27 };
```

3 All in all, four syntaxes, but only one result: an `int` with value 27.

4 But as Item 5 explains, there are advantages to declaring variables using `auto` instead of fixed types, so it'd be nice to replace `int` with `auto` in the above variable declarations. Straightforward textual substitution yields this code:

```
7 auto x1 = 27;
8 auto x2(27);
9 auto x3 = { 27 };
10 auto x4{ 27 };
```

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

```
15 auto x1 = 27;           // type is int, value is 27
16 auto x2(27);           // ditto
17 auto x3 = { 27 };       // type is std::initializer_list<int>,
18                         // value is { 27 }
19 auto x4{ 27 };          // ditto
```

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

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

26 As the comment indicates, type deduction will fail in this case, but it's important to  
27 recognize that there are actually two kinds of type deduction taking place. One  
28 kind stems from the use of `auto`: `x5`'s type has to be deduced. Because `x5`'s initial-  
29 izer is in braces, `x5` must be deduced to be a `std::initializer_list`. But  
30 `std::initializer_list` is a template. Instantiations are  
31 `std::initializer_list<T>` for some type `T`, and that means that `T`'s type must  
32 also be deduced. Such deduction falls under the purview of the second kind of type

1 deduction occurring here: template type deduction. In this example, that deduction  
2 fails, because the values in the braced initializer don't have a single type.

3 The treatment of braced initializers is the only way in which `auto` type deduction  
4 and template type deduction differ. When an `auto`-declared variable is initialized  
5 with a braced initializer, the deduced type is an instantiation of  
6 `std::initializer_list`. But if the corresponding template is passed the same  
7 initializer, type deduction fails, and the code is rejected:

```
8 auto x = { 11, 23, 9 };      // x's type is
9                                // std::initializer_list<int>
10 template<typename T>        // template with parameter
11 void f(T param);           // declaration equivalent to x's
12 f({ 11, 23, 9 });          // error! can't deduce type for T
13 However, if you specify in the template that param is a
14 std::initializer_list<T> for some unknown T, template type deduction will
15 deduce what T is:
```

```
16 template<typename T>
17 void f(std::initializer_list<T> initList);
18 f({ 11, 23, 9 });          // T deduced as int, and initList's
19                                // type is std::initializer_list<int>
```

20 So the only real difference between `auto` and template type deduction is that `auto`  
21 *assumes* that a braced initializer represents a `std::initializer_list`, but tem-  
22 plate type deduction doesn't.

23 You might wonder why `auto` type deduction has a special rule for braced initializ-  
24 ers, but template type deduction does not. I wonder this myself. Alas, I have not  
25 been able to find a convincing explanation. But the rule is the rule, and this means  
26 you must remember that if you declare a variable using `auto` and you initialize it  
27 with a braced initializer, the deduced type will always be  
28 `std::initializer_list`. It's especially important to bear this in mind if you  
29 embrace the philosophy of uniform initialization—of enclosing initializing values  
30 in braces as a matter of course. A classic mistake in C++11 programming is acci-  
31 dentally declaring a `std::initializer_list` variable when you mean to declare

1 something else. This pitfall is one of the reasons some developers put braces  
2 around their initializers only when they have to. (When you have to is discussed in  
3 Item 7.)

4 For C++11, this is the full story, but for C++14, the tale continues. C++14 permits  
5 `auto` to indicate that a function's return type should be deduced (see Item 3), and  
6 C++14 lambdas may use `auto` in parameter declarations. However, these uses of  
7 `auto` employ *template type deduction*, not `auto` type deduction. So a function with  
8 an `auto` return type that returns a braced initializer won't compile:

```
9 auto createInitList()  
10 {  
11     return { 1, 2, 3 };           // error: can't deduce type  
12 }                           // for { 1, 2, 3 }
```

13 The same is true when `auto` is used in a parameter type specification in a C++14  
14 lambda:

```
15 std::vector<int> v;  
16 ...  
17 auto resetV =  
18     [&v](const auto& newValue) { v = newValue; };      // C++14  
19 ...  
20 resetV({ 1, 2, 3 });           // error! can't deduce type  
21                               // for { 1, 2, 3 }
```

## 22 Things to Remember

- 23 • `auto` type deduction is usually the same as template type deduction, but `auto`  
24 type deduction assumes that a braced initializer represents a  
25 `std::initializer_list`, and template type deduction doesn't.
- 26 • `auto` in a function return type or a lambda parameter implies template type de-  
27 duction, not `auto` type deduction.

## 28 Item 3: Understand `decltype`.

29 `decltype` is an odd creature. Given a name or an expression, `decltype` tells you  
30 the name's or the expression's type. Typically, what it tells you is exactly what

1 you'd predict. Occasionally however, it provides results that leave you scratching  
2 your head and turning to reference works or online Q&A sites for revelation.

3 We'll begin with the typical cases—the ones harboring no surprises. In contrast to  
4 what happens during type deduction for templates and `auto` (see Items 1 and 2),  
5 `decltype` typically parrots back the exact type of the name or expression you give  
6 it:

```
7 const int i = 0;           // decltype(i) is const int
8 bool f(const Widget& w); // decltype(w) is const Widget&
9                           // decltype(f) is bool(const Widget&)

10 struct Point {
11     int x, y;             // decltype(Point::x) is int
12 };                      // decltype(Point::y) is int
13 Widget w;                // decltype(w) is Widget
14 if (f(w)) ...            // decltype(f(w)) is bool
15 template<typename T>    // simplified version of std::vector
16 class vector {
17 public:
18 ...
19     T& operator[](std::size_t index);
20 ...
21 };
22 vector<int> v;          // decltype(v) is vector<int>
23 ...
24 if (v[0] == 0) ...        // decltype(v[0]) is int&
```

25 See? No surprises.

26 In C++11, perhaps the primary use for `decltype` is declaring function templates  
27 where the function's return type depends on its parameter types. For example,  
28 suppose we'd like to write a function that takes a container that supports indexing  
29 via square brackets (i.e., the use of “[ ]”) plus an index, then authenticates the user  
30 before returning the result of the indexing operation. The return type of the func-  
31 tion should be the same as the type returned by the indexing operation.

32 `operator[]` on a container of objects of type `T` typically returns a `T&`. This is the  
33 case for `std::deque`, for example, and it's almost always the case for  
34 `std::vector`. For `std::vector<bool>`, however, `operator[]` does not return a

1    `bool&`. Instead, it returns a brand new object. The whys and hows of this situation  
2    are explored in Item 6, but what's important here is that the type returned by a  
3    container's `operator[]` depends on the container.

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

```
7  template<typename Container, typename Index>      // works, but
8  auto authAndAccess(Container& c, Index i)          // requires
9    -> decltype(c[i])                                // refinement
10 {
11   authenticateUser();
12   return c[i];
13 }
```

14   The use of `auto` before the function name has nothing to do with type deduction.  
15   Rather, it indicates that C++11's *trailing return type* syntax is being used, i.e., that  
16   the function's return type will be declared following the parameter list (after the  
17   “`->`”). A trailing return type has the advantage that the function's parameters can  
18   be used in the specification of the return type. In `authAndAccess`, for example, we  
19   specify the return type using `c` and `i`. If we were to have the return type precede  
20   the function name in the conventional fashion, `c` and `i` would be unavailable, be-  
21   cause they would not have been declared yet.

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

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

```
31 template<typename Container, typename Index>      // C++14;
32 auto authAndAccess(Container& c, Index i)          // not quite
33 {
```

```
1     authenticateUser();  
2     return c[i];           // return type deduced from c[i]  
3 }
```

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

6 Perhaps surprisingly, functions with an `auto` return type perform type deduction  
7 using the template type deduction rules. It might seem that use of the `auto` type  
8 deduction rules would better correspond to the declaration syntax, but remember  
9 that template type deduction and `auto` type deduction are nearly identical. The  
10 only difference is that template type deduction deduces no type for a braced ini-  
11 tializer.

12 In this case, deducing `authAndAccess`'s return type using template type deduc-  
13 tion is problematic, but `auto`'s type deduction rules would fare no better. The dif-  
14 ficulty stems from something these forms of type deduction have in common: their  
15 treatment of expressions that are references.

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

```
21 std::deque<int> d;  
22 ...  
23 authAndAccess(d, 5) = 10; // authenticate user, return d[5],  
24                         // then assign 10 to it;  
25                         // this won't compile!
```

26 Here, `d[5]` returns an `int&`, but `auto` return type deduction for `authAndAccess`  
27 will strip off the reference, thus yielding a return type of `int`. That `int`, being the  
28 return value of a function, is an rvalue, and the code above thus attempts to assign  
29 10 to an rvalue `int`. That's forbidden in C++, so the code won't compile.

30 The problem is that we're using template type deduction, which discards reference  
31 qualifiers from its initializing expression. What we want in this case is `decltype`

1 type deduction. Such type deduction would permit us to say that `authAndAccess`  
2 should return exactly the same type that the expression `c[i]` returns.

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

```
9 template<typename Container, typename Index> // C++14; works,
10 decltype(auto) // but still
11 authAndAccess(Container& c, Index i) // requires
12 { // refinement
13     authenticateUser();
14     return c[i];
15 }
```

16 Now `authAndAccess` will truly return whatever `c[i]` returns. In particular, for  
17 the common case where `c[i]` returns a `T&`, `authAndAccess` will also return a `T&`,  
18 and in the uncommon case where `c[i]` returns an object, `authAndAccess` will  
19 return an object, too.

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

```
23 Widget w;
24 const Widget& cw = w;
25 auto myWidget1 = cw; // auto type deduction:
26 // myWidget1's type is Widget
27 decltype(auto) myWidget2 = cw; // decltype type deduction:
28 // myWidget2's type is
29 // const Widget&
```

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

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

```
1 template<typename Container, typename Index>
2 decltype(auto) authAndAccess(Container& c, Index i);
3
4 The container is passed by lvalue-reference-to-non-const, because returning a
5 reference to an element of the container permits clients to modify that container.
6 But this means it's not possible to pass rvalue containers to this function. Rvalues
7 can't bind to lvalue references (unless they're lvalue-references-to-const, which is
8 not the case here).
9
10 Admittedly, passing an rvalue container to authAndAccess is an edge case. An
11 rvalue container, being a temporary object, would typically be destroyed at the
12 end of the statement containing the call to authAndAccess, and that means that a
13 reference to an element in that container (which is typically what authAndAccess
14 would return) would dangle at the end of the statement that created it. Still, it
15 could make sense to pass a temporary object to authAndAccess. A client might
16 simply want to make a copy of an element in the temporary container, for exam-
17 ple:
```

```
16 std::deque<std::string> makeStringDeque(); // factory function
17 // make copy of 5th element of deque returned
18 // from makeStringDeque
19 auto s = authAndAccess(makeStringDeque(), 5);
```

```
20 Supporting such use means we need to revise the declaration for authAndAccess
21 to accept both lvalues and rvalues. Overloading would work (one overload would
22 declare an lvalue reference parameter, the other an rvalue reference parameter),
23 but then we'd have two functions to maintain. A way to avoid that is to have au-
24 thAndAccess employ a reference parameter that can bind to lvalues and rvalues,
25 and Item 24 explains that that's exactly what universal references do. authAndAc-
26 cess can therefore be declared like this:
```

```
27 template<typename Container, typename Index> // c is now a
28 decltype(auto) authAndAccess(Container&& c, // universal
29                               Index i); // reference
```

```
30 In this template, we don't know what type of container we're operating on, and
31 that means we're equally ignorant of the type of index objects it uses. Employing
32 pass-by-value for objects of an unknown type generally risks the performance hit
33 of unnecessary copying, the behavioral problems of object slicing (see Item 41),
```

1 and the sting of our coworkers' derision, but in the case of container indices, fol-  
2 lowing the example of the Standard Library for index values (e.g., in `operator[]`  
3 for `std::string`, `std::vector`, and `std::deque`) seems reasonable, so we'll  
4 stick with pass-by-value for them.

5 However, we need to update the template's implementation to bring it into accord  
6 with Item 25's admonition to apply `std::forward` to universal references:

```
7 template<typename Container, typename Index>          // final
8 decltype(auto)                                         // C++14
9 authAndAccess(Container&& c, Index i)                // version
10 {
11     authenticateUser();
12     return std::forward<Container>(c)[i];
13 }
```

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

```
17 template<typename Container, typename Index>          // final
18 auto                                                 // C++11
19 authAndAccess(Container&& c, Index i)                // version
20 -> decltype(std::forward<Container>(c)[i])
21 {
22     authenticateUser();
23     return std::forward<Container>(c)[i];
24 }
```

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

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

32 Applying `decltype` to a name yields the declared type for that name. Names are  
33 lvalue expressions, but that doesn't affect `decltype`'s behavior. For lvalue expres-  
34 sions more complicated than names, however, `decltype` ensures that the type  
35 reported is always an lvalue reference. That is, if an lvalue expression other than a

1 name has type T, `decltype` reports that type as T&. This seldom has any impact,  
2 because the type of most lvalue expressions inherently includes an lvalue refer-  
3 ence qualifier. Functions returning lvalues, for example, always return lvalue ref-  
4 erences.

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

6 `int x = 0;`

7 x is the name of a variable, so `decltype(x)` is `int`. But wrapping the name x in  
8 parentheses—“(x)”—yields an expression more complicated than a name. Being a  
9 name, x is an lvalue, and C++ defines the expression (x) to be an lvalue, too.  
10 `decltype((x))` is therefore `int&`. Putting parentheses around a name can change  
11 the type that `decltype` reports for it!

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

```
15 decltype(auto) f1()
16 {
17     int x = 0;
18     ...
19     return x;           // decltype(x) is int, so f1 returns int
20 }
21 decltype(auto) f2()
22 {
23     int x = 0;
24     ...
25     return (x);       // decltype((x)) is int&, so f2 returns int&
26 }
```

27 Note that not only does f2 have a different return type from f1, it’s also returning  
28 a reference to a local variable! That’s the kind of code that puts you on the express  
29 train to undefined behavior—a train you certainly don’t want to be on.

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

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

6 **Things to Remember**

- 7   ♦ `decltype` almost always yields the type of a variable or expression without  
8       any modifications.
- 9   ♦ For lvalue expressions of type T other than names, `decltype` always reports a  
10      type of `T&`.
- 11   ♦ C++14 supports `decltype(auto)`, which, like `auto`, deduces a type from its  
12      initializer, but it performs the type deduction using the `decltype` rules.

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

14 The choice of tools for viewing the results of type deduction is dependent on the  
15 phase of the software development process where you want the information. We'll  
16 explore three possibilities: getting type deduction information as you edit your  
17 code, getting it during compilation, and getting it at run time.

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-

1 piler (or at least the front end of one) running inside the IDE. If that compiler can't  
2 make enough sense of your code to parse it and perform type deduction, it can't  
3 show you what types it deduced.

4 For simple types like `int`, information from IDEs is generally fine. As we'll see  
5 soon, however, when more complicated types are involved, the information dis-  
6 played by IDEs may not be particularly helpful.

## 7 Compiler Diagnostics

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

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

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

16 Attempts to instantiate this template will elicit an error message, because there's  
17 no template definition to instantiate. To see the types for `x` and `y`, just try to instan-  
18 tiate `TD` with their types:

```
19 TD<decltype(x)> xType;      // elicit errors containing  
20 TD<decltype(y)> yType;      // x's and y's types;  
21                                         // see Item 3 for info on decltype
```

22 I use variable names of the form `variableNameType`, because they tend to yield  
23 error messages that help me find the information I'm looking for. For the code  
24 above, one of my compilers issues diagnostics reading, in part, as follows. (I've  
25 highlighted the type information we're after.)

```
26 error: aggregate 'TD<int> xType' has incomplete type and  
27     cannot be defined  
28 error: aggregate 'TD<const int *> yType' has incomplete type  
29     and cannot be defined
```

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

```
1 error: 'xType' uses undefined class 'TD<int>'  
2 error: 'yType' uses undefined class 'TD<const int *>'  
3 Formatting differences aside, all the compilers I've tested produce error messages  
4 with useful type information when this technique is employed.
```

## 5 Runtime Output

```
6 The printf approach to displaying type information (not that I'm recommending  
7 you use printf) can't be employed until run time, but it offers full control over the  
8 formatting of the output. The challenge is to create a textual representation of the  
9 type you care about that is suitable for display. "No sweat," you're thinking, "it's  
10 typeid and std::type_info::name to the rescue." In our continuing quest to  
11 see the types deduced for x and y, you may figure we can write this:
```

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

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

```
18 Calls to std::type_info::name are not guaranteed to return anything sensible,  
19 but implementations try to be helpful. The level of helpfulness varies. The GNU and  
20 Clang compilers report that the type of x is "i", and the type of y is "PKi", for ex-  
21 ample. These results make sense once you learn that, in output from these compil-  
22 ers, "i" means "int" and "PK" means "pointer to const const." (Both compilers  
23 support a tool, c++filt, that decodes such "mangled" types.) Microsoft's compiler  
24 produces less cryptic output: "int" for x and "int const *" for y.
```

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

```
28 template<typename T> // template function to  
29 void f(const T& param); // be called  
30 std::vector<Widget> createVec(); // factory function
```

```
1 const auto vw = createVec(); // init vw w/factory return
2 if (!vw.empty()) {
3     f(&vw[0]); // call f
4     ...
5 }
```

6 This code, which involves a user-defined type (`Widget`), an STL container  
7 (`std::vector`), and an `auto` variable (`vw`), is more representative of the situations where you might want some visibility into the types your compilers are deducing. For example, it'd be nice to know what types are inferred for the template type parameter `T` and the function parameter `param` in `f`.

11 Loosing `typeid` on the problem is straightforward. Just add some code to `f` to display the types you'd like to see:

```
13 template<typename T>
14 void f(const T& param)
15 {
16     using std::cout;
17     cout << "T = " << typeid(T).name() << '\n'; // show T
18     cout << "param = " << typeid(param).name() << '\n'; // show
19     ... // param's
20 } // type
```

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

```
22 T = PK6Widget
23 param = PK6Widget
```

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

28 Microsoft’s compiler concurs:

```
29 T = class Widget const *
30 param = class Widget const *
```

31 Three independent compilers producing the same information suggests that the information is accurate. But look more closely. In the template `f`, `param`’s declared type is `const T&`. That being the case, doesn’t it seem odd that `T` and `param` have

1 the same type? If `T` were `int`, for example, `param`'s type should be `const int&`—  
2 not the same type at all.

3 Sadly, the results of `std::type_info::name` are not reliable. In this case, for ex-  
4 ample, the type that all three compilers report for `param` are incorrect. Further-  
5 more, they're essentially *required* to be incorrect, because the specification for  
6 `std::type_info::name` mandates that the type be treated as if it had been  
7 passed to a template function as a by-value parameter. As Item 1 explains, that  
8 means that if the type is a reference, its reference-ness is ignored, and if the type  
9 after reference removal is `const` (or `volatile`), its constness (or volatileness)  
10 is also ignored. That's why `param`'s type—which is `const Widget * const &`—is  
11 reported as `const Widget*`. First the type's reference-ness is removed, and then  
12 the constness of the resulting pointer is eliminated.

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

16 `const`  
17 `std::_Simple_types<std::_Wrap_alloc<std::_Vec_base_types<Widget,`  
18 `std::allocator<Widget> >::_Alloc>::value_type>::value_type *`

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

20 `const std::_Simple_types<...>::value_type *const &`

21 That's less intimidating than the type for `T`, but the “`...`” in the middle is confusing  
22 until you realize that it's the IDE editor's way of saying “I'm omitting all that stuff  
23 that's part of `T`'s type.” With any luck, your development environment does a bet-  
24 ter job on code like this.

25 If you're more inclined to rely on libraries than luck, you'll be pleased to know that  
26 where `std::type_info::name` and IDEs may fail, the Boost TypeIndex library  
27 (often written as `Boost.TypeIndex`) is designed to succeed. The library isn't part of  
28 Standard C++, but neither are IDEs or templates like TD. Furthermore, the fact that  
29 Boost libraries (available at [boost.com](http://boost.com)) are cross-platform, open source, and  
30 available under a license designed to be palatable to even the most paranoid cor-

1 porate legal team means that code using Boost libraries is nearly as portable as  
2 code relying on the Standard Library.

3 Here's how our function `f` can produce accurate type information using  
4 `Boost.TypeIndex`:

```
5 #include <boost/type_index.hpp>
6
7 template<typename T>
8 void f(const T& param)
9 {
10     using std::cout;
11     using boost::typeindex::type_id_with_cvr;
12
13     // show T
14     cout << "T = "
15     << type_id_with_cvr<T>().pretty_name()
16     << '\n';
17
18     // show param's type
19     cout << "param = "
20     << type_id_with_cvr<decltype(param)>().pretty_name()
21     << '\n';
22
23     ...
24 }
```

21 The way this works is that the function template  
22 `boost::typeindex::type_id_with_cvr` takes a type argument (the type about  
23 which we want information) and *doesn't* remove `const`, `volatile`, or reference  
24 qualifiers (hence the “`with_cvr`” in the template name). The result is a  
25 `boost::typeindex::type_index` object, whose `pretty_name` member function  
26 produces a `std::string` containing a human-friendly representation of the type.

27 With this implementation for `f`, consider again the call that yields incorrect type  
28 information for `param` when `typeid` is used:

```
29 std::vector<Widget> createVec();      // factory function
30 const auto vw = createVec();           // init vw w/factory return
31 if (!vw.empty()) {
32     f(&vw[0]);                      // call f
33     ...
34 }
```

1 Under compilers from GNU and Clang, Boost.TypeIndex produces this (accurate)  
2 output:

3 `T = Widget const*`  
4 `param = Widget const* const&`

5 Results under Microsoft's compiler are essentially the same:

6 `T = class Widget const *`  
7 `param = class Widget const * const &`

8 Such near-uniformity is nice, but it's important to remember that IDE editors,  
9 compiler error messages, and libraries like Boost.TypeIndex are merely tools you  
10 can use to help you figure out what types your compilers are deducing. All can be  
11 helpful, but at the end of the day, there's no substitute for understanding the type  
12 deduction information in Items 1-3.

### 13 **Things to Remember**

- 14   ♦ Deduced types can often be seen using IDE editors, compiler error messages,  
15   and the Boost TypeIndex library.
- 16   ♦ The results of some tools may be neither helpful nor accurate, so an under-  
17   standing of C++'s type deduction rules remains essential.

1    **Chapter 2 auto**

2    In concept, `auto` is as simple as simple can be, but it's more subtle than it looks.  
3    Using it saves typing, sure, but it also prevents correctness and performance issues  
4    that can bedevil manual type declarations. Furthermore, some of `auto`'s type de-  
5    duction results, while dutifully conforming to the prescribed algorithm, are, from  
6    the perspective of a programmer, just wrong. When that's the case, it's important  
7    to know how to guide `auto` to the right answer, because falling back on manual  
8    type declarations is an alternative that's often best avoided.

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

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

11    Ah, the simple joy of

12    `int x;`

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

15    Never mind. Let's move on to the simple joy of declaring a local variable to be ini-  
16    tialized by dereferencing an iterator:

```
17 template<typename It>      // algorithm to dwim ("do what I mean")  
18 void dwim(It b, It e)       // for all elements in range from  
19 {                          // b to e  
20     while (b != e) {  
21         typename std::iterator_traits<It>::value_type  
22         currValue = *b;  
23         ...  
24     }  
25 }
```

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

29    Okay, simple joy (take three): the delight of declaring a local variable whose type is  
30    that of a closure. Oh, right. The type of a closure is known only to the compiler,  
31    hence can't be written out. Sigh. Damn.

1 Damn, damn, damn! Programming in C++ is not the joyous experience it should be!  
2 Well, it didn't used to be. But as of C++11, all these issues go away, courtesy of au-  
3 to. `auto` variables have their type deduced from their initializer, so they must be  
4 initialized. That means you can wave goodbye to a host of uninitialized variable  
5 problems as you speed by on the modern C++ superhighway:

```
6 int x1; // potentially uninitialized  
7 auto x2; // error! initializer required  
8 auto x3 = 0; // fine, x's value is well-defined
```

9 Said highway lacks the potholes associated with declaring a local variable whose  
10 value is that of a dereferenced iterator:

```
11 template<typename It> // as before  
12 void dwim(It b, It e)  
13 {  
14     while (b != e) {  
15         auto currValue = *b;  
16         ...  
17     }  
18 }
```

19 And because `auto` uses type deduction (see Item 2), it can represent types known  
20 only to compilers:

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

25 Very cool. In C++14, the temperature drops further, because parameters to lambda  
26 expressions may involve `auto`:

```
27 auto derefLess = // C++14 comparison  
28 [](const auto& p1, // function for  
29         const auto& p2) // values pointed  
30     { return *p1 < *p2; }; // to by anything  
31                                 // pointer-like
```

32 Coolness notwithstanding, perhaps you're thinking we don't really need `auto` to  
33 declare a variable that holds a closure, because we can use a `std::function` ob-

1 ject. It's true, we can, but possibly that's not what you were thinking. And maybe  
2 now you're thinking "What's a `std::function` object?" So let's clear that up.

3 `std::function` is a template in the C++11 Standard Library that generalizes the  
4 idea of a function pointer. Whereas function pointers can point only to functions,  
5 however, `std::function` objects can refer to any callable object, i.e., to anything  
6 that can be invoked like a function. Just as you must specify the type of function to  
7 point to when you create a function pointer (i.e., the signature of the functions you  
8 want to point to), you must specify the type of function to refer to when you create  
9 a `std::function` object. You do that through `std::function`'s template param-  
10 eter. For example, to declare a `std::function` object named `func` that could re-  
11 fer to any callable object acting as if it had this signature,

```
12 bool(const std::unique_ptr<Widget>&, // C++11 signature for
13       const std::unique_ptr<Widget>&) // std::unique_ptr<Widget>
14       // comparison function
```

15 you'd write this:

```
16 std::function<bool(const std::unique_ptr<Widget>&,
17                     const std::unique_ptr<Widget>&)> func;
```

18 Because lambda expressions yield callable objects, closures can be stored in  
19 `std::function` objects. That means we could declare the C++11 version of  
20 `derefUPLess` without using `auto` as follows:

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

26 It's important to recognize that even setting aside the syntactic verbosity and need  
27 to repeat the parameter types, using `std::function` is not the same as using `au-`  
28 `to`. An `auto`-declared variable holding a closure has the same type as the closure,  
29 and as such it uses only as much memory as the closure requires. The type of a  
30 `std::function`-declared variable holding a closure is an instantiation of the  
31 `std::function` template, and that has a fixed size for any given signature. This  
32 size may not be adequate for the closure it's asked to store, and when that's the  
33 case, the `std::function` constructor will allocate heap memory to store the clo-

1 sure. The result is that the `std::function` object typically uses more memory  
2 than the `auto`-declared object. And, thanks to implementation details that restrict  
3 inlining and yield indirect function calls, invoking a closure via a `std::function`  
4 object is almost certain to be slower than calling it via an `auto`-declared object. In  
5 other words, the `std::function` approach is generally bigger and slower than  
6 the `auto` approach, and it may yield out-of-memory exceptions, too. Plus, as you  
7 can see in the examples above, writing “`auto`” is a whole lot less work than writing  
8 the type of the `std::function` instantiation. In the competition between `auto`  
9 and `std::function` for holding a closure, it’s pretty much game, set, and match  
10 for `auto`. (A similar argument can be made for `auto` over `std::function` for  
11 holding the result of calls to `std::bind`, but in Item 34, I do my best to convince  
12 you to use lambdas instead of `std::bind`, anyway.)

13 The advantages of `auto` extend beyond the avoidance of uninitialized variables,  
14 verbose variable declarations, and the ability to directly hold closures. One is the  
15 ability to avoid what I call problems related to “type shortcuts.” Here’s something  
16 you’ve probably seen—possibly even written:

```
17 std::vector<int> v;  
18 ...  
19 unsigned sz = v.size();
```

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

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

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

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

```
2 std::unordered_map<std::string, int> m;
3 ...
4 for (const std::pair<std::string, int>& p : m)
5 {
6     ... // do something with p
7 }
```

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

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

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

```
22 for (const auto& p : m)
23 {
24     ... // as before
25 }
```

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

31 The last two examples—writing `unsigned` when you should have written  
32 `std::vector<int>::size_type` and writing `std::pair<std::string, int>`

1 when you should have written `std::pair<const std::string, int>`—  
2 demonstrate how explicitly specifying types can lead to implicit conversions that  
3 you neither want nor expect. If you use `auto` as the type of the target variable, you  
4 need not worry about mismatches between the type of variable you’re declaring  
5 and the type of the expression used to initialize it.

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

14 First, take a deep breath and relax. `auto` is an option, not a mandate. If, in your  
15 professional judgment, your code will be clearer or more maintainable or in some  
16 other way better by using explicit type declarations, you’re free to continue using  
17 them. But bear in mind that C++ breaks no new ground in adopting what is gener-  
18 ally known in the programming languages world as *type inference*. Other statically  
19 typed procedural languages (e.g., C#, D, Scala, Visual Basic) have a more or less  
20 equivalent feature, to say nothing of a variety of statically typed functional lan-  
21 guages (e.g., ML, Haskell, OCaml, F#, etc.). In part, this is due to the success of dy-  
22 namically typed languages such as Perl, Python, and Ruby, where variables are  
23 rarely explicitly typed. The software development community has extensive expe-  
24 rience with type inference, and it has demonstrated that there is nothing contra-  
25 dictory about such technology and the creation and maintenance of large, industri-  
26 al-strength code bases.

27 Some developers are disturbed by the fact that using `auto` eliminates the ability to  
28 determine an object’s type by a quick glance at the source code. However, IDEs’  
29 ability to show object types often mitigates this problem (even taking into account  
30 the IDE type-display issues mentioned in Item 4), and, in many cases, a somewhat  
31 abstract view of an object’s type is just as useful as the exact type. It often suffices,  
32 for example, to know that an object is a container or a counter or a smart pointer,

1 without knowing exactly what kind of container, counter, or smart pointer it is.  
2 Assuming well-chosen variable names, such abstract type information should al-  
3 most always be at hand.

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

13 **Things to Remember:**

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

19 **Item 6: Use the explicitly typed initializer idiom when `auto`  
20 deduces undesired types.**

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

26 `std::vector<bool> features(const Widget& w);`

27 Further suppose that bit 5 indicates whether the `Widget` has high priority. We can  
28 thus write code like this:

```

1 Widget w;
2 ...
3 bool highPriority = features(w)[5]; // is w high priority?
4 ...
5 processWidget(w, highPriority);      // process w in accord
6                                // with its priority
7 There's nothing wrong with this code. It'll work fine. But if we make the seemingly
8 innocuous change of replacing the explicit type for highPriority with auto,
9 auto highPriority = features(w)[5]; // is w high priority?
10 the situation changes. All the code will continue to compile, but its behavior is no
11 longer predictable:
12 processWidget(w, highPriority);      // undefined behavior!
13 As the comment indicates, the call to processWidget now has undefined behav-
14 ior. But why? The answer is likely to be surprising. In the code using auto, the type
15 of highPriority is no longer bool. Though std::vector<bool> conceptually
16 holds bools, operator[] for std::vector<bool> doesn't return a reference to
17 an element of the container (which is what std::vector::operator[] returns
18 for every type except bool). Instead, it returns an object of type
19 std::vector<bool>::reference (a class nested inside std::vector<bool>).
20 std::vector<bool>::reference exists because std::vector<bool> is speci-
21 fied to represent its bools in packed form, one bit per bool. That creates a prob-
22 lem for std::vector<bool>'s operator[], because operator[] for
23 std::vector<T> is supposed to return a T&, but C++ forbids references to bits.
24 Not being able to return a bool&, operator[] for std::vector<bool> returns
25 an object that acts like a bool&. For this act to succeed,
26 std::vector<bool>::reference objects must be usable in essentially all con-
27 texts where bool&s can be. Among the features in
28 std::vector<bool>::reference that make this work is an implicit conversion
29 to bool. (Not to bool&, to bool. To explain the full set of techniques used by
30 std::vector<bool>::reference to emulate the behavior of a bool& would
31 take us too far afield, so I'll simply remark that this implicit conversion is only one
32 stone in a larger mosaic.)

```

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

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

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

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

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

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

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

22 The call to `features` returns a temporary `std::vector<bool>` object. This ob-  
23 ject has no name, but for purposes of this discussion, I'll call it `temp`. `operator[]`  
24 is invoked on `temp`, and the `std::vector<bool>::reference` it returns con-  
25 tains a pointer to a word in the data structure holding the bits that are managed by  
26 `temp`, plus the offset into that word corresponding to bit 5. `highPriority` is a  
27 copy of this `std::vector<bool>::reference` object, so `highPriority`, too,  
28 contains a pointer to a word in `temp`, plus the offset corresponding to bit 5. At the  
29 end of the statement, `temp` is destroyed, because it's a temporary object. There-  
30 fore, `highPriority` contains a dangling pointer, and that's the cause of the unde-  
31 fined behavior in the call to `processWidget`:

```
1 processWidget(w, highPriority);      // undefined behavior!
2                                         // highPriority contains
3                                         // dangling pointer!

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

13 Some proxy classes are designed to be apparent to clients. That's the case for
14 std::shared_ptr and std::unique_ptr, for example. Other proxy classes are
15 designed to act more or less invisibly. std::vector<bool>::reference is an
16 example of such "invisible" proxies, as is its std::bitset compatriot,
17 std::bitset::reference.

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

22 Matrix sum = m1 + m2 + m3 + m4;
23 can be computed much more efficiently if operator+ for Matrix objects returns a
24 proxy for the result instead of the result itself. That is, operator+ for two Matrix
25 objects would return an object of a proxy class such as Sum<Matrix, Matrix> in-
26 stead of a Matrix object. As was the case with std::vector<bool>::reference
27 and bool, there'd be an implicit conversion from the proxy class to Matrix, which
28 would permit the initialization of sum from the proxy object produced by the ex-
29 pression on the right side of the "=".
(The type of that object would traditionally
30 encode the entire initialization expression, i.e., be something like
31 Sum<Sum<Sum<Matrix, Matrix>, Matrix>, Matrix>. That's definitely a type
32 from which clients should be shielded.)
```

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

6 You therefore want to avoid code of this form:

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

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

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

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

```
21 namespace std {                                     // from C++ Standards
22     template <class Allocator>
23     class vector<bool, Allocator> {
24     public:
25         ...
26         class reference { ... };
27         reference operator[](size_type n);
28         ...
29     };
30 }
```

31 Assuming you know that `operator[]` for `std::vector<T>` normally returns a  
32 `T&`, the unconventional return type for `operator[]` in this case is a tip-off that a

1 proxy class is in use. Paying careful attention to the interfaces you're using can often reveal the existence of proxy classes.

3 In practice, many developers discover the use of proxy classes only when they try  
4 to track down mystifying compilation problems or debug incorrect unit test results.  
5 Regardless of how you find them, once `auto` has been determined to be deducing  
6 the type of a proxy class instead of the type being proxied, the solution  
7 need not involve abandoning `auto`. `auto` itself isn't the problem. The problem is  
8 that `auto` isn't deducing the type you want it to deduce. The solution is to force a  
9 different type deduction. The way you do that is what I call *the explicitly typed ini-*  
10 *tializer idiom*.

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

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

15 Here, `features(w)[5]` continues to return a `std::vector<bool>::reference`  
16 object, just as it always has, but the cast changes the type of the expression to  
17 `bool`, which `auto` then deduces as the type for `highPriority`. At run time, the  
18 `std::vector<bool>::reference` object returned from  
19 `std::vector<bool>::operator[]` executes the conversion to `bool` that it supports,  
20 and as part of that conversion, the still-valid pointer to the  
21 `std::vector<bool>` returned from `features` is dereferenced. That avoids the  
22 undefined behavior we ran into earlier. The index 5 is then applied to the bits  
23 pointed to by the pointer, and the `bool` value that emerges is used to initialize  
24 `highPriority`.

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

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

27 Applications of the idiom aren't limited to initializers yielding proxy class types. It  
28 can also be useful to emphasize that you are deliberately creating a variable of a  
29 type that is different from that generated by the initializing expression. For example,  
30 suppose you have a function to calculate some tolerance value:

```
1 double calcEpsilon();           // return tolerance value
2 calcEpsilon clearly returns a double, but suppose you know that for your application,
3 the precision of a float is adequate, and you care about the difference in
4 size between floats and doubles. You could declare a float variable to store the
5 result of calcEpsilon,
```

```
6 float ep = calcEpsilon();      // implicitly convert
7                                     // double→float
```

8 but this hardly announces “I’m deliberately reducing the precision of the value returned by the function.” A declaration using the explicitly typed initializer idiom,  
9 however, does:

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

12 Similar reasoning applies if you have a floating point expression that you are deliberately storing as an integral value. Suppose you need to calculate the index of  
13 an element in a container with random access iterators (e.g., a std::vector,  
14 std::deque, or std::array), and you’re given a double between 0.0 and 1.0  
15 indicating how far from the beginning of the container the desired element is located. (0.5 would indicate the middle of the container.) Further suppose that  
16 you’re confident that the resulting index will fit in an int. If the container is c and  
17 the double is d, you could calculate the index this way,  
18

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

21 but this obscures the fact that you’re intentionally converting the double on the  
22 right to an int. The explicitly typed initializer idiom makes things transparent:

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

#### 24 Things to Remember

- 25   ♦ “Invisible” proxy types can cause auto to deduce the “wrong” type for an initializing expression.
- 27   ♦ The explicitly typed initializer idiom forces auto to deduce the type you want it to have.

## 1    **Chapter 3   Moving to Modern C++**

2    When it comes to big-name features, C++11 and C++14 have a lot to boast of. `auto`,  
3    smart pointers, move semantics, lambdas, concurrency—each is so important, I  
4    devote a chapter to it. It's essential to master those features, but becoming an ef-  
5    fective modern C++ programmer requires a series of smaller steps, too. Each step  
6    answers specific questions that arise during the journey from C++98 to modern  
7    C++. When should you use braces instead of parentheses for object creation? Why  
8    are alias declarations better than `typedefs`? How does `constexpr` differ from  
9    `const`? What's the relationship between `const` member functions and thread  
10   safety? The list goes on and on. And one by one, this chapter provides the answers.

### 11    **Item 7: Distinguish between () and {} when creating ob- 12         jects.**

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
```

```
1 Widget w2 = w1;           // not an assignment; calls copy ctor
2 w1 = w2;                 // an assignment; calls copy operator=
3 Even with several initialization syntaxes, there were some situations where C++98
4 had no way to express a desired initialization. For example, it wasn't possible to
5 directly indicate that an STL container should be created holding a particular set of
6 values (e.g., 1, 3, and 5).
```

7 To address the confusion of multiple initialization syntaxes, as well as the fact that  
8 they don't cover all initialization scenarios, C++11 introduces *uniform initialization*:  
9 a single initialization syntax that can, at least in concept, be used anywhere  
10 and express everything. It's based on braces, and for that reason I prefer the term  
11 *braced initialization*. "Uniform initialization" is an idea. "Braced initialization" is a  
12 syntactic construct.

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

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

```
19 class Widget {
20 ...
21 private:
22     int x{ 0 };           // fine, x's default value is 0
23     int y = 0;            // also fine
24     int z(0);             // error!
25 };
```

26 On the other hand, uncopyable objects (e.g., `std::atomic`s—see Item 40) may be  
27 initialized using braces or parentheses, but not using "=":

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

1 It's thus easy to understand why braced initialization is called "uniform." Of C++'s  
2 three ways to designate an initializing expression, only braces can be used every-  
3 where.

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 by 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 Functions can't be declared using braces for the parameter list, so default-  
29 constructing an object using braces doesn't have this problem:

```
30 Widget w3{};          // calls Widget ctor with no args
```

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

5 The drawback to braced initialization is the sometimes-surprising behavior that  
6 accompanies it. Such behavior grows out of the unusually tangled relationship  
7 among braced initializers, `std::initializer_lists`, and constructor overload  
8 resolution. Their interactions can lead to code that seems like it should do one  
9 thing, but actually does another. For example, Item 2 explains that when an `auto`-  
10 declared variable has a braced initializer, the type deduced is  
11 `std::initializer_list`, even though other ways of declaring a variable with  
12 the same initializer would yield a more intuitive type. As a result, the more you like  
13 `auto`, the less enthusiastic you're likely to be about braced initialization.

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

```
16 class Widget {  
17 public:  
18     Widget(int i, bool b);           // ctors not declaring  
19     Widget(int i, double d);        // std::initializer_list params  
20     ...  
21 };  
22 Widget w1(10, true);                // calls first ctor  
23 Widget w2{10, true};                // also calls first ctor  
24 Widget w3(10, 5.0);                // calls second ctor  
25 Widget w4{10, 5.0};                // also calls second ctor
```

26 If, however, one or more constructor declares a parameter of type  
27 `std::initializer_list`, calls using the braced initialization syntax strongly  
28 prefer the overloads taking `std::initializer_lists`. *Strongly*. If there's *any*  
29 way for compilers to construe a call using a braced initializer to be to a constructor  
30 taking a `std::initializer_list`, compilers will employ that interpretation. If  
31 the `Widget` class above is augmented with a constructor taking a  
32 `std::initializer_list<long double>`, for example,

```
1 class Widget {
2 public:
3     Widget(int i, bool b); // as before
4     Widget(int i, double d); // as before
5     Widget(std::initializer_list<long double> il); // added
6     ...
7 };
8 Widgets w2 and w4 will be constructed using the new constructor, even though the
9 type of the std::initializer_list elements (long double) is, compared to
10 the non-std::initializer_list constructors, a worse match for both argu-
11 ments! Look:
```

```
12 Widget w1(10, true); // uses parens and, as before,
13 // calls first ctor
14 Widget w2{10, true}; // uses braces, but now calls
15 // std::initializer_list ctor
16 // (10 and true convert to long double)
17 Widget w3(10, 5.0); // uses parens and, as before,
18 // calls second ctor
19 Widget w4{10, 5.0}; // uses braces, but now calls
20 // std::initializer_list ctor
21 // (10 and 5.0 convert to long double)
```

```
22 Even what would normally be copy and move construction can be hijacked by
23 std::initializer_list constructors:
```

```
24 class Widget {
25 public:
26     Widget(int i, bool b); // as before
27     Widget(int i, double d); // as before
28     Widget(std::initializer_list<long double> il); // as before
29     operator int() const; // convert to int
30     ...
31 };
32 Widget w5(w4); // uses parens, calls copy ctor
33 Widget w6{w4}; // uses braces, calls
34 // std::initializer_list ctor
35 // (w4 converts to int, and int
36 // converts to long double)
```

```
1 Widget w7(std::move(w4));           // uses parens, calls move ctor
2 Widget w8{std::move(w4)};           // uses braces, calls
3                                         // std::initializer_list ctor
4                                         // (for same reason as w6)
```

5 Compilers' determination to match braced initializers with constructors taking  
6 `std::initializer_lists` is so strong, it prevails even if the best-match  
7 `std::initializer_list` constructor can't be called. For 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); // element type is
13                                         // now bool
14    ...
15 };
16 Widget w{10, 5.0}; // error! requires narrowing conversions
```

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

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

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

```
1     ...                                // no implicit  
2 };                                     // conversion funcs  
3 Widget w1(10, true);        // uses parens, still calls first ctor  
4 Widget w2{10, true};        // uses braces, now calls first ctor  
5 Widget w3(10, 5.0);         // uses parens, still calls second ctor  
6 Widget w4{10, 5.0};         // uses braces, now calls second ctor
```

7 This brings us near the end of our examination of braced initializers and construc-  
8 tor overloading, but there's an interesting edge case that needs to be addressed.  
9 Suppose you use an empty set of braces to construct an object that supports de-  
10 fault construction and also supports `std::initializer_list` construction.  
11 What do your empty braces mean? If they mean "no arguments," you get default  
12 construction, but if they mean "empty `std::initializer_list`," you get con-  
13 struction from a `std::initializer_list` with no elements.

14 You get default construction. Empty braces mean no arguments, not an empty  
15 `std::initializer_list`:

```
16 class Widget {  
17 public:  
18     Widget();                           // default ctor  
19     Widget(std::initializer_list<int> il); // std::initializer-  
20                                         // _list ctor  
21     ...                                // no implicit  
22 };                                     // conversion funcs  
23 Widget w1;                             // calls default ctor  
24 Widget w2{};                            // also calls default ctor  
25 Widget w3();                           // most vexing parse! declares a function!  
26 If you want to call a std::initializer_list constructor with an empty  
27 std::initializer_list, you do it by making the empty braces a constructor  
28 argument—by putting the empty braces inside the parentheses or braces demar-  
29 cating what you're passing!  
30 Widget w4({});                         // calls std::initializer_list ctor  
31                                         // with empty list
```

```
1 Widget w5{{}};           // ditto
2 At this point, with seemingly arcane rules about braced initializers,
3 std::initializer_lists, and constructor overloading burbling about in your
4 brain, you may be wondering how much of this information matters in day-to-day
5 programming. More than you might think, because one of the classes directly af-
6 fected is std::vector. std::vector has a non-std::initializer_list con-
7 structor that allows you to specify the initial size of the container and a value each
8 of the initial elements should have, but it also has a constructor taking a
9 std::initializer_list that permits you to specify the initial values in the con-
10 tainer. If you create a std::vector of a numeric type (e.g., a std::vector<int>)
11 and you pass two arguments to the constructor, whether you enclose those argu-
12 ments in parentheses or braces makes a tremendous difference:
```

```
13 std::vector<int> v1(10, 20); // use non-std::initializer_list
14                               // ctor: create 10-element
15                               // std::vector, all elements have
16                               // value of 20
17 std::vector<int> v2{10, 20}; // use std::initializer_list ctor:
18                               // create 2-element std::vector,
19                               // element values are 10 and 20
```

20 But let's step back from std::vector and also from the details of parentheses,
21 braces, and constructor overloading resolution rules. There are two primary take-
22 aways from this discussion. First, as a class author, you need to be aware that if
23 your set of overloaded constructors includes one or more functions taking a
24 std::initializer\_list, client code using braced initialization may see only the
25 std::initializer\_list overloads. As a result, it's best to design your construc-
26 tors so that the overload called isn't affected by whether clients use parentheses or
27 braces. In other words, learn from what is now viewed as an error in the design of
28 the std::vector interface, and design your classes to avoid it.

29 An implication is that if you have a class with no std::initializer\_list con-
30 structor, and you add one, client code using braced initialization may find that calls
31 that used to resolve to non-std::initializer\_list constructors now resolve
32 to the new function. Of course, this kind of thing can happen any time you add a
33 new function to a set of overloads: calls that used to resolve to one of the old over-

1 loads might start calling the new one. The difference with  
2 `std::initializer_list` constructor overloads is that a  
3 `std::initializer_list` overload doesn't just compete with other overloads, it  
4 overshadows them to the point where the other overloads may hardly be consid-  
5 ered. So add such overloads only with great deliberation.

6 The second lesson is that as a class client, you must choose carefully between pa-  
7 rentheses and braces when creating objects. Most developers end up choosing one  
8 kind of delimiter as a default, using the other only when they have to. Braces-by-  
9 default folks are attracted by their unrivaled breadth of applicability, their prohibi-  
10 tion of narrowing conversions, and their immunity to C++'s most vexing parse.  
11 Such folks understand that in some cases (e.g., creation of a `std::vector` with a  
12 given size and initial element value), parentheses are required. On the other hand,  
13 the go-parentheses-go crowd embraces parentheses as their default argument de-  
14 limiter. They're attracted to its consistency with the C++98 syntactic tradition, its  
15 avoidance of the auto-deduced-a-`std::initializer_list` problem, and the  
16 knowledge that their object creation calls won't be inadvertently waylaid by  
17 `std::initializer_list` constructors. They concede that sometimes only brac-  
18 es will do (e.g., when creating a container with particular values). There's no con-  
19 sensus that either approach is better than the other, so my advice is to pick one  
20 and apply it consistently.

21 If you're a template author, the tension between parentheses and braces for object  
22 creation can be especially frustrating, because, in general, it's not possible to know  
23 which should be used. For example, suppose you'd like to create an object of an  
24 arbitrary type from an arbitrary number of arguments. A variadic template makes  
25 this conceptually straightforward:

```
26 template<typename T,           // type of object to create
27         typename... Ts>        // types of arguments to use
28 void doSomeWork(Ts&&... params)
29 {
30     create Local T object from params...
31 ...
32 }
```

1 There are two ways to turn the line of pseudocode into real code (see Item 25 for  
2 information about `std::forward`):

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

5 So consider this calling code:

6 `std::vector<int> v;`  
7 `...`  
8 `doSomeWork<std::vector<int>>(10, 20);`

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

13 This is precisely the problem faced by the Standard Library functions  
14 `std::make_unique` and `std::make_shared` (see Item 21). These functions re-  
15 solve the problem by internally using parentheses and by documenting this deci-  
16 sion as part of their interfaces.<sup>†</sup>

## 17 Things to Remember

- 18   ♦ Braced initialization is the most widely usable initialization syntax, it prevents  
19   narrowing conversions, and it's immune to C++'s most vexing parse.
- 20   ♦ During constructor overload resolution, braced initializers are matched to  
21   `std::initializer_list` parameters if at all possible, even if other construc-  
22   tors offer seemingly better matches.
- 23   ♦ An example of where the choice between parentheses and braces can make a  
24   significant difference is creating a `std::vector<numeric type>` with two ar-  
25   guments.

---

<sup>†</sup> More flexible designs—ones that permit callers to determine whether parentheses or braces should be used in functions generated from a template—are possible. For details, see the 5 June 2013 entry of *Andrzej's C++ blog*, “[Intuitive interface — Part I](#).”

- 1   ♦ Choosing between parentheses and braces for object creation inside templates  
2   can be challenging.

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

4   So here's the deal: the literal `0` is an `int`, not a pointer. If C++ finds itself looking at  
5   `0` in a context where only a pointer can be used, it'll grudgingly interpret `0` as a null  
6   pointer, but that's a fallback position. C++'s primary policy is that `0` is an `int`, not a  
7   pointer.

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

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

```
16 void f(int);           // three overloads of f
17 void f(bool);
18 void f(void*);
```

```
19 f(0);                 // calls f(int), not f(void*)
20 f(NULL);              // might not compile, but typically calls
                        // f(int). Never calls f(void*)
```

22   The uncertainty regarding the behavior of `f(NULL)` is a reflection of the leeway  
23   granted to implementations regarding the type of `NULL`. If `NULL` is defined to be,  
24   say, `0L` (i.e., `0` as a `long`), the call is ambiguous, because conversion from `long` to  
25   `int`, `long` to `bool`, and `0L` to `void*` are considered equally good. The interesting  
26   thing about that call is the contradiction between the *apparent* meaning of the  
27   source code ("I'm calling `f` with `NULL`—the null pointer") and its *actual* meaning  
28   ("I'm calling `f` with some kind of integer—not the null pointer"). This counterintu-  
29   itive behavior is what led to the guideline for C++98 programmers to avoid over-  
30   loading on pointer and integral types. That guideline remains valid in C++11, be-

1 cause, the advice of this Item notwithstanding, it's likely that some developers will  
2 continue to use `0` and `NULL`, even though `nullptr` is a better choice.

3 `nullptr`'s advantage is that it doesn't have an integral type. To be honest, it  
4 doesn't have a pointer type, either, but you can think of it as a pointer of *all* types.  
5 `nullptr`'s actual type is `std::nullptr_t`, and, in a wonderfully circular definition,  
6 `std::nullptr_t` is defined to be the type of `nullptr`. The type  
7 `std::nullptr_t` implicitly converts to all raw pointer types, and that's what  
8 makes `nullptr` act as if it were a pointer of all types.

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

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

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

15 `auto result = findRecord( /* arguments */ );`  
16 `if (result == 0) {`  
17  `...`  
18 `}`

19 If you don't happen to know (or can't easily find out) what `findRecord` returns, it  
20 may not be clear whether `result` is a pointer type or an integral type. After all, `0`  
21 (what `result` is tested against) could go either way. If you see the following, on  
22 the other hand,

23 `auto result = findRecord( /* arguments */ );`  
24 `if (result == nullptr) {`  
25  `...`  
26 `}`

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

28 `nullptr` shines especially brightly when templates enter the picture. Suppose you  
29 have some functions that should be called only when the appropriate mutex has  
30 been locked. Each function takes a different kind of pointer:

```

1 int     f1(std::shared_ptr<Widget> spw); // call these only when
2 double f2(std::unique_ptr<Widget> upw); // the appropriate
3 bool    f3(Widget* pw);                  // mutex is locked
4
5 Calling code that wants to pass null pointers could look like this:
6
7 std::mutex f1m, f2m, f3m;           // mutexes for f1, f2, and f3
8 using MuxGuard =                   // C++11 typedef; see Item 9
9   std::lock_guard<std::mutex>;
10 ...
11 {
12     MuxGuard g(f1m);             // lock mutex for f1
13     auto result = f1(0);        // pass 0 as null ptr to f1
14     g.unlock();                // unlock mutex
15 ...
16 {
17     MuxGuard g(f2m);             // lock mutex for f2
18     auto result = f2(NULL);    // pass NULL as null ptr to f2
19     g.unlock();                // unlock mutex
20 ...
21 {
22     MuxGuard g(f3m);             // lock mutex for f3
23     auto result = f3(nullptr);  // pass nullptr as null ptr to f3
24     g.unlock();                // unlock mutex

```

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

```

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

```

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

```
5 template<typename FuncType,
6         typename MuxType,
7         typename PtrType>
8 decltype(auto) lockAndCall(FuncType func,           // C++14
9                           MuxType& mutex,
10                          PtrType ptr)
11 {
12     MuxGuard g(mutex);
13     return func(ptr);
14 }
```

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

```
16 auto result1 = lockAndCall(f1, f1m, 0);           // error!
17 ...
18 auto result2 = lockAndCall(f2, f2m, NULL);        // error!
19 ...
20 auto result3 = lockAndCall(f3, f3m, nullptr);      // fine
```

21 Well, they can write it, but, as the comments indicate, in two of the three cases, the  
22 code won't compile. The problem in the first call is that when `0` is passed to  
23 `lockAndCall`, template type deduction kicks in to figure out its type. The type of `0`  
24 is, was, and always will be `int`, so that's the type of the parameter `ptr` inside the  
25 instantiation of this call to `lockAndCall`. Unfortunately, this means that in the call  
26 to `func` inside `lockAndCall`, an `int` is being passed, and that's not compatible  
27 with the `std::shared_ptr<Widget>` parameter that `f1` expects. The `0` passed in  
28 the call to `lockAndCall` was intended to represent a null pointer, but what actual-  
29 ly got passed was a run-of-the-mill `int`. Trying to pass this `int` to `f1` as a  
30 `std::shared_ptr<Widget>` is a type error. The call to `lockAndCall` with `0` fails  
31 because inside the template, an `int` is being passed to a function that requires a  
32 `std::shared_ptr<Widget>`.

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

5 In contrast, the call involving `nullptr` has no trouble. When `nullptr` is passed to  
6 `lockAndCall`, the type for `ptr` is deduced to be `std::nullptr_t`. When `ptr` is  
7 passed to `f3`, there's an implicit conversion from `std::nullptr_t` to `Widget*`,  
8 because `std::nullptr_t` implicitly convert to all pointer types.

9 The fact that template type deduction deduces the “wrong” types for `0` and `NULL`  
10 (i.e., their true types, rather than their fallback meaning as a representation for a  
11 null pointer) is the most compelling reason to use `nullptr` instead of `0` or `NULL`  
12 when you want to refer to a null pointer. With `nullptr`, templates pose no special  
13 challenge. Combined with the fact that `nullptr` doesn't suffer from the overload  
14 resolution surprises that `0` and `NULL` are susceptible to, the case is iron-clad. When  
15 you want to refer to a null pointer, use `nullptr`, not `0` or `NULL`.

## 16 **Things to Remember**

- 17 ♦ Prefer `nullptr` to `0` and `NULL`.
- 18 ♦ Avoid overloading on integral and pointer types.

## 19 **Item 9: Prefer alias declarations to `typedefs`.**

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

26 Avoiding such medical tragedies is easy. Introduce a `typedef`:

```
27 typedef
28   std::unique_ptr<std::unordered_map<std::string, std::string>>
29   UPtrMapSS;
```

1 But **typedefs** are soooo C++98. They work in C++11, sure, but C++11 also offers  
2 *alias declarations*:

3 **using UPtrMapSS =**  
4     **std::unique\_ptr<std::unordered\_map<std::string, std::string>>;**

5 Given that the **typedef** and the alias declaration do *exactly* the same thing, it's  
6 reasonable to wonder whether there is a solid technical reason for preferring one  
7 over the other.

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

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

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

19 But a compelling reason does exist: templates. In particular, alias declarations may  
20 be templatized (in which case they're called *alias templates*), while **typedefs** can-  
21 not. This gives C++11 programmers a straightforward mechanism for expressing  
22 things that in C++98 had to be hacked together with **typedefs** nested inside tem-  
23 platized **structs**. For example, consider defining a synonym for a linked list that  
24 uses a custom allocator, **MyAlloc**. With an alias template, it's a piece of 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    // 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 nested  
29 `typedef`), but you're not a compiler. When compilers process the `Widget` tem-  
30 plate and encounter the use of `MyAllocList<T>` (i.e. use of the alias template),  
31 they know that `MyAllocList<T>` is the name of a type, because `MyAllocList` is  
32 an alias template: it *must* name a type. `MyAllocList<T>` is thus a *non-dependent*  
33 *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, e.g., you might want to turn `const`  
27 `std::string&` into `std::string`. Or you might want to add `const` to a type or  
28 turn it into an lvalue reference, e.g., turn `Widget` into `const Widget` or into `Widget&`. (If you haven't done any TMP, that's too bad, because if you want to be a truly  
29 effective C++ programmer, you need to be familiar with at least the basics of this  
30 facet of C++. You can see examples of TMP in action, including the kinds of type  
31 transformations I just mentioned, in Items 23 and 27.)

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  
25 C++98-style `enums`. The names of such enumerators belong to the scope contain-  
26 ing the `enum`, and that means that nothing else in that scope may have the same  
27 name:

```
28 enum Color { black, white, red }; // black, white, red are
29 // in same scope as Color
30 auto 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 term for this kind of `enum`: *unscoped*. Their new  
3 C++11 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 auto 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`”, they’re sometimes referred  
14 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  
17 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 completely  
20 valid:

```
21 enum Color { black, white, red };      // unscoped enum
22 std::vector<std::size_t>               // func. returning
23 primeFactors(std::size_t x);           // prime factors of x
24 Color c = red;
25 ...
26 if (c < 14.5) {                      // compare Color to double (!)
27     auto factors =                    // compute prime factors
28     primeFactors(c);                // 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. There are no implicit con-  
33 versions 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. The work grows out of the fact that every `enum`  
25 in C++ has an integral *underlying type* that is determined by compilers. For an un-  
26 scoped `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. However, some enums have a range of values that is much
30 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 unusual  
5 machines (where a `char` consists of at least 32 bits), compilers will have to select  
6 an integral type larger than `char` for the representation of `Status` values.

7 To make efficient use of memory, compilers often want to choose the smallest un-  
8 derlying type for an `enum` that's sufficient to represent its range of enumerator  
9 values. In some cases, compilers will optimize for speed instead of size, and in that  
10 case, they may not choose the smallest permissible underlying type, but they cer-  
11 tainly want to be *able* to optimize for size. To make that possible, C++98 supports  
12 only `enum` definitions (where all enumerators are listed); `enum` declarations are  
13 not allowed. That makes it possible for compilers to select an underlying type for  
14 each `enum` prior to the `enum` being used.

15 But the inability to forward-declare `enums` has drawbacks. The most notable is  
16 probably the increase in compilation dependencies. Consider again the `Status`  
17 `enum`:

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

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

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

34 it's likely that the entire system will have to be recompiled, even if only a single  
35 subsystem—possibly only a single function!—uses the new enumerator. This is

1 the kind of thing that people *hate*. And it's the kind of thing that the ability to forward-declare `enums` in C++11 eliminates. For example, here's a perfectly valid declaration of a scoped `enum` and a function that takes one as a parameter:

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

6 The header containing these declarations requires no recompilation if `Status`'s  
7 definition is revised. Furthermore, if `Status` is modified (e.g., to add the `audited`  
8 enumerator), but `continueProcessing`'s behavior is unaffected (e.g., because  
9 `continueProcessing` doesn't use `audited`), `continueProcessing`'s  
10 implementation need not be recompiled, either.

11 But if compilers need to know the size of an `enum` before it's used, how can C++11's  
12 `enums` get away with forward declarations when C++98's `enums` can't? The answer  
13 is simple: the underlying type for a scoped `enum` is always known, and for  
14 unscoped `enums`, you can specify it.

15 By default, the underlying type for scoped `enums` is `int`:

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

17 If the default doesn't suit you, you can override it:

```
18 enum class Status: std::uint32_t; // underlying type for
19                                     // Status is std::uint32_t
20                                     // (from <cstdint>)
```

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

22 To specify the underlying type for an unscoped `enum`, you do the same thing as for  
23 a scoped `enum`, and the result may be forward-declared:

```
24 enum Color: std::uint8_t;      // fwd decl for unscoped enum;
25                                     // underlying type is
26                                     // std::uint8_t
```

27 Underlying type specifications can also go on an `enum`'s definition:

```
28 enum class Status: std::uint32_t { good = 0,
29                                     failed = 1,
30                                     incomplete = 100,
```

```
1                     corrupt = 200,
2                     audited = 500,
3                     indeterminate = 0xFFFFFFFF
4     };

5 In view of the fact that scoped enums avoid namespace pollution and aren't suscep-
6 tible to nonsensical implicit type conversions, it may surprise you to hear that
7 there's at least one situation where unscoped enums may be useful. That's when
8 referring to fields within C++11's std::tuples. For example, suppose we have a
9 tuple holding values for the name, email address, and reputation value for a user at
10 a social networking web site:
```

```
11 using UserInfo =           // type alias; see Item 9
12   std::tuple<std::string,      // name
13   std::string,            // email
14   std::size_t> ;        // reputation
```

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

```
18 UserInfo uInfo;           // object of tuple type
19 ...
20 auto val = std::get<1>(uInfo); // get value of field 1
```

```
21 As a programmer, you have a lot of stuff to keep track of. Should you really be
22 expected to remember that field 1 corresponds to the user's email address? I think
23 not. Using an unscoped enum to associate names with field numbers avoids the
24 need to:
```

```
25 enum UserInfoFields { uiName, uiEmail, uiReputation };
26 UserInfo uInfo;           // as before
27 ...
28 auto val = std::get<uiEmail>(uInfo); // ah, get value of
29                                // email field
```

```
30 What makes this work is the implicit conversion from UserInfoFields to
31 std::size_t, which is the type that std::get requires.
```

```
32 The corresponding code with scoped enums is substantially more verbose:
```

```
1 enum class UserInfoFields { uiName, uiEmail, uiReputation };
2 UserInfo uInfo; // as before
3 ...
4 auto val =
5     std::get<static_cast<std::size_t>(UserInfoFields::uiEmail)>
6     (uInfo);
7 The verbosity can be reduced by writing a function that takes an enumerator and
8 returns its corresponding std::size_t value, but it's a bit tricky. std::get is a
9 template, and the value you provide is a template argument (notice the use of an-
10 gle brackets, not parentheses), so the function that transforms an enumerator into
11 a std::size_t has to produce its result during compilation. As Item 15 explains,
12 that means it must be a constexpr function.
```

13 In fact, it should really be a `constexpr` function template, because it should work  
14 with any kind of `enum`. And if we're going to make that generalization, we should  
15 generalize the return type, too. Rather than returning `std::size_t`, we'll return  
16 the `enum`'s underlying type. It's available via the `std::underlying_type` type  
17 trait. (See Item 9 for information on type traits.) Finally, we'll declare it `noexcept`  
18 (see Item 14), because we know it will never yield an exception. The result is a  
19 function template `toUType` that takes an arbitrary enumerator and can return its  
20 value as a compile-time constant:

```
21 template<typename E>
22 constexpr typename std::underlying_type<E>::type
23     toUType(E enumerator) noexcept
24 {
25     return
26         static_cast<typename
27             std::underlying_type<E>::type>(enumerator);
28 }
```

29 In C++14, `toUType` can be simplified by replacing `typename`  
30 `std::underlying_type<E>::type` with the sleeker `std::underlying_type_t`  
31 (see Item 9):

```
32 template<typename E> // C++14
33 constexpr std::underlying_type_t<E>
34     toUType(E enumerator) noexcept
35 {
```

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

3 Either way, `toUType` permits us to access a field of the tuple like this:

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

5 It's still more to write than use of the unscoped `enum`, but it also avoids namespace  
6 pollution and inadvertent conversions involving enumerators. In many cases, you  
7 may decide that typing a few extra characters is a reasonable price to pay for the  
8 ability to avoid the pitfalls of an enum technology that dates to a time when the  
9 state 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 The situation arises only for the *special member functions*, i.e., the member func-  
26 tions that C++ automatically generates when they're needed. Item 17 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 largely devoted to common  
29 practices in C++98 that have been superseded by better practices in C++11, and in

1 C++98, if you want to suppress use of a member function, it's almost always the  
2 copy constructor, the assignment operator, or both.

3 The C++98 approach to preventing use of these functions is to declare them **private**  
4 and not define them. For example, near the base of the iostreams hierarchy  
5 in the C++ Standard Library is the class template **basic\_ios**. All **istream** and **os-  
6 tream** classes inherit (possibly indirectly) from this class. Copying **istreams** and  
7 **ostreams** is undesirable, because it's not really clear what such operations should  
8 do. An **istream** object, for example, represents a stream of input values, some of  
9 which may have already been read, and some of which will potentially be read lat-  
10 er. If an **istream** were to be copied, would that entail copying all the values that had  
11 already been read as well as all the values that would be read in the future? The  
12 easiest way to deal with such questions is to define them out of existence. Prohibit-  
13 ing the copying of streams does just that.

14 To render **istream** and **ostream** classes uncopyable, **basic\_ios** is specified in  
15 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  
25 failing to define them means that if code that still has access to them (i.e., member  
26 functions or **friends** of the class) uses them, linking will fail due to missing func-  
27 tion definitions.

28 In C++11, there's a better way to achieve essentially the same end: use “**= delete**”  
29 to mark the copy constructor and the copy assignment operator as **deleted func-  
30 tions**. 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 deleted status. When client code tries to use a deleted private func-
15 tion, 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 better 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 surprise you, but your surprise will dissipate once you recall  
12 that, given a choice between converting a `float` to an `int` or to a `double`, C++ pre-  
13 fers the conversion to `double`. Calling `isLucky` with a `float` will therefore call  
14 the `double` overload, not the `int` one. Well, it'll try to. The fact that that overload  
15 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 `void*` or a `char*` is invalid, it's probably  
11 also invalid to call it with a `const void*` or a `const char*`, so those instantiations  
12 will typically need to be deleted, too:

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

17 And if you really want to be thorough, you'll also delete the `const volatile`  
18 `void*` and `const volatile char*` overloads, and then you'll get to work on the  
19 overloads for pointers to the other standard character types: `std::wchar_t`,  
20 `std::char16_t`, and `std::char32_t`.

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

```
28 class Widget {
29 public:
30 ...
31     template<typename T>
32     void processPointer(T* ptr)
33     { ... }
```

```
1 private:  
2     template<> // error!  
3     void processPointer<void>(void*);  
4 };  
5  
6 The problem is that template specializations must be written at namespace scope,  
7 not class scope. This issue doesn't arise for deleted functions, because they don't  
8 need a different access level. They can be deleted outside the class (hence at  
namespace scope):
```

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

```
22 The truth is that the C++98 practice of declaring functions private and not defining  
23 them was really an attempt to achieve what C++11's deleted functions actually  
24 accomplish. As an emulation, the C++98 approach is not as good as the real thing.  
25 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.
```

## 26 Things to Remember

- 27 • Prefer deleted functions to private undefined ones.
- 28 • Any function may be deleted, including non-member functions and template  
29 instantiations.

## 30 Item 12: Declare overriding functions `override`.

```
31 The world of object-oriented programming in C++ revolves around classes, inheritance,  
32 and virtual functions. Among the most fundamental ideas in this world is  
33 that virtual function implementations in derived classes override the implementa-
```

1 tions of their base class counterparts. It's disheartening, then, to realize just how  
2 easily virtual function overriding can go wrong. It's almost as if this part of the lan-  
3 guage were designed with the idea that Murphy's Law wasn't just to be obeyed, it  
4 was to be honored.

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

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

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

- 26 • The base class function must be virtual.
- 27 • The base and derived function names must be identical (except in the case of  
28 destructors).
- 29 • The parameter types of the base and derived functions must be identical.
- 30 • The **constness** of the base and derived functions must be identical.
- 31 • The return types and exception specifications of the base and derived func-  
32 tions must be compatible.

33 To these constraints, which were also part of C++98, C++11 adds one more:

- 1   • The functions' *reference qualifiers* must be identical. Member function reference qualifiers are one of C++11's less-publicized features, so don't be surprised if you've never heard of them. They make it possible to limit use of a member function to lvalues only or to rvalues only. Member functions need not be virtual to use them:

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

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

26 All these requirements for overriding mean that small mistakes can make a big  
27 difference. Code containing overriding errors is typically valid, but its meaning  
28 isn't what you intended. You therefore can't rely on compilers notifying you if you  
29 do something wrong. For example, the following code is completely legal and, at  
30 first sight, looks reasonable, but it contains no virtual function overrides—not a  
31 single derived class function that is tied to a base class function. Can you identify  
32 the problem in each case, i.e., why each derived class function doesn't override the  
33 base class function with the same name?

```
1 class Base {
2 public:
3     virtual void mf1() const;
4     virtual void mf2(int x);
5     virtual void mf3() &;
6     void mf4() const;
7 };
8 class Derived: public Base {
9 public:
10    virtual void mf1();
11    virtual void mf2(unsigned int x);
12    virtual void mf3() &&;
13    void mf4() const;
14 };

```

15 Need some help?

- 16 • `mf1` is declared `const` in `Base`, but not in `Derived`.
- 17 • `mf2` takes an `int` in `Base`, but an `unsigned int` in `Derived`.
- 18 • `mf3` is lvalue-qualified in `Base`, but rvalue-qualified in `Derived`.
- 19 • `mf4` isn't declared `virtual` in `Base`.

20 You may think, "Hey, in practice, these things will elicit compiler warnings, so I  
21 don't need to worry." Maybe that's true. But maybe it's not. With two of the com-  
22 pilers I checked, the code was accepted without complaint, and that was with all  
23 warnings enabled. (Other compilers provided warnings about some of the issues,  
24 but not all of them.)

25 Because declaring derived class overrides is important to get right, but easy to get  
26 wrong, C++11 gives you a way to make explicit that a derived class function is sup-  
27 posed to override a base class version. Declare it `override`. Applying this to the  
28 example above would yield this derived class:

```
29 class Derived: public Base {
30 public:
31     virtual void mf1() override;
32     virtual void mf2(unsigned int x) override;
33     virtual void mf3() && override;
34     virtual void mf4() const override;
35 };

```

1 This won't compile, of course, because when written this way, compilers will  
2 kvetch about all the overriding-related problems. That's exactly what you want,  
3 and it's why you should declare all your overriding functions **override**.

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

```
6 class Base {  
7 public:  
8     virtual void mf1() const;  
9     virtual void mf2(int x);  
10    virtual void mf3() &;  
11    virtual void mf4() const;  
12 };  
  
13 class Derived: public Base {  
14 public:  
15     virtual void mf1() const override;  
16     virtual void mf2(int x) override;  
17     virtual void mf3() & override;  
18     void mf4() const override;           // adding "virtual" is OK,  
19                                // but not necessary  
20 };
```

20 Note that in this example, part of getting things to work involves declaring **mf4** virtual  
21 in **Base**. Most overriding-related errors occur in derived classes, but it's pos-  
22 sible for things to be incorrect in base classes, too.

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

1 C++ has always had keywords, but C++11 introduces two *contextual keywords*,  
2 `override` and `final`<sup>†</sup>. These keywords have the characteristic that they are re-  
3 served, but only in certain contexts. In the case of `override`, it has a reserved  
4 meaning only when it occurs at the end of a member function declaration. That  
5 means that if you have legacy code that already uses the name `override`, you  
6 don't need to change it for C++11:

```
7 class Warning {           // potential legacy class from C++98
8 public:
9 ...
10 void override();        // legal in both C++98 and C++11
11 ...                     // (with the same meaning)
12 };
```

13 That's all there is to say about `override`, but it's not all there is to say about mem-  
14 ber function reference qualifiers. I promised I'd provide more information on them  
15 later, and now it's later.

16 If we want to write a function that accepts only lvalue arguments, we declare a  
17 non-`const` lvalue reference parameter:

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

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

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

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

---

<sup>†</sup> Applying `final` to a virtual function prevents the function from being overridden in derived classes. `final` may also be applied to a class, in which case the class is prohibited from being used as a base class.

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

```
4 class Widget {  
5     public:  
6         using DataType = std::vector<double>;           // see Item 9 for  
7         ...                                               // info on "using"  
8         DataType& data() { return values; }  
9         ...  
10    private:  
11        DataType values;  
12    };
```

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

```
15 Widget w;  
16 ...  
17 auto vals1 = w.data();           // copy w.values into vals1  
18 The return type of Widget::data is an lvalue reference (a  
19 std::vector<double>&, to be precise), and because lvalue references are de-  
20 fined to be lvalues, we're initializing vals1 from an lvalue. vals1 is thus copy-  
21 constructed from w.values, just as the comment says.
```

22 Now suppose we have a factory function that creates `Widgets`,

```
23 Widget makeWidget();  
24 and we want to initialize a variable with the std::vector inside the Widget re-  
25 turned from makeWidget:
```

```
26 auto vals2 = makeWidget().data();   // copy values inside the  
27                                // Widget into vals2  
28 Again, Widgets::data returns an lvalue reference, and, again, the lvalue refer-  
29 ence is an lvalue, so, again, our new object (vals2) is copy-constructed from val-  
30 ues inside the Widget. This time, though, the Widget is the temporary object re-  
31 turned from makeWidget (i.e., an rvalue), so copying the std::vector inside it is  
32 a waste of time. It'd be preferable to move it, but, because data is returning an
```

1 lvalue reference, the rules of C++ require that compilers generate code for a copy.  
2 (There's some wiggle room for optimization through what is known as the "as if  
3 rule," but you'd be foolish to rely on your compilers finding a way to take ad-  
4 vantage of it.)

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

```
8 class Widget {
9 public:
10    using DataType = std::vector<double>;
11    ...
12    DataType& data() &           // for lvalue Widgets,
13    { return values; }          // return lvalue
14    DataType data() &&          // for rvalue Widgets,
15    { return std::move(values); } // return rvalue
16    ...
17 private:
18    DataType values;
19};
```

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

```
24 auto vals1 = w.data();           // calls lvalue overload for
25                                // Widget::data, copy-
26                                // constructs vals1
27 auto vals2 = makeWidget().data(); // calls rvalue overload for
28                                // Widget::data, move-
29                                // constructs vals2
```

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

1    **Things to Remember**

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

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

6    `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.

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

```
18 std::vector<int> values;
19 ...
20 std::vector<int>::iterator it =
21     std::find(values.begin(),values.end(), 1983);
22 values.insert(it, 1998);
```

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

```
27 typedef std::vector<int>::iterator IterT;           // type-
28 typedef std::vector<int>::const_iterator ConstIterT; // defn
29 std::vector<int> values;
```

```
1 ...
2 ConstIterT ci =
3     std::find(static_cast<ConstIterT>(values.begin()), // cast
4             static_cast<ConstIterT>(values.end()), // cast
5             1983);
6 values.insert(static_cast<IterT>(ci), 1998); // may not
7                                         // compile; see
8                                         // below
```

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

13 The casts in the call to `std::find` are present because `values` is a non-`const`  
14 container and in C++98, there was no simple way to get a `const_iterator` from a  
15 non-`const` container. The casts aren't strictly necessary, because it was possible to  
16 get `const_iterators` in other ways (e.g., you could bind `values` to a reference-  
17 to-`const` variable, then use that variable in place of `values` in your code), but one  
18 way or another, the process of getting `const_iterators` to elements of a non-  
19 `const` container involved some amount of contorting.

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

25 To be honest, the code I've shown might not compile, either, because there's no  
26 portable conversion from a `const_iterator` to an `iterator`, not even with a  
27 `static_cast`. Even the semantic sledgehammer known as `reinterpret_cast`  
28 can't do the job. (That's not a C++98 restriction. It's true in C++11, too.  
29 `const_iterators` simply don't convert to `iterators`, no matter how much it  
30 might seem like they should.) There are some portable ways to generate `itera-`  
31 `tors` that point where `const_iterators` do, but they're not obvious, not  
32 universally applicable, and not worth discussing in this book. Besides, I hope that  
33 by now my point is clear: `const_iterators` were so much trouble in C++98, they

1 were rarely worth the bother. At the end of the day, developers don't use `const`  
2 whenever *possible*, they use it whenever *practical*, and in C++98,  
3 `const_iterators` just weren't very practical.

4 All that changed in C++11. Now `const_iterators` are both easy to get and easy to  
5 use. The container member functions `cbegin` and `cend` produce  
6 `const_iterators`, even for non-`const` containers, and STL member functions  
7 that use iterators to identify positions (e.g., `insert` and `erase`) actually use  
8 `const_iterators`. Revising the original C++98 code that uses `iterators` to use  
9 `const_iterators` in C++11 is truly trivial:

```
10 std::vector<int> values;                                // as before
11 ...
12 auto it =                                                 // use cbegin
13     std::find(values.cbegin(), values.cend(), 1983); // and cend
14 values.insert(it, 1998);
```

15 Now *that's* code using `const_iterators` that's practical!

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

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

```
26 template<typename C, typename V>
27 void findAndInsert(C& container,           // in container, find
28                     const V& targetVal,    // first occurrence
29                     const V& insertVal)   // of targetVal, then
30 {
31     using std::cbegin;                    // insert insertVal
32     using std::cend;
```

```
1 auto it = std::find(cbegin(container), // non-member cbegin
2                     cend(container), // non-member cend
3                     targetVal);
4
5         container.insert(it, insertVal);
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? So was I. But follow the logic. This `cbegin` template accepts any type of  
21 argument representing a container-like data structure, `C`, and it accesses this ar-  
22 gument 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. The

1 elements of a `const` array are `const`, so the pointer that non-member `begin` re-  
2 turns for a `const` array is a pointer-to-`const`, and a pointer-to-`const` is, in fact, a  
3 `const_iterator` for an array. (For insight into how a template can be specialized  
4 for built-in arrays, consult Item 1's discussion of type deduction in templates that  
5 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: Declare functions `noexcept` if they won't emit ex- 16 ceptions.**

17 In C++98, exception specifications were rather temperamental beasts. You had to  
18 summarize the exception types a function might emit, so if the function's imple-  
19 mentation was modified, the exception specification might require revision, too.  
20 Changing an exception specification could break client code, because callers might  
21 be dependent on the original exception specification. Compilers typically offered  
22 no help in maintaining consistency among function implementations, exception  
23 specifications, and client code. Most programmers ultimately decided that C++98  
24 exception specifications weren't worth the trouble.

25 Interest in the idea of exception specifications remained strong, however, and as  
26 work on C++ progressed, a consensus emerged that the truly meaningful infor-  
27 mation about a function's exception-emitting behavior was whether it had any.  
28 Black or white, either a function might emit an exception or it guaranteed that it  
29 wouldn't. This maybe-or-never dichotomy forms the basis of C++11's exception  
30 specifications, which essentially replace C++98's. (C++98-style exception specifica-

1   tions remain valid, but they're deprecated.) In C++11, unconditional `noexcept` is  
2   for functions that guarantee they won't emit exceptions.

3   Whether a function should be so declared is a matter of interface design. The ex-  
4   ception-emitting behavior of a function is of key interest to clients. Callers can  
5   query a function's `noexcept` status, and the results of such a query can affect the  
6   exception safety or efficiency of the calling code. As such, whether a function is `no-`  
7   `except` is as important a piece of information as whether a member function is  
8   `const`. Failure to declare a function `noexcept` when you know that it won't emit  
9   an exception is simply poor interface specification.

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

```
15 int f(int x) throw();      // no exceptions from f: C++98 style  
16 int f(int x) noexcept;     // no exceptions from f: C++11 style
```

17   If, at run time, an exception leaves `f`, `f`'s exception specification is violated. With  
18   the C++98 exception specification, the call stack is unwound to `f`'s caller, and, after  
19   some actions not relevant here, program execution is terminated. With the C++11  
20   exception specification, runtime behavior is slightly different: the stack is only *pos-*  
21   *sibly* unwound before program execution is terminated.

22   The difference between unwinding the call stack and *possibly* unwinding it has a  
23   surprisingly large impact on code generation. In a `noexcept` function, optimizers  
24   need not keep the runtime stack in an unwindable state if an exception would  
25   propagate out of the function, nor must they ensure that objects in a `noexcept`  
26   function are destroyed in the inverse order of construction should an exception  
27   leave the function. Functions with "`throw()`" exception specifications lack such  
28   optimization flexibility, as do functions with no exception specification at all. The  
29   situation can be summarized this way:

```
30 RetType function(params) noexcept;      // most optimizable
```

```
1 RetType function(params) throw();           // less optimizable
2 RetType function(params);                   // less optimizable
3 This alone is sufficient reason to declare functions noexcept whenever you know
4 they won't produce exceptions.
5 For some functions, the case is even stronger. The move operations are the
6 preeminent example. Suppose you have a C++98 code base making use of a
7 std::vector<Widget>. Widgets are added to the std::vector from time to
8 time via push_back:
9 std::vector<Widget> vw;
10 ...
11 Widget w;
12 ...           // work with w
13 vw.push_back(w);           // add w to vw
14 ...
15 Assume this code works fine, and you have no interest in modifying it for C++11.
16 However, you do want to take advantage of the fact that C++11's move semantics
17 can improve the performance of legacy code when move-enabled types are in-
18 volved. You therefore ensure that Widget has move operations, either by writing
19 them yourself or by seeing to it that the conditions for their automatic generation
20 are fulfilled (see Item 17).
21 When a new element is added to a std::vector, it's possible that the
22 std::vector lacks space for it, i.e., that the std::vector's size is equal to its ca-
23 pacity. When that happens, the std::vector allocates a new, larger, chunk of
24 memory to hold its elements, and it transfers the elements from the existing chunk
25 of memory to the new one. In C++98, the transfer was accomplished by copying
26 each element from the old memory to the new memory, then destroying the ob-
27 jects in the old memory. This approach enabled push_back to offer the strong ex-
28 ception safety guarantee: if an exception was thrown during the copying of the el-
29 ements, the state of the std::vector remained unchanged, because none of the
```

1 elements in the old memory were destroyed until all elements had been success-  
2 fully copied into the new memory.

3 In C++11, a natural optimization would be to replace the copying of `std::vector`  
4 elements with moves. Unfortunately, doing this runs the risk of violating  
5 `push_back`'s exception safety guarantee. If  $n$  elements have been moved from the  
6 old memory and an exception is thrown moving element  $n+1$ , the `push_back` op-  
7 eration can't run to completion. But the original `std::vector` has been modified:  
8  $n$  of its elements have been moved from. Restoring their original state may not be  
9 possible, because attempting to move each object back into the original memory  
10 may itself yield an exception.

11 This is a serious problem, because the behavior of legacy code could depend on  
12 `push_back`'s strong exception safety guarantee. Therefore, C++11 implemen-  
13 tations can't silently replace copy operations inside `push_back` with moves unless  
14 it's known that the move operations won't emit exceptions. In that case, having  
15 moves replace copies would be safe, and the only side effect would be improved  
16 performance.

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

---

† 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 23), depending on whether the type's move constructor is `noexcept`. In turn, `std::move_if_noexcept` consults `std::is_nothrow_move_constructible`, and

1 swap functions comprise another case where noexcept is particularly desirable.  
2 swap is a key component of many STL algorithm implementations, and it's com-  
3 monly employed in copy assignment operators, too. Its widespread use renders  
4 the optimizations that noexcept affords especially worthwhile. Interestingly,  
5 whether swaps in the Standard Library are noexcept is sometimes dependent on  
6 whether user-defined swaps are noexcept. For example, the declarations for the  
7 Standard Library's swaps for arrays and std::pair are:

```
8 template <class T, size_t N>
9 void swap(T (&a)[N], // see
10          T (&b)[N]) noexcept(noexcept(swap(*a, *b))); // below
11
12 template <class T1, class T2>
13 struct pair {
14     ...
15     void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
16                               noexcept(swap(second, p.second)));
17     ...
18 };
```

18 These functions are *conditionally noexcept*: whether they are noexcept depends  
19 on whether the expressions inside the noexcept clauses are noexcept. Given two  
20 arrays of Widget, for example, swapping them is noexcept only if swapping indi-  
21 vidual elements in the arrays is noexcept, i.e., if swap for Widget is noexcept.  
22 The author of Widget's swap thus determines whether swapping arrays of Widget  
23 is noexcept. That, in turn, determines whether other swaps, such as the one for  
24 arrays of arrays of Widget, are noexcept. Similarly, whether swapping two  
25 std::pair objects containing Widgets is noexcept depends on whether swap for  
26 Widgets is noexcept. The fact that swapping higher-level data structures can  
27 generally be noexcept only if swapping their lower-level constituents is noexcept  
28 should motivate you to offer noexcept swap functions whenever you can.

29 By now, I hope you're excited about the optimization opportunities that noexcept  
30 affords. Alas, I must temper your enthusiasm. Optimization is important, but cor-

---

the value of this type trait (see Item 9) is set by compilers, based on whether the move constructor has a noexcept (or throw()) designation.

1   rectness is more important. I noted at the beginning of this Item that `noexcept` is  
2   part of a function's interface, so you should declare a function `noexcept` only if  
3   you are willing to commit to a `noexcept` implementation over the long term. If  
4   you declare a function `noexcept` and later regret that decision, your options are  
5   bleak. You can remove `noexcept` from the function's declaration (i.e., change its  
6   interface), thus running the risk of breaking client code. You can change the im-  
7   plementation such that an exception could escape, yet keep the original (now in-  
8   correct) exception specification. If you do that, your program will be terminated if  
9   an exception tries to leave the function. Or you can resign yourself to your existing  
10   implementation, abandoning whatever kindled your desire to change the imple-  
11   mentation in the first place. None of these options is appealing.

12   The fact of the matter is that most functions are *exception-neutral*. Such functions  
13   throw no exceptions themselves, but functions they call might emit one. When that  
14   happens, the exception-neutral function allows the emitted exception to pass  
15   through on its way to a handler further up the call chain. Exception-neutral func-  
16   tions are never `noexcept`, because they may emit such "just passing through" ex-  
17   ceptions. Most functions, therefore, quite properly lack the `noexcept` designation.

18   Some functions, however, have natural implementations that emit no exceptions,  
19   and for a few more—notably the move operations and `swap`—being `noexcept` can  
20   have such a significant payoff, it's worth implementing them in a `noexcept` man-  
21   ner if at all possible.<sup>†</sup> When you can honestly say that a function should never emit  
22   exceptions, you should definitely declare it `noexcept`.

---

<sup>†</sup> The interface specifications 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 aren't thrown, implementers often declare the operations `noexcept`, even though the Standard does not require them to do so.

1 Please note that I said some functions have *natural noexcept* implementations.  
2 Twisting a function's implementation to permit a `noexcept` declaration is the tail  
3 wagging the dog. Is putting the cart before the horse. Is not seeing the forest for  
4 the trees. Is...choose your favorite metaphor. If a straightforward function imple-  
5 mentation might yield exceptions (e.g., by invoking a function that might throw),  
6 the hoops you'll jump through to hide that from callers (e.g., catching all excep-  
7 tions and replacing them with status codes or special return values) will not only  
8 complicate your function's implementation, it will typically complicate code at call  
9 sites, too. For example, callers may have to check for status codes or special return  
10 values. The runtime cost of those complications (e.g., extra branches, larger func-  
11 tions that put more pressure on instruction caches, etc.) could exceed any speedup  
12 you'd hope to achieve via `noexcept`, plus you'd be saddled with source code that's  
13 more difficult to comprehend and maintain. That'd be poor software engineering.

14 For some functions, being `noexcept` is so important, they're that way by default.  
15 In C++98, it was considered bad style to permit the memory deallocation functions  
16 (i.e., `operator delete` and `operator delete[]`) and destructors to emit excep-  
17 tions, and in C++11, this style rule has been all but upgraded to a language rule. By  
18 default, all memory deallocation functions and all destructors—both user-defined  
19 and compiler-generated—are implicitly `noexcept`. There's thus no need to declare  
20 them `noexcept`. (Doing so doesn't hurt anything, it's just unconventional.) The  
21 only time a destructor is not implicitly `noexcept` is when a data member of the  
22 class (including inherited members and those contained inside other data mem-  
23 bers) is of a type that expressly states that its destructor may emit exceptions (e.g.,  
24 declares it "`noexcept(false)`"). Such destructors are uncommon. There are none  
25 in the Standard Library, and if the destructor for an object being used by the  
26 Standard Library (e.g., because it's in a container or was passed to an algorithm)  
27 emits an exception, the behavior of the program is undefined.

28 It's worth noting that some library interface designers distinguish functions with  
29 *wide contracts* from those with *narrow contracts*. A function with a wide contract  
30 has no preconditions. Such a function may be called regardless of the state of the

1 program, and it imposes no constraints on the arguments that callers pass it.<sup>†</sup>  
2 Functions with wide contracts never exhibit undefined behavior.

3 Functions without wide contracts have narrow contracts. For such functions, if a  
4 precondition is violated, results are undefined.

5 If you're writing a function with a wide contract and you know it won't emit exceptions,  
6 following the advice of this Item and declaring it `noexcept` is easy. For functions  
7 with narrow contracts, the situation is trickier. For example, suppose you're  
8 writing a function `f` taking a `std::string` parameter, and suppose `f`'s natural  
9 implementation never yields an exception. That suggests that `f` should be declared  
10 `noexcept`.

11 Now suppose that `f` has a precondition: the length of its `std::string` parameter  
12 doesn't exceed 32 characters. If `f` were to be called with a `std::string` whose  
13 length is greater than 32, behavior would be undefined, because a precondition  
14 violation *by definition* results in undefined behavior. `f` is under no obligation to  
15 check this precondition, because functions may assume that their preconditions  
16 are satisfied. (Callers are responsible for ensuring that such assumptions are val-  
17 id.) Even with a precondition, then, declaring `f noexcept` seems appropriate:

```
18 void f(const std::string& s) noexcept;    // precondition:  
19                                // s.length() <= 32
```

20 But suppose that `f`'s implementer chooses to check for precondition violations, at  
21 least in debug builds. Checking isn't required, but it's also not forbidden, and  
22 checking the precondition could be useful during system testing. Debugging an ex-  
23 ception that's been thrown is generally easier than trying to track down the cause  
24 of undefined behavior. But how should a precondition violation be reported such

---

<sup>†</sup> “Regardless of the state of the program” and “no constraints” doesn't legitimize programs whose behavior is already undefined. For example, `std::vector::size` has a wide contract, but that doesn't require that it behave reasonably if you invoke it on a random chunk of memory that you've cast to a `std::vector`. The result of the cast is undefined, so there are no behavioral guarantees beyond that point.

1 that a test harness could detect it? A straightforward approach would be to throw  
2 a “precondition was violated” exception, but if `f` is declared `noexcept`, that would  
3 be impossible; throwing an exception would lead to program termination. For this  
4 reason, library designers who distinguish wide from narrow contracts generally  
5 reserve `noexcept` for functions with wide contracts.

6 As a final point, let me elaborate on my earlier observation that compilers typically  
7 offer no help in identifying inconsistencies between function implementations and  
8 their exception specifications. Consider this code, which is perfectly legal:

```
9 void setup();           // functions defined elsewhere
10 void cleanup();

11 void doWork() noexcept
12 {
13     setup();           // set up work to be done
14     ...                 // do the actual work
15     cleanup();          // perform cleanup actions
16 }
```

17 Here, `doWork` is declared `noexcept`, even though it calls the non-`noexcept` func-  
18 tions `setup` and `cleanup`. This seems contradictory, but it could be that `setup`  
19 and `cleanup` document that they never emit exceptions, even though they’re not  
20 declared that way. There could be good reasons for their non-`noexcept` declara-  
21 tions. For example, they might be part of a library written in C. (Even functions  
22 from the C Standard Library that have been moved into the `std` namespace lack  
23 exception specifications, e.g., `std::strlen` isn’t declared `noexcept`.) Or they  
24 could be part of a C++98 library that decided not to use C++98 exception specifica-  
25 tions and hasn’t yet been revised for C++11.

26 Because there are legitimate reasons for `noexcept` functions to rely on code lack-  
27 ing the `noexcept` guarantee, C++ permits such code, and compilers generally don’t  
28 issue warnings about it.

## 29 **Things to Remember**

- 30 ♦ `noexcept` is part of a function’s interface, and that means that callers may de-  
31 pend on it.

- 1   ♦ noexcept functions are more optimizable than non-noexcept functions.
- 2   ♦ noexcept is particularly valuable for the move operations, swap, memory
- 3   deallocation functions, and destructors.
- 4   ♦ Most functions are exception-neutral rather than noexcept.

## 5   **Item 15: Use constexpr whenever possible.**

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

11   Conceptually, constexpr indicates a value that's not only constant, it's known  
12   during compilation. The concept is only part of the story, though, because when  
13   constexpr is applied to functions, things are more nuanced than this suggests.  
14   Lest I ruin the surprise ending, for now I'll just say that you can't assume that the  
15   results of constexpr functions are const, nor can you take for granted that their  
16   values are known during compilation. Perhaps most intriguingly, these things are  
17   *features*. It's *good* that constexpr functions need not produce results that are  
18   const or known during compilation!

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

25   Values known during compilation are privileged. They may be placed in read-only  
26   memory, for example, and, especially for developers of embedded systems, this  
27   can be a feature of considerable importance. Of broader applicability is that inte-  
28   gral values that are constant and known during compilation can be used in con-  
29   texts where C++ requires an *integral constant expression*. Such contexts include  
30   specification of array sizes, integral template arguments (including lengths of

1 `std::array` objects), enumerator values, alignment specifiers, and more. If you  
2 want to use a variable for these kinds of things, you certainly want to declare it  
3 `constexpr`, because then compilers will ensure that it has a compile-time value:

```
4 int sz;                                // non-constexpr variable
5 ...
6 constexpr auto arraySize1 = sz;          // error! sz's value not
7                                         // known at compilation
8 std::array<int, sz> data1;             // error! same problem
9 constexpr auto arraySize2 = 10;          // fine, 10 is a
10                                         // compile-time constant
11 std::array<int, arraySize2> data2;      // fine, arraySize
12                                         // is constexpr
```

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

```
15 int sz;                                // as before
16 ...
17 const auto arraySize = sz;              // fine, arraySize is
18                                         // const copy of sz
19 std::array<int, arraySize> data;        // error! arraySize's value
20                                         // not known at compilation
```

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

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

1     • `constexpr` functions can be used in contexts that demand compile-time constants. If the values of the arguments you pass to a `constexpr` function in such  
2         a context are known during compilation, the result will be computed during  
3         compilation. If any of the arguments' values is not known during compilation,  
4         your code will be rejected.

5  
6     • When a `constexpr` function is called with one or more values that are not  
7         known during compilation, it acts like a normal function, computing its result  
8         at run time. This means you don't need two functions to perform the same op-  
9         eration, one for compile-time constants and one for all other values. The `con-`  
10         `stexpr` function does it all.

11 Suppose we need a data structure to hold the results of an experiment that can be  
12 run in a variety of ways. For example, the lighting level can be high, low, or off dur-  
13 ing the course of the experiment, as can the fan speed and the temperature, etc. If  
14 there are  $n$  environmental conditions relevant to the experiment, each of which  
15 has three possible states, the number of combinations is  $3^n$ . Storing experimental  
16 results for all combinations of conditions thus requires a data structure with  
17 enough room for  $3^n$  values. Assuming each result is an `int` and that  $n$  is known (or  
18 can be computed) during compilation, a `std::array` could be a reasonable data  
19 structure choice. But we'd need a way to compute  $3^n$  during compilation. The C++  
20 Standard Library provides `std::pow`, which is the mathematical functionality we  
21 need, but, for our purposes, there are two problems with it. First, `std::pow` works  
22 on floating point types, and we need an integral result. Second, `std::pow` isn't  
23 `constexpr` (i.e., isn't guaranteed to return a compile-time result when called with  
24 compile-time values), so we can't use it to specify the size of a `std::array`.

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

```
27 constexpr // pow's a constexpr func
28     int pow(int base, int exp) noexcept // that never throws
29 {
30     ...
31 } // impl is below
32 constexpr auto numConds = 5; // # of conditions
```

```
1 std::array<int, pow(3, numConds)> results; // results has
2 // 3^numConds
3 // elements

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

```
10 auto base = readFromDB("base"); // get these values
11 auto exp = readFromDB("exponent"); // at run time

12 auto baseToExp = pow(base, exp); // call pow function
13 // at run time
```

14 Because **constexpr** functions must be able to return compile-time results when  
15 called with compile-time values, restrictions are imposed on their implementation.  
16 The restrictions differ between C++11 and C++14.

17 In C++11, **constexpr** functions may contain no more than a single executable  
18 statement: a **return**. That sounds more limiting than it is, because two tricks can  
19 be used to extend the expressiveness of **constexpr** functions beyond what you  
20 might think. First, the conditional "?:" operator can be used in place of **if-else**  
21 statements, and second, recursion can be used instead of loops. **pow** can therefore  
22 be implemented like this:

```
23 constexpr int pow(int base, int exp) noexcept
24 {
25     return (exp == 0 ? 1 : base * pow(base, exp - 1));
26 }
```

27 This works, but it's hard to imagine that anybody except a hard-core functional  
28 programmer would consider it pretty. In C++14, the restrictions on **constexpr**  
29 functions are substantially looser, so the following implementation becomes pos-
30 sible:

```
31 constexpr int pow(int base, int exp) noexcept // C++14
32 {
33     if (exp == 0) return 1;
```

```
1     auto result = base;
2     for (int i = 1; i < exp; ++i) result *= base;
3
4 }
```

5 `constexpr` functions are limited to taking and returning *literal types*, which essen-  
6 tially means types that can have values determined during compilation. All built-in  
7 types except `void` qualify, but user-defined types may be literal, too, because con-  
8 structors and other member functions may be `constexpr`:

```
9 class Point {
10 public:
11     constexpr Point(double xVal, double yVal) noexcept
12     : x(xVal), y(yVal)
13     {}
14
15     constexpr double xValue() const noexcept { return x; }
16     constexpr double yValue() const noexcept { return y; }
17
18     void setX(double newX) noexcept { x = newX; }
19     void setY(double newY) noexcept { y = newY; }
20 };
```

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

```
25 constexpr Point p1{ 1, 1 };           // fine, "runs" constexpr
26                                         // ctor during compilation
27 constexpr Point p2{ 4, 2 };           // also fine
```

28 Similarly, the getters `xValue` and `yValue` can be `constexpr`, because if they're  
29 invoked on a `Point` object with a value known during compilation (e.g., a `con-`  
30 `stexpr` `Point` object), the values of the data members `x` and `y` can be known dur-  
31 ing compilation. That makes it possible to write `constexpr` functions that call  
32 `Point`'s getters and to initialize `constexpr` objects with the results of such func-  
33 tions:

```

1 constexpr
2     Point midpoint(const Point& p1, const Point& p2) noexcept
3 {
4     return { (p1.xValue() + p2.xValue()) / 2,      // call constexpr
5             (p1.yValue() + p2.yValue()) / 2 }; // member funcs
6 }
7 constexpr auto mid = midpoint(p1, p2);           // init constexpr
8                                         // object w/result of
9                                         // constexpr function

```

10 This is very exciting. It means that the object `mid`, though its initialization involves  
 11 calls to constructors, getters, and a non-member function, can be created in read-  
 12 only memory! It means you could use an expression like `mid.xValue() * 10` in an  
 13 argument to a template or in an expression specifying the value of an enumerator!<sup>†</sup>  
 14 It means that the traditionally fairly strict line between work done during compila-  
 15 tion and work done at run time begins to blur, and some computations traditional-  
 16 ly done at run time can migrate to compile time. The more code taking part in the  
 17 migration, the faster your software will run. (Compilation may take longer, howev-  
 18 er.)

19 The setter functions `setX` and `setY` can't be declared `constexpr`, because they  
 20 have `void` return types, and `void` return types are not permitted for `constexpr`  
 21 functions, not even in C++14. (For a detailed treatment of the restrictions on `con-`  
 22 `stexpr` function implementations, consult your favorite references for C++11 and  
 23 C++14, bearing in mind that C++11 imposes more constraints than C++14.)

24 The advice of this Item is to use `constexpr` whenever possible, and by now I hope  
 25 it's clear why: both `constexpr` objects and `constexpr` functions can be employed  
 26 in a wider range of contexts than non-`constexpr` objects and functions. By using

<sup>†</sup> Because `Point::xValue` returns `double`, the type of `mid.xValue() * 10` is also `double`.

Floating point types can't be used to instantiate templates or to specify enumerator values, but they can be used as part of larger expressions that yield integral types. For example, `static_cast<int>(mid.xValue() * 10)` could be used to instantiate a template or to specify an enumerator value.

1   `constexpr` whenever possible, you maximize the range of situations in which  
2   your objects and functions may be used.

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

13   **Things to Remember**

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

21   **Item 16: Make `const` member functions thread-safe.**

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

```
27   class Polynomial {  
28   public:  
29     using RootsType =               // data structure holding values
```

```
1     std::vector<double>;      // where polynomial evals to zero
2     ...                         // (see Item 9 for info on "using")
3 
4 RootsType roots() const;
5 ...
6 };
```

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

```
11 class Polynomial {
12 public:
13     using RootsType = std::vector<double>;
14     RootsType roots() const
15     {
16         if (!rootsAreValid) {           // if cache not valid
17             ...
18             // compute roots,
19             // store them in rootVals
20         rootsAreValid = true;
21     }
22     return rootVals;
23 }
24 private:
25     mutable bool rootsAreValid{ false };    // see Item 7 for info
26     mutable RootsType rootVals{};           // on initializers
27 };
```

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

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

```
32 Polynomial p;
33 ...
34
```

```

1  /*----- Thread 1 ----- */      /*----- Thread 2 ----- */
2  auto rootsOfP = p.roots();      auto valsGivingZero = p.roots();
3
4  This client code is perfectly reasonable. roots is a const member function, and
5  that means it represents a read operation. Having multiple threads perform a read
6  operation without synchronization is safe. At least it's supposed to be. In this case,
7  it's not, because inside roots, one or both of these threads might try to modify the
8  data members rootsAreValid and rootVals. That means that this code could
9  have different threads reading and writing the same memory without synchroni-
10 The problem is that roots is declared const, but it's not thread-safe. The const
11 declaration is as correct in C++11 as it would be in C++98 (retrieving the roots of a
12 polynomial doesn't change the value of the polynomial), so what requires rectifica-
13 tion is the lack of thread safety.
14 The easiest way to address the issue is the usual one: employ a mutex:
15
16 class Polynomial {
17 public:
18     using RootsType = std::vector<double>;
19
20     RootsType roots() const
21     {
22         std::lock_guard<std::mutex> g(m);           // Lock mutex
23         if (!rootsAreValid) {                      // if cache not valid
24             ...
25             // compute/store roots
26             rootsAreValid = true;
27         }
28         return rootVals;                         // release mutex
29     }
30
31 private:
32     mutable std::mutex m;
33     mutable bool rootsAreValid{ false };
34     mutable RootsType rootVals{};
35 };

```

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

4 It's worth noting that because `std::mutex` is a *move-only type* (i.e., a type that can  
5 be moved, but not copied), a side effect of adding `m` to `Polynomial` is that `Polyno-`  
6 `mial` loses the ability to be copied. It can still be moved, however.

7 In some situations, a mutex is overkill. For example, if all you're doing is counting  
8 how many times a member function is called, a `std::atomic` counter (i.e., one  
9 where other threads are guaranteed to see its operations occur indivisibly—see  
10 Item 40) will often be a less expensive way to go. (Whether it actually is less ex-  
11 pensive depends on the hardware you're running on and the implementation of  
12 mutexes in your Standard Library.) Here's how you can employ a `std::atomic` to  
13 count calls:

```
14 class Point {                                     // 2D point
15 public:
16 ...
17     double distanceFromOrigin() const noexcept    // see Item 14
18     {                                                 // for noexcept
19         ++callCount;                           // atomic increment
20         return std::sqrt((x * x) + (y * y));
21     }
22 private:
23     mutable std::atomic<unsigned> callCount{ 0 };
24     double x, y;
25 };
```

26 Like `std::mutexes`, `std::atomics` are move-only types, so the existence of  
27 `callCount` in `Point` means that `Point` is also move-only.

28 Because operations on `std::atomic` variables are often less expensive than  
29 mutex acquisition and release, you may be tempted to lean on `std::atomics`  
30 more heavily than you should. For example, in a class caching an expensive-to-  
31 compute `int`, you might try to use a pair of `std::atomic` variables instead of a  
32 mutex:

```

1  class Widget {
2  public:
3  ...
4
5      int magicValue() const
6      {
7          if (cacheValid) return cachedValue;
8          else {
9              auto val1 = expensiveComputation1();
10             auto val2 = expensiveComputation2();
11             cachedValue = val1 + val2;           // uh oh, part 1
12             cacheValid = true;                  // uh oh, part 2
13         }
14     }
15
16     private:
17     mutable std::atomic<bool> cacheValid{ false };
18     mutable std::atomic<int> cachedValue;
19 
```

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

- 21
- A thread calls `Widget::magicValue`, sees `cacheValid` as `false`, performs the two expensive computations and assigns their sum to `cachedValue`.

22

  - At that point, a second thread calls `Widget::magicValue`, also sees `cacheValid` as `false` and thus carries out the same expensive computations that the first thread has just finished. (This “second thread” may in fact be *several* other threads.)

26 Such behavior is contrary to the goal of caching. Reversing the order of the assignments to `cachedValue` and `CacheValid` eliminates that problem, but the result is even worse:

```

29 class Widget {
30 public:
31 ...
32
33     int magicValue() const
34     {
35         if (cacheValid) return cachedValue;
36         else {
37             auto val1 = expensiveComputation1();
38             auto val2 = expensiveComputation2();
39             cacheValid = true;                  // uh oh, part 1
40         }
41     }
42 
```

```
1     return cachedValue = val1 + val2;           // uh oh, part 2
2 }
3 }
4 ...
5 };
6
```

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

- ```
7 • One thread calls Widget::magicValue and executes through the point where
8   cacheValid is set to true.
9
10 • At that moment, a second thread calls Widget::magicValue and checks ca-
11   cheValid. Seeing it true, the thread returns cachedValue, even though the
12   first thread has not yet made an assignment to it. The returned value is there-
13   fore incorrect.
```

```
13 There's a lesson here. For a single variable or memory location requiring synchro-
14 nization, use of a std::atomic is adequate, but once you get to two or more vari-
15 ables or memory locations that require manipulation as a unit, you should reach
16 for a mutex. For Widget::magicValue, that would look like this:
```

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

```
1   mutable bool cacheValid{ false };           // no longer atomic
2 };
3 Now, this Item is predicated on the assumption that multiple threads may simulta-
4 neously execute a const member function on an object. If you're writing a const
5 member function where that's not the case—where you can guarantee that there
6 will never be more than one thread executing that member function on an object—the
7 thread safety of the function is immaterial. For example, it's unimportant
8 whether member functions of classes designed for exclusively single-threaded use
9 are thread-safe. In such cases, you can avoid the costs associated with mutexes and
10 std::atomics, as well as the side-effect of their rendering the classes containing
11 them move-only. However, such threading-free scenarios are increasingly un-
12 common, and they're likely to become rarer still. The safe bet is that const mem-
13 ber functions will be subject to concurrent execution, and that's why you should
14 ensure that your const member functions are thread-safe.
```

## 15 Things to Remember

- 16 • Make `const` member functions thread safe unless you're *certain* they'll never  
17 be used in a concurrent context.
- 18 • Use of `std::atomic` variables may offer better performance than a mutex, but  
19 they're suited for manipulation of only a single variable or memory location.

## 20 Item 17: Understand special member function generation.

21 In official C++ parlance, the “special member functions” are the ones that C++ is  
22 willing to generate on its own. C++98 has four such functions: the default construc-
23 tor, the destructor, the copy constructor, and the copy assignment operator.  
24 There’s fine print, of course. These functions are generated only if they’re needed,  
25 i.e., if some code uses them without their being expressly declared in the class. A  
26 default constructor is generated only if the class declares no constructors at all.  
27 (This prevents compilers from creating a default constructor for a class where  
28 you’ve specified that constructor arguments are required.) Generated special  
29 member functions are implicitly public and `inline`, and they’re nonvirtual unless  
30 the function in question is a destructor in a derived class inheriting from a base

1 class with a virtual destructor. In that case, the compiler-generated destructor for  
2 the derived class is also virtual.

3 But you already know these things. Yes, yes, ancient history: Mesopotamia, the  
4 Shang dynasty, FORTRAN, C++98. But times have changed, and the rules for special  
5 member function generation in C++ have changed with them. It's important to be  
6 aware of the new rules, because few things are as central to effective C++ pro-  
7 gramming as knowing when compilers silently insert member functions into your  
8 classes.

9 As of C++11, the special member functions club has two more inductees: the move  
10 constructor and the move assignment operator. Their signatures are:

```
11 class Widget {  
12 public:  
13 ...  
14     Widget(Widget&& rhs);           // move constructor  
15     Widget& operator=(Widget&& rhs); // move assignment operator  
16 ...  
17 };
```

18 The rules governing their generation and behavior are analogous to those for their  
19 copying siblings. The move operations are generated only if they're needed, and if  
20 they are generated, they perform "memberwise moves" on the non-static data  
21 members of the class. That means that the move constructor move-constructs each  
22 non-static data member of the class from the corresponding member of its param-  
23 eter `rhs`, and the move assignment operator move-assigns each non-static data  
24 member from its parameter. The move constructor also move-constructs its base  
25 class parts (if there are any), and the move assignment operator move-assigns its  
26 base class parts.

27 Now, when I refer to a move operation move-constructing or move-assigning a  
28 data member or base class, there is no guarantee that a move will actually take  
29 place. "Memberwise moves" are, in reality, more like memberwise move *requests*,  
30 because types that aren't *move-enabled* (i.e., that offer no special support for move  
31 operations, e.g., most C++98 legacy classes) will be "moved" via their copy opera-  
32 tions. The heart of each memberwise "move" is application of `std::move` to the  
33 object to be moved from, and the result is used during function overload resolu-

1 tion to determine whether a move or a copy should be performed. Item 23 covers  
2 this process in detail. For this Item, simply remember that a memberwise move  
3 consists of move operations on data members and base classes that support move  
4 operations, but a copy operation for those that don't.

5 As is the case with the copy operations, the move operations aren't generated if  
6 you declare them yourself. However, the precise conditions under which they are  
7 generated differ a bit from those for the copy operations.

8 The two copy operations are independent: declaring one doesn't prevent compil-  
9 ers from generating the other. So if you declare a copy constructor, but no copy  
10 assignment operator, then write code that requires copy assignment, compilers  
11 will generate the copy assignment operator for you. Similarly, if you declare a copy  
12 assignment operator, but no copy constructor, yet your code requires copy con-  
13 struction, compilers will generate the copy constructor for you. That was true in  
14 C++98, and it's still true in C++11.

15 The two move operations are not independent. If you declare either, that prevents  
16 compilers from generating the other. The rationale is that if you declare, say, a  
17 move constructor for your class, you're indicating that there's something about  
18 how move construction should be implemented that's different from the default  
19 memberwise move that compilers would generate. And if there's something wrong  
20 with memberwise move construction, there'd probably be something wrong with  
21 memberwise move assignment, too. So declaring a move constructor prevents a  
22 move assignment operator from being generated, and declaring a move assign-  
23 ment operator prevents compilers from generating a move constructor.

24 Furthermore, move operations won't be generated for any class that explicitly de-  
25 clares a copy operation. The justification is that declaring a copy operation (con-  
26 struction or assignment) indicates that the normal approach to copying an object  
27 (memberwise copy) isn't appropriate for the class, and compilers figure that if  
28 memberwise copy isn't appropriate for the copy operations, memberwise move  
29 probably isn't appropriate for the move operations.

30 This goes in the other direction, too. Declaring a move operation (construction or  
31 assignment) in a class causes compilers to disable the copy operations. (The copy

operations are disabled by deleting them—see Item 11). After all, if memberwise move isn't the proper way to move an object, there's no reason to expect that memberwise copy is the proper way to copy it. This may sound like it could break C++98 code, because the conditions under which the copy operations are enabled are more constrained in C++11 than in C++98, but this is not the case. C++98 code can't have move operations, because there was no such thing as "moving" objects in C++98. The only way a legacy class can have user-declared move operations is if they were added for C++11, and classes that are modified to take advantage of move semantics have to play by the C++11 rules for special member function generation.

Perhaps you've heard of a guideline known as the *Rule of Three*. The Rule of Three emerged fairly early in the C++ era (the early 1990s), and it states that if you declare any of a copy constructor, copy assignment operator, or destructor, you should declare all three. It grew out of the observation that the need to take over the meaning of a copy operation almost always stemmed from the class performing some kind of resource management, and that almost always implied that (1) whatever resource management was being done in one copy operation probably needed to be done in the other copy operation and (2) the class destructor would also be participating in management of the resource (usually releasing it). The classic resource to be managed was memory, and this is why all Standard Library classes that manage memory (e.g., the STL containers that perform dynamic memory management) all declare "the big three:" both copy operations and a destructor.

A consequence of the Rule of Three is that the presence of a user-declared destructor indicates that simple memberwise copy is unlikely to be appropriate for the copying operations in the class. That, in turn, suggests that if a class declares a destructor, the copy operations probably shouldn't be automatically generated, because they wouldn't do the right thing. At the time C++98 was adopted, the significance of this line of reasoning was not fully appreciated, so in C++98, the existence of a user-declared destructor had no impact on compilers' willingness to generate copy operations. That continues to be the case in C++11, but only because restrict-

1 ing the conditions under which the copy operations are generated would break too  
2 much legacy code.

3 The reasoning behind the Rule of Three remains valid, however, and combined  
4 with the observation that declaration of a copy operation precludes the implicit  
5 generation of the move operations, C++11 does *not* generate move operations for a  
6 class with a user-declared destructor.

7 So move operations are generated for classes (when needed) only if these three  
8 things are true:

9 • No copy operations are declared in the class.

10 • No move operations are declared in the class.

11 • No destructor is declared in the class.

12 At some point, analogous rules may be extended to the copy operations, because  
13 C++11 deprecates the automatic generation of copy operations for classes declar-  
14 ing copy operations or a destructor. This means that if you have code that depends  
15 on the generation of copy operations in classes declaring a destructor or one of the  
16 copy operations, you should consider upgrading these classes to eliminate the de-  
17 pendence. Provided the behavior of the compiler-generated functions is correct  
18 (i.e., if memberwise copying of the class's non-static data members is what you  
19 want), your job is 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; // default copy ctor  
26   // behavior is OK  
27     Widget& operator=(const Widget&) = default; // default copy assign  
28   // behavior is OK  
29     ...  
30 };
```

31 This approach is often useful in polymorphic base classes, i.e., classes defining in-  
32 terfaces through which derived class objects are manipulated. Polymorphic base

1 classes normally have virtual destructors, because if they don't, some operations  
2 (e.g., the use of `delete` or `typeid` on a derived class object through a base class  
3 pointer or reference) yield undefined or misleading results. Unless a class inherits  
4 a destructor that's already virtual, the only way to make a destructor virtual is to  
5 explicitly declare it that way. Often, the default implementation would be correct,  
6 and “`= default`” is a good way to express that. However, a user-declared destruc-  
7 tor suppresses generation of the move operations, so if movability is to be sup-  
8 ported, “`= default`” often finds a second application. Declaring the move opera-  
9 tions disables the copy operations, so if copyability is also desired, one more round  
10 of “`= default`” does the job:

```
11 class Base {  
12 public:  
13     virtual ~Base() = default;           // make dtor virtual  
14     Base(Base&&) = default;           // support moving  
15     Base& operator=(Base&&) = default;  
16     Base(const Base&) = default;       // support copying  
17     Base& operator=(const Base&) = default;  
18     ...  
19 };
```

20 In fact, even if you have a class where compilers are willing to generate the copy  
21 and move operations and where the generated functions would behave as you  
22 want, you may choose to adopt a policy of declaring them yourself and using “`=  
23 default`” for their definitions. It’s more work, but it makes your intentions clear-  
24 er, and it can help you side-step some fairly subtle bugs. For example, suppose you  
25 have a class representing a string table, i.e., a data structure that permits fast  
26 lookups of string values via an integer ID:

```
27 class StringTable {  
28 public:  
29     StringTable() {}  
30     ...           // functions for insertion, erasure, lookup,  
31                  // etc., but no copy/move/dtor functionality  
32 private:  
33     std::map<int, std::string> values;  
34 };
```

1 Assuming that the class declares no copy operations, no move operations, and no  
2 destructor, compilers will automatically generate these functions if they are used.  
3 That's very convenient.

4 But suppose that sometime later, it's decided that logging the default construction  
5 and the destruction of such objects would be useful. Adding that functionality is  
6 easy:

```
7 class StringTable {  
8 public:  
9     StringTable()  
10    { makeLogEntry("Creating StringTable object"); }      // added  
11    ~StringTable()   // also  
12    { makeLogEntry("Destroying StringTable object"); }     // added  
13    ...   // other funcs as before  
14 private:  
15     std::map<int, std::string> values;      // as before  
16 };
```

17 This looks reasonable, but declaring a destructor has a potentially significant side  
18 effect: it prevents the move operations from being generated. However, creation of  
19 the class's copy operations is unaffected. The code is therefore likely to compile,  
20 run, and pass its functional testing. That includes testing its move functionality,  
21 because even though this class is no longer move-enabled, requests to move it will  
22 compile and run. Such requests will, as noted earlier in this Item, cause copies to  
23 be made. Which means that code "moving" `StringTable` objects actually copies  
24 them, i.e., copies the underlying `std::map<int, std::string>` objects. And  
25 copying a `std::map<int, std::string>` is likely to be *orders of magnitude*  
26 slower than moving it. The simple act of adding a destructor to the class could  
27 thereby have introduced a significant performance problem! Had the copy and  
28 move operations been explicitly defined using "`= default`", the problem would  
29 not have arisen.

30 Now, having endured my endless blathering about the rules governing the copy  
31 and move operations in C++11, you may wonder when I'll turn my attention to the  
32 two other special member functions, the default constructor and the destructor.

1 That time is now, but only for this sentence, because almost nothing has changed  
2 for these member functions: the rules in C++11 are nearly the same as in C++98.

3 The C++11 rules governing the special member functions are thus:

4 • **Default constructor**: Same rules as C++98. Generated only if the class con-  
5 tains no user-declared constructors.

6 • **Destructor**: Essentially same rules as C++98; sole difference is that destruc-  
7 tors are `noexcept` by default (see Item 14). As in C++98, virtual only if a base  
8 class destructor is virtual.

9 • **Copy constructor**: Same runtime behavior as C++98: memberwise copy con-  
10 struction of non-static data members. Generated only if the class lacks a user-  
11 declared copy constructor. Deleted if the class declares a move operation. Gen-  
12 eration of this function in a class with a user-declared copy assignment opera-  
13 tor or destructor is deprecated.

14 • **Copy assignment operator**: Same runtime behavior as C++98: memberwise  
15 copy assignment of non-static data members. Generated only if the class lacks  
16 a user-declared copy assignment operator. Deleted if the class declares a move  
17 operation. Generation of this function in a class with a user-declared copy con-  
18 structor or destructor is deprecated.

19 • **Move constructor** and **move assignment operator**: Each performs mem-  
20 berwise moving of non-static data members. Generated only if the class con-  
21 tains no user-declared copy operations, move operations, or destructor.

22 Note that there's nothing in the rules about the existence of a member function  
23 *template* preventing compilers from generating the special member functions.  
24 That means that if `Widget` looks like this,

```
25 class Widget {  
26     ...  
27     template<typename T>           // construct Widget  
28     Widget(const T& rhs);          // from anything  
29     template<typename T>           // assign Widget  
30     Widget& operator=(const T& rhs); // from anything
```

```
1     ...
2 };
3 compilers will still generate copy and move operations for Widget (assuming the
4 usual conditions governing their generation are fulfilled), even though these tem-
5 plates could be instantiated to produce the signatures for the copy constructor and
6 copy assignment operator. (That would be the case when T is Widget.) In all like-
7 lihood, this will strike you as an edge case barely worth acknowledging, but there's
8 a reason I'm mentioning it. Item 26 demonstrates that it can have important con-
9 sequences.
```

## 10 **Things to Remember**

- 11 • The special member functions are those compilers may generate on their own:  
12 default constructor, destructor, copy operations, and move operations.
- 13 • Move operations are generated only for classes lacking explicitly-declared  
14 move operations, copy operations, and a destructor.
- 15 • The copy constructor is generated only for classes lacking an explicitly-
 16 declared copy constructor, and it's deleted if a move operation is declared. The
 17 copy assignment operator is generated only for classes lacking an explicitly-
 18 declared copy assignment operator, and it's deleted if a move operation is de-
 19 clared. Generation of the copy operations in classes with an explicitly-declared
 20 destructor is deprecated.
- 21 • Member function templates never suppress generation of special member
 22 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 determine that you should destroy what the pointer points to, there’s no  
13   way to tell how. Should you use `delete`, or is there a different destruction  
14   mechanism (e.g., a dedicated destruction function the pointer should be passed  
15   to)?

16   4. If you manage to find out that `delete` is the way to go, Reason 1 means it may  
17   not be possible to know whether to use the single-object form (“`delete`”) or  
18   the array form (“`delete []`”). If you use the wrong form, results are unde-  
19   fined.

20   5. Assuming you ascertain that the pointer owns what it points to and you dis-  
21   cover how to destroy it, it’s difficult to ensure that you perform the destruction  
22   *exactly once* along every path in your code (including those due to exceptions).  
23   Missing a path leads to resource leaks, and doing the destruction more than  
24   once leads to undefined behavior.

25   6. There’s typically no way to tell if the pointer dangles, i.e., points to memory  
26   that no longer holds the object the pointer is supposed to point to. Dangling  
27   pointers arise when objects are destroyed while pointers still point to them.

1 Raw pointers are powerful tools, to be sure, but decades of experience have  
2 demonstrated that with only the slightest lapse in concentration or discipline,  
3 these tools can turn on their ostensible masters.

4 *Smart pointers* are one way to address these issues. Smart pointers are wrappers  
5 around raw pointers that act much like the raw pointers they wrap, but that avoid  
6 many of their pitfalls. You should therefore prefer smart pointers to raw pointers.  
7 Smart pointers can do virtually everything raw pointers can, but with far fewer  
8 opportunities for error.

9 There are four smart pointers in C++11: `std::auto_ptr`, `std::unique_ptr`,  
10 `std::shared_ptr`, and `std::weak_ptr`. All are designed to help manage the life-  
11 times of dynamically allocated objects, i.e., to avoid resource leaks by ensuring that  
12 such objects are destroyed in the appropriate manner at the appropriate time (in-  
13 cluding in the event of exceptions).

14 `std::auto_ptr` is a deprecated leftover from C++98. It was an attempt to stand-  
15 ardize what later became C++11's `std::unique_ptr`. Doing the job right required  
16 move semantics, but C++98 didn't have them. As a workaround, `std::auto_ptr`  
17 co-opted its copy operations for moves. This led to surprising code (copying a  
18 `std::auto_ptr` sets it to null!) and frustrating usage restrictions (e.g., it's not  
19 possible to store `std::auto_ptr`s in containers).

20 `std::unique_ptr` does everything `std::auto_ptr` does, plus more. It does it as  
21 efficiently, and it does it without warping what it means to copy an object. It's bet-  
22 ter than `std::auto_ptr` in every way. The only legitimate use case for  
23 `std::auto_ptr` is a need to compile code with C++98 compilers. Unless you have  
24 that constraint, you should replace `std::auto_ptr` with `std::unique_ptr` and  
25 never look back.

26 The smart pointer APIs are remarkably varied. About the only functionality com-  
27 mon to all is default construction. Because comprehensive references for these  
28 APIs are widely available in both electronic and print form, I'll focus my discus-  
29 sions on information that's often missing from API overviews, e.g., noteworthy use  
30 cases, runtime cost analyses, etc. Mastering such information can be the difference  
31 between merely using these smart pointers and using them *effectively*.

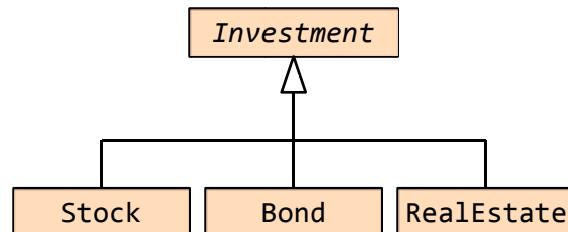
1   **Item 18: Use `std::unique_ptr` for exclusive-ownership re-**  
2   **source management.**

3   When you reach for a smart pointer, `std::unique_ptr` should generally be the  
4   one closest at hand. It's reasonable to assume that, by default, `std::unique_ptr`s  
5   are the same size as raw pointers, and for most operations (including dereferenc-  
6   ing), they execute exactly the same instructions. This means you can use them  
7   even in situations where memory and cycles are tight. If a raw pointer is small  
8   enough and fast enough for you, a `std::unique_ptr` almost certainly is, too.

9   `std::unique_ptr` embodies *exclusive ownership* semantics. A non-null  
10   `std::unique_ptr` always owns what it points to. Moving a `std::unique_ptr`  
11   transfers ownership from the source pointer to the destination pointer. (The  
12   source pointer is set to null.) Copying a `std::unique_ptr` isn't allowed, because  
13   if you could copy a `std::unique_ptr`, you'd end up with two `std::unique_ptr`s  
14   to the same resource, each thinking it owned (and should therefore destroy) that  
15   resource. `std::unique_ptr` is thus a *move-only type*. Upon destruction, a non-null  
16   `std::unique_ptr` destroys its resource. By default, resource destruction is ac-  
17   complished by applying `delete` to the raw pointer inside the `std::unique_ptr`.

18   A common use for `std::unique_ptr` is as a factory function return type for ob-  
19   jects in a hierarchy. Suppose we have a hierarchy for types of investments (e.g.,  
20   stocks, bonds, real estate, etc.) with a base class `Investment`.

```
class Investment { ... };  
  
class Stock:  
    public Investment { ... };  
  
class Bond:  
    public Investment { ... };  
  
class RealEstate:  
    public Investment { ... };
```



21   A factory function for such a hierarchy typically allocates an object on the heap and  
22   returns a pointer to it, with the caller being responsible for deleting the object  
23   when it's no longer needed. That's a perfect match for `std::unique_ptr`, because

1 the caller acquires responsibility for the resource returned by the factory (i.e., exclusive ownership of it), and the `std::unique_ptr` automatically deletes what it  
2 points to when the `std::unique_ptr` is destroyed. A factory function for the `Investment`  
3 hierarchy could be declared like this:

```
5 template<typename... Ts> // return std::unique_ptr
6 std::unique_ptr<Investment> // to an object created
7     makeInvestment(Ts&&... params); // from the given args
```

8 Callers could use the returned `std::unique_ptr` in a single scope, as follows,

```
9 {
10 ...
11     auto pInvestment = // pInvestment is of type
12         makeInvestment( arguments ); // std::unique_ptr<Investment>
13 ...
14 } // destroy *pInvestment
```

15 but they could also use it in ownership-migration scenarios, such as when the  
16 `std::unique_ptr` returned from the factory is moved into a container, the container  
17 element is subsequently moved into a data member of an object, and that  
18 object is later destroyed. When that happens, the object's `std::unique_ptr` data  
19 member would also be destroyed, and its destruction would cause the resource  
20 returned from the factory to be destroyed. If the ownership chain got interrupted  
21 due to an exception or other atypical control flow (e.g., early function return or  
22 `break` from a loop), the `std::unique_ptr` owning the managed resource would  
23 eventually have its destructor called,<sup>†</sup> and the resource it was managing would  
24 thereby be destroyed.

---

<sup>†</sup> There are a few exceptions to this rule. Most 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 14), local objects may not be destroyed, and if `std::abort` or an exit function (i.e., `std::_Exit`, `std::exit`, or `std::quick_exit`) is called, they definitely won't be.

1 By default, that destruction would take place via `delete`, but, during construction,  
2 `std::unique_ptr` objects can be configured to use *custom deleters*: arbitrary  
3 functions (or function objects, including those arising from lambda expressions) to  
4 be invoked when it's time for their resources to be destroyed. If the object created  
5 by `makeInvestment` shouldn't be directly `deleted`, but instead should first have a  
6 log entry written, `makeInvestment` could be implemented as follows. (An expla-  
7 nation follows the code, so don't worry if you see something whose motivation is  
8 less than obvious.)

```
9 auto delInvmt = [](Investment* pInvestment)           // custom
10 {   // deleter
11     makeLogEntry(pInvestment);                         // (a lambda
12     delete pInvestment;                               // expression)
13 };
14 template<typename... Ts>                                // revised
15 std::unique_ptr<Investment, decltype(delInvmt)>        // return type
16 makeInvestment(Ts&&... params)
17 {
18     std::unique_ptr<Investment, decltype(delInvmt)> // ptr to be
19     pInv(nullptr, delInvmt);                          // returned
20
21     if ( /* a Stock object should be created */ )
22     {
23         pInv.reset(new Stock(std::forward<Ts>(args)...));
24     }
25     else if ( /* a Bond object should be created */ )
26     {
27         pInv.reset(new Bond(std::forward<Ts>(args)...));
28     }
29     else if ( /* a RealEstate object should be created */ )
30     {
31         pInv.reset(new RealEstate(std::forward<Ts>(args)...));
32     }
33 }
```

34 In a moment, I'll explain how this works, but first consider how things look if  
35 you're a caller. Assuming you store the result of the `makeInvestment` call in an  
36 `auto` variable, you frolic in blissful ignorance of the fact that the resource you're  
37 using requires special treatment during deletion. In fact, you veritably bathe in  
38 bliss, because the use of `std::unique_ptr` means you need not concern yourself  
39 with when the resource should be destroyed, much less ensure that the destruc-

1      tion happens exactly once along every path through the program.  
2      `std::unique_ptr` takes care of all those things automatically. From a client's  
3      perspective, `makeInvestment`'s interface is sweet.

4      The implementation is pretty nice, too, once you understand the following:

5      • `delInvmt` is the custom deleter for the object returned from `makeInvest-`  
6      `ment`. All custom deletion functions accept a raw pointer to the object to be de-  
7      stroyed, then do what is necessary to destroy that object. In this case, the ac-  
8      tion is to call `makeLogEntry` and then apply `delete`. Using a lambda expres-  
9      sion to create `delInvmt` is convenient, but, as we'll see shortly, it's also more  
10     efficient than writing a conventional function.

11     • When a custom deleter is to be used, its type must be specified as the second  
12     type argument to `std::unique_ptr`. In this case, that's the type of `delInvmt`,  
13     and that's why the return type of `makeInvestment` is  
14     `std::unique_ptr<Investment, decltype(delInvmt)>`. (For information  
15     about `decltype`, see Item 3.)

16     • The basic strategy of `makeInvestment` is to create a null `std::unique_ptr`,  
17     make it point to an object of the appropriate type, and then return it. To asso-  
18     ciate the custom deleter `delInvmt` with `pInv`, we pass that as its second con-  
19     structor argument.

20     • Attempting to assign a raw pointer (e.g., from `new`) to a `std::unique_ptr`  
21     won't compile, because it would constitute an implicit conversion from a raw  
22     to a smart pointer. Such implicit conversions can be problematic, so C++11's  
23     smart pointers prohibit them. That's why `reset` is used to have `pInv` assume  
24     ownership of the object created via `new`.

25     • With each use of `new`, we use `std::forward` to perfect-forward the arguments  
26     passed to `makeInvestment` (see Item 25). This makes all the information pro-  
27     vided by callers available to the constructors of the objects being created.

28     • The custom deleter takes a parameter of type `Investment*`. Regardless of the  
29     actual type of object created inside `makeInvestment` (i.e., `Stock`, `Bond`, or `Re-`

1      `alEstate)`, it will ultimately be deleted inside the lambda expression as an  
2      `Investment*` object. This means we'll be deleting a derived class object via a  
3      base class pointer. For that to work, the base class—`Investment`—must have  
4      a virtual destructor:

```
5      class Investment {  
6      public:  
7          ... // essential  
8          virtual ~Investment(); // design  
9          ... // component!  
10     };
```

11     In C++14, the existence of function return type deduction (see Item 3) means that  
12     `makeInvestment` could be implemented in this simpler and more encapsulated  
13     fashion:

```
14    template<typename... Ts>  
15    auto makeInvestment(Ts&&... params) // C++14  
16    {  
17       auto delInvmt = [](Investment* pInvestment) // this is now  
18            { // inside  
19               makeLogEntry(pInvestment); // make-  
20               delete pInvestment; // Investment  
21            };  
22       std::unique_ptr<Investment, decltype(delInvmt)> // as  
23       pInv(nullptr, delInvmt); // before  
24       if ( ... ) // as before  
25       {  
26           pInv.reset(new Stock(std::forward<Ts>(args)...));  
27       }  
28       else if ( ... ) // as before  
29       {  
30           pInv.reset(new Bond(std::forward<Ts>(args)...));  
31       }  
32       else if ( ... ) // as before  
33       {  
34           pInv.reset(new RealEstate(std::forward<Ts>(args)...));  
35       }  
36       return pInv; // as before  
37 }
```

38     I remarked earlier that, when using the default deleter (i.e., `delete`), you can rea-  
39     sonably assume that `std::unique_ptr` objects are the same size as raw pointers.  
40     When custom deleters enter the picture, this may no longer be the case. Deleters

1 that are function pointers generally cause the size of a `std::unique_ptr` to grow  
2 from one word to two. For deleters that are function objects, the change in size de-  
3 pends on how much state is stored in the function object. Stateless function objects  
4 (e.g., from lambda expressions with no captures) incur no size penalty, and this  
5 means that when a custom deleter can be implemented as either a function or a  
6 captureless lambda expression, the lambda is preferable:

```
7 auto delInvmt1 = [](Investment* pInvestment)           // custom
8 {   // deleter
9     makeLogEntry(pInvestment);                         // as
10    delete pInvestment;                                // stateless
11};   // Lambda

12 template<typename... Ts>                               // return type
13 std::unique_ptr<Investment, decltype(delInvmt1)> // has size of
14 makeInvestment(Ts&... args);                         // Investment*

15

16 void delInvmt2(Investment* pInvestment)           // custom
17 {   // deleter
18     makeLogEntry(pInvestment);                      // as function
19     delete pInvestment;
20}

21 template<typename... Ts>                           // return type has
22 std::unique_ptr<Investment, // size of Investment*
23         void (*)(Investment*)> // plus at least size
24 makeInvestment(Ts&... params);                    // of function pointer!
```

25 Function object deleters with extensive state can yield `std::unique_ptr` objects  
26 of significant size. If you find that a custom deleter makes your  
27 `std::unique_ptr`s unacceptably large, you probably need to change your design.

28 Factory functions are not the only common use case for `std::unique_ptr`s.  
29 They're even more popular as a mechanism for implementing the Pimpl Idiom. The  
30 code for that isn't complicated, but in some cases it's less than straightforward, so  
31 I'll refer you to Item 22, which is dedicated to the topic.

32 `std::unique_ptr` comes in two forms, one for individual objects  
33 (`std::unique_ptr<T>`) and one for arrays (`std::unique_ptr<T[]>`). As a re-  
34 sult, there's never any ambiguity about what kind of entity a `std::unique_ptr`  
35 points to. The `std::unique_ptr` API is designed to match the form you're using.  
36 For example, there's no indexing operator (`operator[]`) for the single-object

1 form, while the array form lacks dereferencing operators (`operator*` and `operator->`).  
2

3 The existence of `std::unique_ptr` for arrays should be of only intellectual inter-  
4 est to you, because `std::array`, `std::vector`, and `std::string` are virtually  
5 always better data structure choices than raw arrays. About the only situation I  
6 can conceive of when a `std::unique_ptr<T[]>` would make sense would be  
7 when you're using a C-like API that returns a raw pointer to a heap array that you  
8 assume ownership of.

9 `std::unique_ptr` is the C++11 way to express exclusive ownership, but one of its  
10 most attractive features is that it easily and efficiently converts to a  
11 `std::shared_ptr`:

12 `std::shared_ptr<Investment> sp = // converts std::unique_ptr`  
13 `makeInvestment( arguments ); // to std::shared_ptr`

14 This is a key part of why `std::unique_ptr` is so well suited as a factory function  
15 return type. Factory functions can't know whether callers will want to use exclu-  
16 sive-ownership semantics for the object they return or whether shared ownership  
17 (i.e., `std::shared_ptr`) would be more appropriate. By returning a  
18 `std::unique_ptr`, factories provide callers with the most efficient smart pointer,  
19 but they don't hinder callers from replacing it with its more flexible sibling. (For  
20 information about `std::shared_ptr`, proceed to Item 19.)

## 21 Things to Remember

- 22   ♦ `std::unique_ptr` is a small, fast, move-only smart pointer for managing re-  
23         sources with exclusive-ownership semantics.
- 24   ♦ By default, resource destruction takes place via `delete`, but custom deleters  
25         can be specified. Stateful deleters and function pointers as deleters increase  
26         the size of `std::unique_ptr` objects.
- 27   ♦ Converting a `std::unique_ptr` to a `std::shared_ptr` is easy.

1   **Item 19: Use `std::shared_ptr` for shared-ownership re-**  
2   **source management.**

3   Programmers using languages with garbage collection point and laugh at what C++  
4   programmers go through to prevent resource leaks. “How primitive!”, they jeer.  
5   “Didn’t you get the memo from Lisp in the 1960s? Machines should manage re-  
6   source lifetimes, not humans.” C++ developers roll their eyes. “You mean the  
7   memo where the only resource is memory and the timing of resource reclamation  
8   is nondeterministic? We prefer the generality and predictability of destructors,  
9   thank you.” But our bravado is part bluster. Garbage collection really is convenient,  
10   and manual lifetime management really can seem akin to constructing a mnemonic  
11   memory circuit using stone knives and bear skins. Why can’t we have the best of  
12   both worlds: a system that works automatically (like garbage collection), yet ap-  
13   plies to all resources and has predictable timing (like destructors)?

14   `std::shared_ptr` is the C++11 way of binding these worlds together. An object  
15   accessed via `std::shared_ptr`s has its lifetime managed by those pointers  
16   through *shared ownership*. No specific `std::shared_ptr` owns the object. Instead,  
17   all `std::shared_ptr`s pointing to it collaborate to ensure its destruction at the  
18   point where it’s no longer needed. When the last `std::shared_ptr` pointing to an  
19   object stops pointing there (e.g., because the `std::shared_ptr` is destroyed or  
20   made to point to a different object), that `std::shared_ptr` destroys the object it  
21   points to. As with garbage collection, clients need not concern themselves with  
22   managing the lifetime of pointed-to objects, but as with destructors, the timing of  
23   the objects’ destruction is deterministic.

24   A `std::shared_ptr` can tell whether it’s the last one pointing to a resource by  
25   consulting the resource’s *reference count*, a value associated with the resource that  
26   keeps track of how many `std::shared_ptr`s point to it. `std::shared_ptr` con-  
27   structors increment this count (usually—see below), `std::shared_ptr` destruc-  
28   tors decrement it, and copy assignment operators do both. (If `sp1` and `sp2` are  
29   `std::shared_ptr`s to different objects, the assignment “`sp1 = sp2`” modifies `sp1`  
30   such that it points to the object pointed to by `sp2`. The net effect of the assignment  
31   is that the reference count for the object originally pointed to by `sp1` is decre-  
32   mented, while that for the object pointed to by `sp2` is incremented.) If a

1    `std::shared_ptr` sees a reference count of zero after performing a decrement,  
2    no more `std::shared_ptr`s point to the resource, so the `std::shared_ptr` de-  
3    stroys it.

4    The existence of the reference count has performance implications:

5    • **`std::shared_ptr`s are twice the size of a raw pointer**, because they inter-  
6    nally contain a raw pointer to the resource as well as a raw pointer to the re-  
7    source's reference count.<sup>†</sup>

8    • **Memory for the reference count must be dynamically allocated**. Conceptu-  
9    ally, the reference count is associated with the object being pointed to, but  
10   pointed-to objects know nothing about this. They thus have no place to store a  
11   reference count. (A pleasant implication is that any object—even those of  
12   built-in types—may be managed by `std::shared_ptr`s.) Item 21 explains  
13   that the cost of the dynamic allocation is avoided when the `std::shared_ptr`  
14   is created by `std::make_shared`, but there are situations where  
15   `std::make_shared` can't be used. Either way, the reference count is stored as  
16   dynamically allocated data.

17   • **Increments and decrements of the reference count must be atomic**, be-  
18   cause there can be simultaneous readers and writers in different threads. For  
19   example, a `std::shared_ptr` pointing to a resource in one thread could be  
20   executing its destructor (hence decrementing the reference count for the re-  
21   source it points to), while, in a different thread, a `std::shared_ptr` to the  
22   same object could be copied (and therefore incrementing the same reference  
23   count). Atomic operations are typically slower than non-atomic operations, so  
24   even though reference counts are usually only a word in size, you should as-  
25   sume that reading and writing them is comparatively costly.

---

<sup>†</sup> This implementation is not required by the Standard, but every Standard Library imple-  
mentation I'm familiar with employs it.

1 Did I pique your curiosity when I wrote that `std::shared_ptr` constructors only  
2 “usually” increment the reference count for the object they point to? Creating a  
3 `std::shared_ptr` pointing to an object always yields one more  
4 `std::shared_ptr` pointing to that object, so why mustn’t we *always* increment  
5 the reference count?

6 Move construction, that’s why. Move-constructing a `std::shared_ptr` from an-  
7 other `std::shared_ptr` sets the source `std::shared_ptr` to null, and that  
8 means that the old `std::shared_ptr` stops pointing to the resource at the mo-  
9 ment the new `std::shared_ptr` starts. As a result, no reference count manipula-  
10 tion is required. Moving `std::shared_ptr`s is therefore faster than copying  
11 them: copying requires incrementing the reference count, but moving doesn’t. This  
12 is as true for assignment as for construction, so move construction is faster than  
13 copy construction, and move assignment is faster than copy assignment.

14 Like `std::unique_ptr` (see Item 18), `std::shared_ptr` uses `delete` as its de-  
15 fault resource-destruction mechanism, but it also supports custom deleters. The  
16 design of this support differs from that for `std::unique_ptr`, however. For  
17 `std::unique_ptr`, the type of the deleter is part of the type of the smart pointer.  
18 For `std::shared_ptr`, it’s not:

```
19 auto loggingDel = [](Widget *pw)           // custom deleter
20 {   // (as in Item 18)
21     makeLogEntry(pw);
22     delete pw;
23 };
24 std::unique_ptr<           // deleter type is
25     Widget, decltype(loggingDel)>        // part of ptr type
26 > upw(new Widget, loggingDel);
27 std::shared_ptr<Widget>           // deleter type is not
28 spw(new Widget, loggingDel);         // part of ptr type
```

29 The `std::shared_ptr` design is more flexible. Consider two  
30 `std::shared_ptr<Widget>`s, each with a custom deleter of a different type (e.g.,  
31 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:

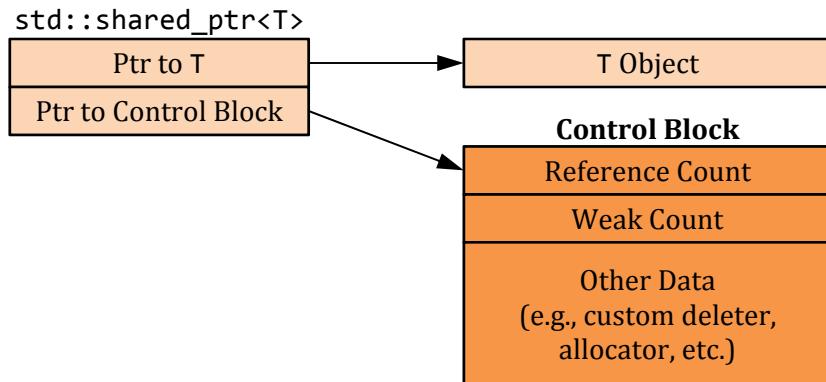
```
8 std::vector<std::shared_ptr<Widget>> vpw{ pw1, pw2 };
```

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 the 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. The control block may also contain additional data, including, as  
32 Item 21 explains, a secondary reference count known as the weak count, but we'll

1 ignore such data in this Item. We can envision the memory associated with a  
2 `std::shared_ptr<T>` object as looking like this:



3

4 An object's control block is set up by the function creating the first  
5 `std::shared_ptr` to the object. At least that's what's supposed to happen. In  
6 general, it's impossible for a function creating a `std::shared_ptr` to an object to  
7 know whether some other `std::shared_ptr` already points to that object, so the  
8 following rules for control block creation are used:

- **`std::make_shared` (see Item 21) always creates a control block.** It manufactures a new object to point to, so there is certainly no control block for that object at the time `std::make_shared` is called.
- **A control block is created when a `std::shared_ptr` is constructed from a unique-ownership pointer (i.e., a `std::unique_ptr` or `std::auto_ptr`).** Unique-ownership pointers don't use control blocks, so there should be no control block for the pointed-to object. (As part of its construction, the `std::shared_ptr` assumes ownership of the pointed-to object, so the unique-ownership pointer is set to null.)
- **When a `std::shared_ptr` constructor is called with a raw pointer, it creates a control block.** If you wanted to create a `std::shared_ptr` from an object that already had a control block, you'd presumably pass a `std::shared_ptr` or a `std::weak_ptr` (see Item 20) as a constructor argument, not a raw pointer. `std::shared_ptr` constructors taking `std::shared_ptrs` or `std::weak_ptrs` as constructor arguments don't cre-

1       ate new control blocks, because they can rely on the smart pointers passed to  
2       them to point to any necessary control blocks.

3       A consequence of these rules is that constructing more than one  
4       `std::shared_ptr` from a single raw pointer gives you a complimentary ride on  
5       the particle accelerator of undefined behavior, because the pointed-to object will  
6       have multiple control blocks. Multiple control blocks means multiple reference  
7       counts, and multiple reference counts means the object will be destroyed multiple  
8       times (once for each reference count). That means that code like this is bad, bad,  
9       bad:

```
10 auto pw = new Widget; // pw is raw ptr
11 ...
12 std::shared_ptr<Widget> spw1(pw, loggingDel); // create control
13 // block for *pw
14 ...
15 std::shared_ptr<Widget> spw2(pw, loggingDel); // create 2nd
16 // control block
17 // for *pw!
```

18       The creation of the raw pointer `pw` to a dynamically allocated object is bad, be-  
19       cause it runs contrary to the advice behind this entire chapter: to prefer smart  
20       pointers to raw pointers. (If you've forgotten the motivation for that advice, turn to  
21       page 127 to refresh your memory.) But set that aside. The line creating `pw` is a sty-  
22       listic abomination, but at least it doesn't cause undefined program behavior.

23       Now, the constructor for `spw1` is called with a raw pointer, so it creates a control  
24       block (and thereby a reference count) for what's pointed to. In this case, that's `*pw`  
25       (i.e., the object pointed to by `pw`). In and of itself, that's okay, but the constructor  
26       for `spw2` is called with the same raw pointer, so it also creates a control block  
27       (hence a reference count) for `*pw`. `*pw` thus has two reference counts, each of  
28       which will eventually become zero, and that will ultimately lead to an attempt to  
29       destroy `*pw` twice. The second destruction is responsible for the undefined behav-  
30       ior.

31       There are at least two lessons regarding `std::shared_ptr` use here. First, try to  
32       avoid passing raw pointers to a `std::shared_ptr` constructor. The usual alterna-

1 tive is to use `std::make_shared` (see Item 21), but in the example above, we're  
2 using custom deleters, and that's not possible with `std::make_shared`. Second, if  
3 you must pass a raw pointer to a `std::shared_ptr` constructor, pass the result of  
4 `new` directly instead of going through a raw pointer variable. If the first part of the  
5 code above were rewritten like this,

```
6 std::shared_ptr<Widget> spw1(new Widget,           // direct use of new
7                               loggingDel);
```

8 it'd be a lot less tempting to create a second `std::shared_ptr` from the same raw  
9 pointer. Instead, the author of the code creating `spw2` would naturally use `spw1` as  
10 an initialization argument (i.e., would call the `std::shared_ptr` copy construc-  
11 tor), and that would pose no problem whatsoever:

```
12 std::shared_ptr<Widget> spw2(spw1);           // spw2 uses same
13                                     // control block as spw1
```

14 An especially surprising way that using raw pointer variables as  
15 `std::shared_ptr` constructor arguments can lead to multiple control blocks in-  
16 volves the `this` pointer. Suppose our program uses `std::shared_ptr`s to man-  
17 age `Widget` objects, and we have a data structure that keeps track of `Widgets` that  
18 have been processed:

```
19 std::vector<std::shared_ptr<Widget>> processedWidgets;
```

20 Further suppose that `Widget` has a member function that does the processing:

```
21 class Widget {
22 public:
23 ...
24     void process();
25 ...
26 };
```

27 Here's a reasonable-looking approach for `Widget::process`:

```
28 void Widget::process()
29 {
30     ...                                // process the Widget
31     processedWidgets.emplace_back(this); // add it to list of
32 }                                     // processed Widgets;
   // this is wrong!
```

1 The comment about this being wrong says it all—or at least most of it. (The part  
2 that's wrong is the passing of `this`, not the use of `emplace_back`. If you're not fa-  
3 miliar with `emplace_back`, see Item 42.) This code will compile, but it's passing a  
4 raw pointer (`this`) to a container of `std::shared_ptr`s. The `std::shared_ptr`  
5 thus constructed will create a new control block for the pointed-to `Widget`  
6 (`*this`). That doesn't sound harmful until you realize that if there are  
7 `std::shared_ptr`s outside the member function that already point to that `Widget`,  
8 it's game, set, and match for undefined behavior.

9 The `std::shared_ptr` API includes a facility for just this kind of situation. It has  
10 probably the oddest of all names in the Standard C++ Library:  
11 `std::enable_shared_from_this`. That's a template for a base class you inherit  
12 from if you want a class managed by `std::shared_ptr`s to be able to safely cre-  
13 ate a `std::shared_ptr` from a `this` pointer. In our example, `Widget` would in-  
14 herit from `std::enable_shared_from_this` as follows:

```
15 class Widget: public std::enable_shared_from_this<Widget> {  
16     public:  
17         ...  
18         void process();  
19         ...  
20     };
```

21 As I said, `std::enable_shared_from_this` is a base class template. Its type pa-  
22 rameter is always the name of the class being derived, so `Widget` inherits from  
23 `std::enable_shared_from_this<Widget>`. If the idea of a derived class inher-  
24 iting from a base class templatized on the derived class makes your head hurt, try  
25 not to think about it. The code is completely legal, and the design pattern behind it  
26 is so well established, it has a standard name, albeit one that's almost as odd as  
27 `std::enable_shared_from_this`. The name is *The Curiously Recurring Tem-  
28 plate Pattern (CRTP)*. If you'd like to learn more about it, unleash your search en-  
29 gine, because here we need to get back to `std::enable_shared_from_this`.

30 `std::enable_shared_from_this` defines a member function that creates a  
31 `std::shared_ptr` to the current object, but it does it without duplicating control  
32 blocks. The member function is `shared_from_this`, and you use it in member

1 functions whenever you want a `std::shared_ptr` that points to the same object  
2 as the `this` pointer. Here's a safe implementation of `Widget::process`:

```
3 void Widget::process()
4 {
5     // as before, process the Widget
6     ...
7     // add std::shared_ptr to current object to processedWidgets
8     processedWidgets.emplace_back(shared_from_this());
9 }
```

10 Internally, `shared_from_this` looks up the control block for the current object,  
11 and it creates a new `std::shared_ptr` that refers to that control block. The de-  
12 sign relies on the current object having an associated control block. For that to be  
13 the case, there must be an existing `std::shared_ptr` (e.g., one outside the mem-  
14 ber function calling `shared_from_this`) that points to the current object. If no  
15 such `std::shared_ptr` exists (i.e., if the current object has no associated control  
16 block), behavior is officially undefined, although `shared_from_this` typically  
17 throws an exception.

18 To prevent clients from calling member functions that invoke `shared_from_this`  
19 before a `std::shared_ptr` points to the object, classes inheriting from  
20 `std::enable_shared_from_this` often declare their constructors `private` and  
21 have clients create objects by calling factory functions that return  
22 `std::shared_ptr`s. `Widget`, for example, could look like this:

```
23 class Widget: public std::enable_shared_from_this<Widget> {
24 public:
25     // factory function that perfect-forwards args
26     // to a private ctor
27     template<typename... Ts>
28     static std::shared_ptr<Widget> create(Ts&&... params);
29     ...
30     void process();           // as before
31     ...
32 private:
33     ...                      // ctors
34 };
```

1 By now, you may only dimly recall that our discussion of control blocks was moti-  
2 vated by a desire to understand the costs associated with `std::shared_ptr`s.  
3 Now that we understand how to avoid creating too many control blocks, let's re-  
4 turn to the original topic.

5 A control block is typically only a few words in size, although custom deleters and  
6 allocators may make it larger. The usual control block implementation is more so-  
7 phisticated than you might expect. It makes use of inheritance, and there's even a  
8 virtual function. (It's used to ensure that the pointed-to object is properly de-  
9 stroyed.) That means that using `std::shared_ptr`s also incurs the cost of the  
10 machinery for the virtual function used by the control block.

11 Having read about dynamically allocated control blocks, arbitrarily large deleters  
12 and allocators, virtual function machinery, and atomic reference count manipula-  
13 tions, your enthusiasm for `std::shared_ptr`s may have waned somewhat. That's  
14 fine. They're not the best solution to every resource management problem. But for  
15 the functionality they provide, `std::shared_ptr`s exact a very reasonable cost.  
16 Under typical conditions, where the default deleter and default allocator are used  
17 and where the `std::shared_ptr` is created by `std::make_shared`, the control  
18 block is only about three words in size, and its allocation is essentially free. (It's  
19 incorporated into the memory allocation for the object being pointed to. For de-  
20 tails, see Item 21.) Dereferencing a `std::shared_ptr` is no more expensive than  
21 dereferencing a raw pointer. Performing an operation requiring a reference count  
22 manipulation (e.g., copy construction or copy assignment, destruction) entails one  
23 or two atomic operations, but these operations typically map to individual ma-  
24 chine instructions, so although they may be expensive compared to non-atomic  
25 instructions, they're still just single instructions. The virtual function machinery in  
26 the control block is generally used only once per object managed by  
27 `std::shared_ptr`s: when the object is destroyed.

28 In exchange for these rather modest costs, you get automatic lifetime management  
29 of dynamically allocated resources. Most of the time, using `std::shared_ptr` is  
30 vastly preferable to trying to manage the lifetime of an object with shared owner-  
31 ship by hand. If you find yourself doubting whether you can afford use of  
32 `std::shared_ptr`, reconsider whether you really need shared ownership. If ex-

1   clusive ownership will do or even *may* do, `std::unique_ptr` is a better choice. Its  
2   performance profile is close to that for raw pointers, and “upgrading” from  
3   `std::unique_ptr` to `std::shared_ptr` is easy, because a `std::shared_ptr`  
4   can be created from a `std::unique_ptr`.

5   The reverse is not true. Once you’ve turned lifetime management of a resource  
6   over to a `std::shared_ptr`, there’s no changing your mind. Even if the reference  
7   count is one, you can’t reclaim ownership of the resource in order to, say, have a  
8   `std::unique_ptr` manage it. The ownership contract between a resource and the  
9   `std::shared_ptr`s that point to it is of the ‘til-death-do-us-part variety. No di-  
10   vorce, no annulment, no dispensations.

11   Something else `std::shared_ptr`s can’t do is work with arrays. In yet another  
12   difference from `std::unique_ptr`, `std::shared_ptr` has an API that’s designed  
13   only for pointers to single objects. There’s no `std::shared_ptr<T[]>`. From time  
14   to time, “clever” programmers stumble on the idea of using a  
15   `std::shared_ptr<T>` to point to an array, specifying a custom deleter to perform  
16   an array delete (i.e., `delete []`). This can be made to compile, but it’s a horrible  
17   idea. For one thing, `std::shared_ptr` offers no `operator[]`, so indexing into the  
18   array requires awkward expressions based on pointer arithmetic. For another,  
19   `std::shared_ptr` supports derived-to-base pointer conversions that make sense  
20   for single objects, but that open holes in the type system when applied to arrays.  
21   (For this reason, the `std::unique_ptr<T[]>` API prohibits such conversions.)  
22   Most importantly, given the variety of C++11 alternatives to built-in arrays (e.g.,  
23   `std::array`, `std::vector`, `std::string`), declaring a smart pointer to a dumb  
24   array is almost always a sign of bad design.

25   **Things to Remember**

- 26   ♦ `std::shared_ptr`s offer convenience approaching that of garbage collection  
27   for the shared lifetime management of arbitrary resources.
- 28   ♦ Compared to `std::unique_ptr`, `std::shared_ptr` objects are twice as big,  
29   incur overhead for control blocks, and require atomic reference count manipu-  
30   lations.

- 1   ♦ Default resource destruction is via `delete`, but custom deleters are supported.
- 2   The type of the deleter has no effect on the type of the `std::shared_ptr`.
- 3   ♦ Avoid creating `std::shared_ptr`s from variables of raw pointer type.

#### 4   **Item 20: Use `std::weak_ptr` for `std::shared_ptr`-like 5    pointers that can dangle.**

6   Paradoxically, it can be convenient to have a smart pointer that acts like a  
7   `std::shared_ptr` (see Item 19), but that doesn't participate in the shared own-  
8   ership of the pointed-to resource. In other words, a pointer like  
9   `std::shared_ptr` that doesn't affect an object's reference count. This kind of  
10   smart pointer has to contend with a problem unknown to `std::shared_ptr`s: the  
11   possibility that what it points to has been destroyed. A truly smart pointer would  
12   deal with this problem by tracking when it *dangles*, i.e., when the object it is sup-  
13   posed to point to no longer exists. That's precisely the kind of smart pointer  
14   `std::weak_ptr` is.

15   You may be wondering how a `std::weak_ptr` could be useful. You'll probably  
16   wonder even more when you examine the `std::weak_ptr` API. It looks anything  
17   but smart. `std::weak_ptr`s can't be dereferenced, nor can they be tested for null-  
18   ness. That's because `std::weak_ptr` isn't a standalone smart pointer. It's an  
19   augmentation of `std::shared_ptr`.

20   The relationship begins at birth. `std::weak_ptr`s are typically created from  
21   `std::shared_ptr`s. They point to the same place as the `std::shared_ptr`s ini-  
22   tializing them, but they don't affect the reference count of the object they point to:

```
23 auto spw = std::make_shared<Widget>(); // after spw is constructed,  
24 // the pointed-to Widget's  
25 // ref count (RC) is 1. (See  
26 // Item 21 for info on  
27 // std::make_shared.)  
28 ...  
29 std::weak_ptr<Widget> wpw(spw); // wpw points to same Widget  
30 // as spw. RC remains 1  
31 ...
```

```
1 spw = nullptr; // RC goes to 0, and the  
2 // Widget is destroyed.  
3 // wpw now dangles
```

4 `std::weak_ptr`s that dangle are said to have *expired*. You can test for this directly,

```
6 if (wpw.expired()) ... // if wpw doesn't point  
7 // to an object...
```

8 but often what you desire is a check to see if a `std::weak_ptr` has expired and, if 9 it hasn't (i.e., if it's not dangling), to access the object it points to. This is easier 10 desired than done. Because `std::weak_ptr`s lack dereferencing operations, there's 11 no way to write the code. Even if there were, separating the check and the dereference 12 would introduce a race condition: between the call to `expired` and the dereferencing 13 action, another thread might reassign or destroy the last 14 `std::shared_ptr` pointing to the object, thus causing that object to be destroyed. 15 In that case, your dereference would yield undefined behavior.

16 What you need is an atomic operation that checks to see if the `std::weak_ptr` has 17 expired and, if not, gives you access to the object it points to. This is done by creating 18 a `std::shared_ptr` from the `std::weak_ptr`. The operation comes in two 19 forms, depending on what you'd like to have happen if the `std::weak_ptr` has 20 expired when you try to create a `std::shared_ptr` from it. One form is 21 `std::weak_ptr::lock`, which returns a `std::shared_ptr`. The 22 `std::shared_ptr` is null if the `std::weak_ptr` has expired:

```
23 std::shared_ptr<Widget> spw1 = wpw.lock(); // if wpw's expired,  
24 // spw1 is null  
  
25 auto spw2 = wpw.lock(); // same as above,  
26 // but uses auto
```

27 The other form is the `std::shared_ptr` constructor taking a `std::weak_ptr` as 28 an argument. In this case, if the `std::weak_ptr` has expired, an exception is 29 thrown:

```
30 std::shared_ptr<Widget> spw3(wpw); // if wpw's expired,  
31 // throw std::bad_weak_ptr
```

1 But you're probably still wondering about how `std::weak_ptr`s can be useful.  
2 Consider a factory function that produces smart pointers to read-only objects  
3 based on a unique ID. In accord with Item 18's advice regarding factory function  
4 return types, it returns a `std::unique_ptr`:

5 `std::unique_ptr<const Widget> loadWidget(WidgetID id);`  
6 If `loadWidget` is an expensive call (e.g., because it performs file or database I/O)  
7 and it's common for IDs to be used repeatedly, a reasonable optimization would be  
8 to write a function that does what `loadWidget` does, but also caches its results.  
9 Clogging the cache with every `Widget` that has ever been requested can lead to  
10 performance problems of its own, however, so another reasonable optimization  
11 would be to destroy cached `Widgets` when they're no longer in use.

12 For this caching factory function, a `std::unique_ptr` return type is not a good fit.  
13 Callers should certainly receive smart pointers to cached objects, and callers  
14 should certainly determine the lifetime of those objects, but the cache needs a  
15 pointer to the objects, too. The cache's pointers need to be able to detect when  
16 they dangle, because when factory clients are finished using an object returned by  
17 the factory, that object will be destroyed, and the corresponding cache entry will  
18 dangle. The cached pointers should therefore be `std::weak_ptr`s—pointers that  
19 can detect when they dangle. That means that the factory's return type should be a  
20 `std::shared_ptr`, because `std::weak_ptr`s can detect when they dangle only  
21 when an object's lifetime is managed by `std::shared_ptr`s.

22 Here's a quick-and-dirty implementation of a caching version of `loadWidget`:

```
23 std::shared_ptr<const Widget> fastLoadWidget(WidgetID id)
24 {
25     static std::unordered_map<WidgetID,
26                             std::weak_ptr<const Widget>> cache;
27     auto objPtr = cache[id].lock();    // objPtr is std::shared_ptr
28   // to cached object (or null
29   // if object's not in cache)
30     if (!objPtr) {                  // if not in cache,
31         objPtr = loadWidget(id);   // load it
32         cache[id] = objPtr;       // cache it
33     }
```

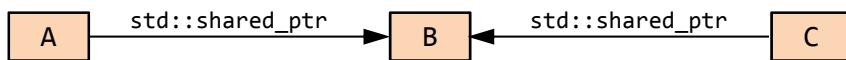
```

1     return objPtr;
2 }

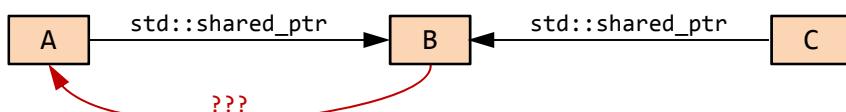
3 This implementation employs one of C++11's hash table containers
4 (std::unordered_map), though it doesn't show the WidgetID hashing and equal-
5 ity-comparison functions that would also have to be present.

6 This implementation of fastLoadWidget ignores the fact that the cache may ac-
7 cumulate expired std::weak_ptrs corresponding to Widgets that are no longer
8 in use (and have therefore been destroyed). The implementation can be refined,
9 but rather than spend time on an issue that lends no additional insight into
10 std::weak_ptr, let's consider a second use case: the Observer design pattern.
11 The primary components of this pattern are subjects (objects whose state may
12 change) and observers (objects to be notified when state changes occur). In most
13 implementations, each subject contains a data member holding pointers to its ob-
14 servers. That makes it easy for subjects to issue state change notifications. Subjects
15 have no interest in controlling the lifetime of their observers (i.e., when they're
16 destroyed), but they have a great interest in making sure that if an observer gets
17 destroyed, subjects don't try to subsequently access it. A reasonable design is for
18 each subject to hold a container of std::weak_ptrs to its observers, thus making
19 it possible for the subject to determine whether a pointer dangles before using it.

20 As a final example of std::weak_ptr's utility, consider a data structure with ob-
21 jects A, B, and C in it, where A and C share ownership of B and therefore hold
22 std::shared_ptrs to it:
```



23  
24 Suppose it'd be useful to also have a pointer from B back to A. What kind of pointer  
25 should this be?



26  
27 There are three choices:

1     • **A raw pointer.** With this approach, if A is destroyed, but C continues to point  
2       to B, B will contain a pointer to A that will dangle. B won't be able to detect that,  
3       so B may inadvertently dereference the dangling pointer. That would yield un-  
4       defined behavior.

5     • **A `std::shared_ptr`.** In this design, A and B contain `std::shared_ptr`s to  
6       each other. The resulting `std::shared_ptr` cycle (A points to B and B points  
7       to A) will prevent both A and B from being destroyed. Even if A and B are un-  
8       reachable from other program data structures (e.g., because C no longer points  
9       to B), each will have a reference count of one. If that happens, A and B will have  
10      been leaked, for all practical purposes: it will be impossible for the program to  
11      access them, yet their resources will never be reclaimed.

12    • **A `std::weak_ptr`.** This avoids both problems above. If A is destroyed, B's  
13      pointer back to it will dangle, but B will be able to detect that. Furthermore,  
14      though A and B will point to one another, B's pointer won't affect A's reference  
15      count, hence can't keep A from being destroyed when `std::shared_ptr`s no  
16      longer point to it.

17    Using `std::weak_ptr` is clearly the best of these choices. However, it's worth not-  
18      ing that the need to employ `std::weak_ptr`s to break prospective cycles of  
19      `std::shared_ptr`s is not terribly common. In strictly hierachal data structures  
20      such as trees, child nodes are typically owned only by their parents. When a parent  
21      node is destroyed, its child nodes should be destroyed, too. Links from parents to  
22      children are thus generally best represented by `std::unique_ptr`s. Back-links  
23      from children to parents can be safely implemented as raw pointers, because a  
24      child node should never have a lifetime longer than its parent. There's thus no risk  
25      of a child node dereferencing a dangling parent pointer.

26    Of course, not all pointer-based data structures are strictly hierarchical, and when  
27      that's the case, as well as in situations such as caching and the implementation of  
28      lists of observers, it's nice to know that `std::weak_ptr` stands at the ready.

29    From an efficiency perspective, the `std::weak_ptr` story is essentially the same  
30      as that for `std::shared_ptr`. `std::weak_ptr` objects are the same size as

1    `std::shared_ptr` objects, they make use of the same control blocks as  
2    `std::shared_ptrs` (see Item 19), and operations such as construction, destruc-  
3    tion, and assignment involve atomic reference count manipulations. That probably  
4    surprises you, because I wrote at the beginning of this Item that `std::weak_ptrs`  
5    don't participate in reference counting. Except that's not quite what I wrote. What  
6    I wrote was that `std::weak_ptrs` don't participate in the *shared ownership* of ob-  
7    jects and hence don't affect the *pointed-to object's reference count*. There's actually  
8    a second reference count in the control block, and it's this second reference count  
9    that `std::weak_ptrs` manipulate. For details, continue on to Item 21.

10   **Things to Remember**

- 11   • Use `std::weak_ptr` for `std::shared_ptr`-like pointers that can dangle.  
12   • Potential use cases for `std::weak_ptr` include caching, observer lists, and the  
13   prevention of `std::shared_ptr` cycles.

14   **Item 21: Prefer `std::make_unique` and `std::make_shared`  
15   to direct use of `new`.**

16   Let's begin by leveling the playing field for `std::make_unique` and  
17   `std::make_shared`. `std::make_shared` is part of C++11, but, sadly,  
18   `std::make_unique` isn't. It joined the Standard Library as of C++14. If you're us-  
19   ing C++11, never fear, because a basic version of `std::make_unique` is easy to  
20   write yourself. Here, look:

```
21 template<typename T, typename... Ts>
22 std::unique_ptr<T> make_unique(Ts&&... params)
23 {
24     return std::unique_ptr<T>(new T(std::forward<Ts>(params)...));
25 }
```

26   As you can see, `make_unique` just perfect-forwards its parameters to the construc-  
27   tor of the object being created, constructs a `std::unique_ptr` from the raw  
28   pointer `new` produces, and returns the `std::unique_ptr` so created. This form of  
29   the function doesn't support arrays or custom deleters (see Item 18), but it

1 demonstrates that with only a little effort, you can create `make_unique` if you  
2 need to.<sup>†</sup> Just remember not to put your version in namespace `std`, because you  
3 won't want it to clash with a vendor-provided version when you upgrade to a  
4 C++14 Standard Library implementation.

5 `std::make_unique` and `std::make_shared` are two of the three *make functions*:  
6 functions that take an arbitrary set of arguments, perfect-forward them to the con-  
7 structor for a dynamically-allocated object, and return a smart pointer to that ob-  
8 ject. The third `make` function is `std::allocate_shared`. It acts just like  
9 `std::make_shared`, except its first argument is an allocator object to be used for  
10 the dynamic memory allocation.

11 Even the most trivial comparison of smart pointer creation using and not using a  
12 `make` function reveals the first reason why using such functions is preferable. Con-  
13 sider:

```
14 auto upw1(std::make_unique<Widget>());           // with make func
15 std::unique_ptr<Widget> upw2(new Widget);         // without make func
16
17 auto spw1(std::make_shared<Widget>());           // with make func
18 std::shared_ptr<Widget> spw2(new Widget);         // without make func
```

19 I've highlighted the essential difference: the versions using `new` repeat the type  
20 being created, but the `make` functions don't. Repeating types runs afoul of a key  
21 tenet of software engineering: code duplication should be avoided. Duplication in  
22 source code increases compilation times, can lead to bloated object code, and gen-  
23 erally renders a code base more difficult to work with. It often evolves into incon-  
24 sistent code, and inconsistency in a code base often leads to bugs. Besides, typing  
25 something twice takes more effort than typing it once, and who's not a fan of re-  
26 ducing their typing burden?

---

<sup>†</sup> 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 N3656 by Stephan T. Lavavej, dated 2013-04-18.

1 The second reason to prefer `make` functions has to do with exception safety. Suppose  
2 we have a function to process a `Widget` in accord with some priority:

3 `void processWidget(std::shared_ptr<Widget> spw, int priority);`  
4 Passing the `std::shared_ptr` by value may look suspicious, but Item 41 explains  
5 that if `processWidget` always makes a copy of the `std::shared_ptr` (e.g., by  
6 storing it in a data structure tracking `Widgets` that have been processed), this can  
7 be a reasonable design choice.

8 Now suppose we have a function to compute the relevant priority,

9 `int computePriority();`

10 and we use that in a call to `processWidget` that uses `new` instead of  
11 `std::make_shared`:

12 `processWidget(std::shared_ptr<Widget>(new Widget), // potential`  
13 `computePriority()); // resource`  
14  `// Leak!`

15 As the comment indicates, this code could leak the `Widget` conjured up by `new`. But  
16 how? Both the calling code and the called function are using `std::shared_ptrs`,  
17 and `std::shared_ptrs` are designed to prevent resource leaks. They automatically  
18 destroy what they point to when the last `std::shared_ptr` pointing there  
19 goes away. If everybody is using `std::shared_ptrs` everywhere, how can this  
20 code leak?

21 The answer has to do with compilers' translation of source code into object code.  
22 At run time, the arguments for a function must be evaluated before the function  
23 can be invoked, so in the call to `processWidget`, the following things must occur  
24 before `processWidget` can begin execution:

- 25 • The expression “`new Widget`” must be evaluated, i.e., a `Widget` must be created  
26 on the heap.
- 27 • The constructor for the `std::shared_ptr<Widget>` responsible for managing  
28 the pointer produced by `new` must be executed.
- 29 • `computePriority` must run.

1 Compilers are not required to generate code that executes them in this order. “`new`  
2 `Widget`” must be executed before the `std::shared_ptr` constructor may be  
3 called, because the result of that `new` is used as an argument to that constructor,  
4 but `computePriority` may be executed before those calls, after them, or, crucial-  
5 ly, *between* them. That is, compilers may emit code to execute the operations in  
6 this order:

7 1. Perform “`new Widget`”.

8 2. Execute `computePriority`.

9 3. Run `std::shared_ptr` constructor.

10 If such code is generated and, at run time, `computePriority` produces an excep-  
11 tion, the dynamically allocated `Widget` from Step 1 will be leaked, because it will  
12 never be stored in the `std::shared_ptr` that’s supposed to start managing it in  
13 Step 3.

14 Using `std::make_shared` avoids this problem. Calling code would look like this:

```
15 processWidget(std::make_shared<Widget>(), // no potential
16                 computePriority()); // resource leak
```

17 At run time, either `std::make_shared` or `computePriority` will be called first. If  
18 it’s `std::make_shared`, the raw pointer to the dynamically allocated `Widget` is  
19 safely stored in the returned `std::shared_ptr` before `computePriority` is  
20 called. If `computePriority` then yields an exception, the `std::shared_ptr` de-  
21 structor will see to it that the `Widget` it owns is destroyed. And if `computePrior-`  
22 `ity` is called first and yields an exception, `std::make_shared` will not be invoked,  
23 and there will hence be no dynamically allocated `Widget` to worry about.

24 If we replace `std::shared_ptr` and `std::make_shared` with  
25 `std::unique_ptr` and `std::make_unique`, exactly the same reasoning applies.  
26 Using `std::make_unique` instead of `new` is thus just as important in writing ex-  
27 ception-safe code as using `std::make_shared`.

28 A special feature of `std::make_shared` (compared to direct use of `new`) is im-  
29 proved efficiency. Using `std::make_shared` allows compilers to generate smaller,

1 faster code that employs leaner data structures. Consider the following direct use  
2 of `new`:

3 `std::shared_ptr<Widget> spw(new Widget);`

4 It's obvious that this code entails a memory allocation, but it actually performs  
5 two. Item 19 explains that every `std::shared_ptr` points to a control block con-  
6 taining, among other things, the reference count for the pointed-to object. Memory  
7 for this control block is allocated in the `std::shared_ptr` constructor. Direct use  
8 of `new`, then, requires one memory allocation for the `Widget` and a second alloca-  
9 tion for the control block.

10 If `std::make_shared` is used instead,

11 `auto spw = std::make_shared<Widget>();`

12 one allocation suffices. That's because `std::make_shared` allocates a single  
13 chunk of memory to hold both the `Widget` object and the control block. This opti-  
14 mization reduces the static size of the program, because the code contains only  
15 one memory allocation call, and it increases the speed of the executable code, be-  
16 cause memory is allocated only once. Furthermore, using `std::make_shared` ob-  
17 viates the need for some of the bookkeeping information in the control block, po-  
18 tentially reducing the total memory footprint for the program.

19 The efficiency analysis for `std::make_shared` is equally applicable to  
20 `std::allocate_shared`, so the performance advantages of `std::make_shared`  
21 extend to that function, as well.

22 The arguments for preferring `make` functions over direct use of `new` are strong  
23 ones. Despite their software engineering, exception-safety, and efficiency ad-  
24 vantages, however, this Item's guidance is to *prefer* the `make` functions, not to rely  
25 on them exclusively. That's because there are circumstances where they can't or  
26 shouldn't be used.

27 For example, none of the `make` functions permit the specification of custom  
28 deleters (see Items 18 and 19), but both `std::unique_ptr` and  
29 `std::shared_ptr` have constructors that do. Given a custom deleter for a `Widget`,

1 auto widgetDeleter = [](Widget\* pw) { ... };  
2 creating a smart pointer using it is straightforward using new:

3 std::unique\_ptr<Widget, decltype(widgetDeleter)>  
4 upw(new Widget, widgetDeleter);  
5 std::shared\_ptr<Widget> spw(new Widget, widgetDeleter);

6 There's no way to do the same thing with a `make` function.

7 A second limitation of `make` functions stems from a syntactic detail of their imple-  
8 mentations. Item 7 explains that when creating an object whose type overloads  
9 constructors both with and without `std::initializer_list` parameters, creat-  
10 ing the object using braces prefers the `std::initializer_list` constructor,  
11 while creating the object using parentheses calls the non-  
12 `std::initializer_list` constructor. The `make` functions perfect-forward their  
13 parameters to an object's constructor, but do they do so using parentheses or us-  
14 ing braces? For some types, the answer to this question makes a big difference. For  
15 example, in these calls,

16 auto upv = std::make\_unique<std::vector<int>>(10, 20);  
17 auto spv = std::make\_shared<std::vector<int>>(10, 20);  
18 do the resulting smart pointers point to `std::vectors` with 10 elements, each of  
19 value 20, or to `std::vectors` with two elements, one with value 10 and the other  
20 with value 20? Or is the result indeterminate?

21 The good news is that it's not indeterminate: both calls create `std::vectors` of  
22 size 10 with all values set to 20. That means that within the `make` functions, the  
23 perfect forwarding code uses parentheses, not braces. The bad news is that if you  
24 want to construct your pointed-to object using a braced initializer, you must use  
25 `new` directly. Using a `make` function would require the ability to perfect-forward a  
26 braced initializer, but, as Item 30 explains, braced initializers can't be perfect-  
27 forwarded. However, Item 30 also describes a workaround: use `auto` type deduc-  
28 tion to create a `std::initializer_list` object from a braced initializer (see  
29 Item 2), then pass the `auto`-created object through the `make` function:

```
1 // create std::initializer_list
2 auto initList = { 10, 20 };

3 // create std::vector using std::initializer_list ctor
4 auto spv = std::make_shared<std::vector<int>>(initList);

5 For std::unique_ptr, these two scenarios (custom deleters and braced initializ-
6 ers) are the only ones where its make functions are problematic. For
7 std::shared_ptr and its make functions, there are two more. Both are edge cas-
8 es, but some developers live on the edge, and you may be one of them.

9 Some classes define their own versions of operator new and operator delete.
10 The presence of these functions implies that the global memory allocation and
11 deallocation routines for objects of these types are inappropriate. Often, class-
12 specific routines are designed only to allocate and deallocate chunks of memory of
13 precisely the size of objects of the class, e.g., operator new and operator delete
14 for class Widget are often designed only to handle allocation and deallocation of
15 chunks of memory of exactly size sizeof(Widget). Such routines are a poor fit
16 for std::shared_ptr's support for custom allocation (via
17 std::allocate_shared) and deallocation (via custom deleters), because the
18 amount of memory that std::allocate_shared requests isn't the size of the dy-
19 namically allocated object, it's the size of that object plus the size of a control
20 block. Consequently, using make functions to create objects of types with class-
21 specific versions of operator new and operator delete is typically a poor idea.

22 The size and speed advantages of std::make_shared vis-à-vis direct use of new
23 stem from std::shared_ptr's control block being placed in the same chunk of
24 memory as the managed object. When that object's reference count goes to zero,
25 the object is destroyed (i.e., its destructor is called). However, the memory it occu-
26 pies can't be released until the control block has also been destroyed, because the
27 same chunk of dynamically allocated memory contains both.

28 As I noted, the control block contains bookkeeping information beyond just the
29 reference count itself. The reference count tracks how many std::shared_ptrs
30 refer to the control block, but the control block contains a second reference count,
```

1 one that tallies how many `std::weak_ptr`s refer to the control block. This second  
2 reference count is known as the *weak count*.<sup>†</sup> When a `std::weak_ptr` checks to  
3 see if it has expired (see Item 19), it does so by examining the reference count (not  
4 the weak count) in the control block that it refers to. If the reference count is zero  
5 (i.e., if the pointed-to object has no `std::shared_ptr`s referring to it and has thus  
6 been destroyed), the `std::weak_ptr` has expired. Otherwise, it hasn't.

7 As long as `std::weak_ptr`s refer to a control block (i.e., the weak count is greater  
8 than zero), that control block must continue to exist. And as long as a control block  
9 exists, the memory containing it must remain allocated. The memory allocated by a  
10 `std::shared_ptr` `make` function, then, can't be deallocated until the last  
11 `std::shared_ptr` and the last `std::weak_ptr` referring to it have been de-  
12 stroyed.

13 If the object type is quite large and the time between destruction of the last  
14 `std::shared_ptr` and the last `std::weak_ptr` is significant, a lag can occur be-  
15 tween when an object is destroyed and when the memory it occupied is freed:

```
16 class ReallyBigType { ... };

17 auto pBigObj = std::make_shared<ReallyBigType>(); // create very large
18 // object via
19 // std::make_shared

20 ... // create std::shared_ptr and std::weak_ptr to
21 // large object, use them to work with it

22 ... // final std::shared_ptr to object destroyed here,
23 // but std::weak_ptr to it remain

24 ... // during this period, memory formerly occupied
25 // by large object remains allocated
```

---

<sup>†</sup> In practice, the value of the weak count isn't always equal to the number of `std::weak_ptr`s referring to the control block, because library implementers have found ways to slip additional information into the weak count that facilitate better code generation. For purposes of this Item, we'll ignore this and assume that the weak count's value is the number of `std::weak_ptr`s referring to the control block.

```

1 ... // final std::weak_ptr to object destroyed here;
2 // memory for control block and object is released
3 With a direct use of new, the memory for the ReallyBigType object can be re-
4 leased as soon as the last std::shared_ptr to it is destroyed:
5
6 class ReallyBigType { ... }; // as before
7 std::shared_ptr<ReallyBigType> pBigObj(new ReallyBigType);
8 // create very large
9 // object via new
10 ...
11 ... // as before, create std::shared_ptrs and
12 // std::weak_ptrs to object, use them with it
13 ...
14 ... // final std::shared_ptr to object destroyed here,
15 // but std::weak_ptrs to it remain;
16 // memory for object is deallocated
17 ...
18 Should you find yourself in a situation where use of std::make_shared is impos-
19 sible or inappropriate, you'll want to guard yourself against the kind of exception-
20 safety problems we saw earlier. The best way to do that is to make sure that when
21 you use new directly, you immediately pass the result to a smart pointer construc-
22 tor in a statement that does nothing else. This prevents compilers from generating
23 code that could emit an exception between the use of new and invocation of the
24 constructor for the smart pointer that will manage the newed object.
25 As an example, consider a minor revision to the exception-unsafe call to the pro-
26 cessWidget function we examined earlier. This time, we'll specify a custom
27 deleter:
28
29 void processWidget(std::shared_ptr<Widget> spw, // as before
30 int priority);
31 void cusDel(Widget *ptr); // custom
32 Here's the exception-unsafe call:

```

```
1 processWidget(                                // as before,
2     std::shared_ptr<Widget>(new Widget, cusDel), // potential
3     computePriority()                           // resource
4 );  // Leak!
```

5 Recall: if `computePriority` is called after “`new Widget`” but before the  
6 `std::shared_ptr` constructor, and if `computePriority` yields an exception, the  
7 dynamically allocated `Widget` will be leaked.

8 Here the use of a custom deleter precludes use of `std::make_shared`, so the way  
9 to avoid the problem is to put the allocation of the `Widget` and the construction of  
10 the `std::shared_ptr` into their own statement, then call `processWidget` with  
11 the resulting `std::shared_ptr`. Here’s the essence of the technique, though, as  
12 we’ll see in a moment, we can tweak it to improve its performance:

```
13 std::shared_ptr<Widget> spw(new Widget, cusDel);
14 processWidget(spw, computePriority());          // correct, but not
15   // optimal; see below
```

16 This works, because a `std::shared_ptr` assumes ownership of the raw pointer  
17 passed to its constructor, even if that constructor yields an exception. In this ex-  
18 ample, if `spw`’s constructor throws an exception (e.g., due to an inability to dynam-  
19 ically allocate memory for a control block), it’s still guaranteed that `cusDel` will be  
20 invoked on the pointer resulting from “`new Widget`”.

21 The minor performance hitch is that in the exception-unsafe call, we’re passing an  
22 rvalue to `processWidget`,

```
23 processWidget(
24     std::shared_ptr<Widget>(new Widget, cusDel), // arg is rvalue
25     computePriority()
26 );
```

27 but in the exception-safe call, we’re passing an lvalue:

```
28 processWidget(spw, computePriority());          // arg is lvalue
```

29 Because `processWidget`’s `std::shared_ptr` parameter is passed by value, con-  
30 struction from an rvalue entails only a move, while construction from an lvalue  
31 requires a copy. For `std::shared_ptr`, the difference can be significant, because  
32 copying a `std::shared_ptr` requires an atomic increment of its reference count,

1 while moving a `std::shared_ptr` requires no reference count manipulation at  
2 all. For the exception-safe code to achieve the level of performance of the excep-  
3 tion-unsafe code, we need to apply `std::move` to `spw` to turn it into an rvalue (see  
4 Item 23):

```
5 processWidget(std::move(spw),           // both efficient and
6                 computePriority());      // exception-safe
```

7 That's interesting and worth knowing, but it's also typically irrelevant, because  
8 you'll rarely have a reason not to use a `make` function. And unless you have a com-  
9 pelling reason for doing otherwise, using a `make` function is what you should do.

## 10 Things to Remember

- 11   ♦ Compared to direct use of `new`, `make` functions eliminate source code duplica-  
12     tion, improve exception safety, and, for `std::make_shared` and  
13     `std::allocate_shared`, generate code that's smaller and faster.
- 14   ♦ Situations where use of `make` functions is inappropriate include the need to  
15     specify custom deleters and a desire to pass braced initializers.
- 16   ♦ For `std::shared_ptr`s, additional situations where `make` functions may be  
17     ill-advised include (1) classes with custom memory management and (2) sys-  
18     tems with memory concerns, very large objects, and `std::weak_ptr`s that  
19     outlive the corresponding `std::shared_ptr`s.

## 20 Item 22: When using the Pimpl Idiom, define special mem- 21       ber functions in the implementation file.

22 If you've ever had to combat excessive build times, you're familiar with the *Pimpl*  
23 ("pointer to implementation") *Idiom*. That's the technique whereby you replace the  
24 data members of a class with a pointer to an implementation class (or struct), put  
25 the data members that used to be in the primary class into the implementation  
26 class, and access those data members indirectly through the pointer. For example,  
27 suppose `Widget` looks like this:

```
28 class Widget {                         // in header "widget.h"
29 public:
30     Widget();
31     ...
```

```
1 private:  
2     std::string name;  
3     std::vector<double> data;  
4     Gadget g1, g2, g3;           // Gadget is some user-  
5 };                           // defined type  
  
6 Because Widget's data members are of types std::string, std::vector, and  
7 Gadget, headers for those types must be present for Widget to compile, and that  
8 means that Widget clients must #include <string>, <vector>, and gadget.h.  
9 Those headers increase the compilation time for Widget clients, plus they make  
10 those clients dependent on the contents of the headers. If a header's content  
11 changes, Widget clients must recompile. The standard headers <string> and  
12 <vector> don't change very often, but it could be that gadget.h is subject to fre-  
13 quent revision.
```

14 Applying the Pimpl Idiom in C++98 could have Widget replace its data members  
15 with a raw pointer to a struct that has been declared, but not defined:

```
16 class Widget {                  // still in header "widget.h"  
17 public:  
18     Widget();  
19     ~Widget();                 // dtor is needed—see below  
20     ...  
  
21 private:  
22     struct Impl;              // declare implementation struct  
23     Impl *pImpl;              // and pointer to it  
24 };
```

25 Because Widget no longer mentions the types std::string, std::vector, and  
26 Gadget, Widget clients no longer need to #include the headers for these types.  
27 That speeds compilation, and it also means that if something in these headers  
28 changes, Widget clients are unaffected.

29 A type that has been declared, but not defined, is known as an *incomplete type*.  
30 Widget::Impl is such a type. There are very few things you can do with an in-  
31 complete type, but declaring a pointer to it is one of them. The Pimpl Idiom takes  
32 advantage of that.

33 Part 1 of the Pimpl Idiom is the declaration of a data member that's a pointer to an  
34 incomplete type. Part 2 is the dynamic allocation and deallocation of the object

1 that holds the data members that used to be in the original class. The allocation  
2 and deallocation code goes in the implementation file, e.g., for `Widget`, in `widget-  
3 et.cpp`:

```
4 #include "widget.h"           // in impl. file "widget.cpp"
5 #include "gadget.h"
6 #include <string>
7 #include <vector>
8 struct Widget::Impl {        // definition of Widget::Impl
9     std::string name;         // with data members formerly
10    std::vector<double> data; // in Widget
11    Gadget g1, g2, g3;
12 };
13 Widget::Widget()            // allocate data members for
14 : pImpl(new Impl)          // this Widget object
15 {}
16 Widget::~Widget()          // destroy data members for
17 { delete pImpl; }           // this object
```

18 Here I'm showing `#include` directives to make clear that the overall dependen-  
19 cies on the headers for `std::string`, `std::vector`, and `Gadget` continue to ex-  
20 ist. However, these dependencies have been moved from `widget.h` (which is visi-  
21 ble to and used by `Widget` clients) to `widget.cpp` (which is visible to and used  
22 only by the `Widget` implementer). I've also highlighted the code that dynamically  
23 allocates and deallocates the `Impl` object. The need to deallocate this object when  
24 a `Widget` is destroyed is what necessitates the `Widget` destructor.

25 But I've shown you C++98 code, and that reeks of a bygone millennium. It uses raw  
26 pointers and raw `new` and raw `delete` and it's all just so...raw. This chapter is built  
27 on the idea that smart pointers are preferable to raw pointers, and if what we want  
28 is to dynamically allocate a `Widget::Impl` object inside the `Widget` constructor  
29 and have it destroyed at the same time the `Widget` is, `std::unique_ptr` (see  
30 Item 18) is precisely the tool we need. Replacing the raw `pImpl` pointer with a  
31 `std::unique_ptr` yields this code for the header file,

```
32 class Widget {                // in "widget.h"
33 public:
34     Widget();
35     ...
```

```
1 private:  
2     struct Impl;  
3     std::unique_ptr<Impl> pImpl;           // use smart pointer  
4 };   // instead of raw pointer
```

5 and this for the implementation file:

```
6 #include "widget.h"                         // in "widget.cpp"  
7 #include "gadget.h"  
8 #include <string>  
9 #include <vector>  
10 struct Widget::Impl {                      // as before  
11     std::string name;  
12     std::vector<double> data;  
13     Gadget g1, g2, g3;  
14 };  
15 Widget::Widget()                          // per Item 21, create  
16 : pImpl(std::make_unique<Impl>())        // std::unique_ptr  
17 {}   // via std::make_unique
```

18 You'll note that the `Widget` destructor is no longer present. That's because we  
19 have no code to put into it. `std::unique_ptr` automatically deletes what it points  
20 to when it (the `std::unique_ptr`) is destroyed, so we need not delete anything  
21 ourselves. That's one of the attractions of smart pointers: they eliminate the need  
22 for us to sully our hands with manual resource release.

23 This code compiles, but, alas, the most trivial client use doesn't:

```
24 #include "widget.h"  
25 Widget w;                                // error!
```

26 The error message you receive depends on the compiler you're using, but the text  
27 generally mentions something about applying `sizeof` or `delete` to an incomplete  
28 type. Those operations aren't among the things you can do with such types.

29 This apparent failure of the Pimpl Idiom using `std::unique_ptr`s is alarming,  
30 because (1) `std::unique_ptr` is advertised as supporting incomplete types, and  
31 (2) the Pimpl Idiom is one of `std::unique_ptr`s most common use cases. Fortu-  
32 nately, getting the code to work is easy. All that's required is a basic understanding  
33 of the cause of the problem.

1 The issue arises due to the code that's generated when `w` is destroyed (e.g., goes  
2 out of scope). At that point, its destructor is called. In the class definition using  
3 `std::unique_ptr`, we didn't declare a destructor, because we didn't have any  
4 code to put into it. In accord with the usual rules for compiler-generated special  
5 member functions (see Item 17), the compiler generates a destructor for us. Within  
6 that destructor, the compiler inserts code to call the destructor for `Widget`'s da-  
7 ta member `pImpl`. `pImpl` is a `std::unique_ptr<Widget::Impl>`, i.e., a  
8 `std::unique_ptr` using the default deleter. The default deleter is a function that  
9 uses `delete` on the raw pointer inside the `std::unique_ptr`. Prior to using `de-`  
10 `lete`, however, implementations typically have the default deleter employ C++11's  
11 `static_assert` to ensure that the raw pointer doesn't point to an incomplete  
12 type. When the compiler generates code for the destruction of the `Widget w`, then,  
13 it generally encounters a `static_assert` that fails, and that's usually what leads  
14 to the error message. This message is associated with the point where `w` is de-  
15 stroyed, because `Widget`'s destructor, like all compiler-generated special member  
16 functions, is implicitly `inline`. The message itself often refers to the line where `w`  
17 is created, because it's the source code explicitly creating the object that leads to  
18 its later implicit destruction.

19 To fix the problem, you just need to make sure that at the point where the code to  
20 destroy the `std::unique_ptr<Widget::Impl>` is generated, `Widget::Impl` is a  
21 complete type. The type becomes complete when its definition has been seen, and  
22 `Widget::Impl` is defined inside `widget.cpp`. The key to successful compilation,  
23 then, is to have the compiler see the body of `Widget`'s destructor (i.e., the place  
24 where the compiler will generate code to destroy the `std::unique_ptr` data  
25 member) only inside `widget.cpp` after `Widget::Impl` has been defined.

26 Arranging for that is simple. Declare `Widget`'s destructor in `widget.h`, but don't  
27 define it there:

```
28 class Widget {                      // as before, in "widget.h"  
29 public:  
30     Widget();  
31     ~Widget();                     // declaration only  
32     ...
```

```

1 private:                                // as before
2     struct Impl;
3     std::unique_ptr<Impl> pImpl;
4 };
5 Define it in widget.cpp after Widget::Impl has been defined:
6 #include "widget.h"                      // as before, in "widget.cpp"
7 #include "gadget.h"
8 #include <string>
9 #include <vector>
10 struct Widget::Impl {                  // as before, definition of
11     std::string name;                  // Widget::Impl
12     std::vector<double> data;
13     Gadget g1, g2, g3;
14 };
15 Widget::Widget()                      // as before
16 : pImpl(std::make_unique<Impl>())
17 {}
18 Widget::~Widget()                    // ~Widget definition
19 {}


```

20 This works well, and it requires the least typing, but if you want to emphasize that  
21 the compiler-generated destructor would do the right thing—that the only reason  
22 you declared it was to cause its definition to be generated in `Widget`'s implemen-  
23 tation file, you can define the destructor body with “`= default`”:

```

24 Widget::~Widget() = default;          // same effect as above
25 Classes using the Pimpl Idiom are natural candidates for move support, because
26 compiler-generated move operations do exactly what's desired: perform a move
27 on the underlying std::unique_ptr. As Item 17 explains, the declaration of a de-
28 structor in Widget prevents compilers from generating the move operations, so if
29 you want move support, you must declare the functions yourself. Given that the
30 compiler-generated versions would behave correctly, you're likely to be tempted
31 to implement them as follows:


```

```

32 class Widget {                        // still in
33 public:                                // "widget.h"
34     Widget();
35     ~Widget();


```

```
1  Widget(Widget&& rhs) = default;           // right idea,
2  Widget& operator=(Widget&& rhs) = default; // wrong code!
3
4  ...
5  private:                                // as before
6      struct Impl;
7      std::unique_ptr<Impl> pImpl;
8  };
9
10 This approach leads to the same kind of problem as declaring the class without a
11 destructor, and for the same fundamental reason. The compiler-generated move
12 assignment operator needs to destroy the object pointed to by pImpl before reas-
13 signing it, but in the Widget header file, pImpl points to an incomplete type. The
14 situation is different for the move constructor. The problem there is that compilers
15 typically generate code to destroy pImpl in the event that an exception arises in-
16 side the move constructor, and destroying pImpl requires that Impl be complete.
```

Because the problem is the same as before, so is the fix: move the definition of the move operations into the implementation file:

```
17 class Widget {                           // still in "widget.h"
18 public:
19     Widget();
20     ~Widget();
21
22     Widget(Widget&& rhs);           // declarations
23     Widget& operator=(Widget&& rhs); // only
24
25     ...
26
27 private:                                // as before
28     struct Impl;
29     std::unique_ptr<Impl> pImpl;
30 };
31
32 #include <string>                      // as before,
33 ...                                      // in "widget.cpp"
34
35 struct Widget::Impl { ... };            // as before
36
37 Widget::Widget()                      // as before
38 : pImpl(std::make_unique<Impl>())
39 {}
40
41 Widget::~Widget() = default;          // as before
```

```
1 Widget::Widget(Widget&& rhs) = default;           // define-
2 Widget& Widget::operator=(Widget&& rhs) = default; // tions
3
4 The Pimpl Idiom is a way to reduce compilation dependencies between a class's
5 implementation and the class's clients, but, conceptually, use of the idiom doesn't
6 change what the class represents. The original Widget class contained
7 std::string, std::vector, and Gadget data members, and, assuming that
8 Gadgets, like std::strings and std::vectors, can be copied, it would make
9 sense for Widget to support the copy operations. We have to write these functions
10 ourselves, because (1) compilers won't generate copy operations for classes with
11 move-only types like std::unique_ptr and (2) even if they did, the generated
12 functions would copy only the std::unique_ptr (i.e., perform a shallow copy),
13 and we want to copy what the pointer points to (i.e., perform a deep copy).
```

13 In a ritual that is by now familiar, we declare the functions in the header file and  
14 implement them in the implementation file:

```
15 class Widget {                                // still in "widget.h"
16 public:
17 ...
18 ...
19 ...
20 ...
21 ...
22 ...
23 ...
24
25 ...
26 ...
27 ...
28 ...
29 ...
30 ...
31 ...
32 ...
33 ...
34 *
```

```
15 class Widget {                                // still in "widget.h"
16 public:
17 ...
18 ...
19 ...
20 ...
21 ...
22 ...
23 ...
24
25 ...
26 ...
27 ...
28 ...
29 ...
30 ...
31 ...
32 ...
33 ...
34 *
```

```
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 21 to prefer use of `std::make_unique`  
10 over direct use of `new`.

11 For purposes of implementing the Pimpl Idiom, `std::unique_ptr` is the smart  
12 pointer to use, because the `pImpl` pointer inside an object (e.g., inside a `Widget`)  
13 has exclusive ownership of the corresponding implementation object (e.g., the  
14 `Widget::Impl` object). Still, it's interesting to note that if we were to use  
15 `std::shared_ptr` instead of `std::unique_ptr` for `pImpl`, we'd find that the ad-  
16 vice of this Item no longer applied. There'd be no need to declare a destructor in  
17 `Widget`, and without a user-declared destructor, compilers would happily gener-  
18 ate the move operations, which would do exactly what we'd want them to. That is,  
19 given this code in `widget.h`,

```
20 class Widget {                         // in "widget.h"
21 public:
22     Widget();
23     ...                                // no declarations for dtor
24   // or move operations
25 private:
26     struct Impl;
27     std::shared_ptr<Impl> pImpl;        // std::shared_ptr
28 };
```

29 and this client code that `#includes` `widget.h`,

```
30 Widget w1;
31 auto w2(std::move(w1));                // move-construct w2
32 w1 = std::move(w2);                    // move-assign w1
```

33 everything would compile and run as we'd hope: `w1` would be default-constructed,  
34 its value would be moved into `w2`, that value would be moved back into `w1`, and

1 then both w1 and w2 would be destroyed (thus causing the pointed-to `Widget::Impl` object to be destroyed).

3 The difference in behavior between `std::unique_ptr` and `std::shared_ptr`  
4 for `pImpl` pointers stems from the differing ways these smart pointers support  
5 custom deleters. For `std::unique_ptr`, the type of the deleter is part of the type  
6 of the smart pointer, and this makes it possible for compilers to generate smaller  
7 runtime data structures and faster runtime code. A consequence of this greater  
8 efficiency is that pointed-to types must be complete when compiler-generated  
9 special functions (e.g., destructors or move operations) are used. For  
10 `std::shared_ptr`, the type of the deleter is not part of the type of the smart  
11 pointer. This necessitates larger runtime data structures and somewhat slower  
12 code, but pointed-to types need not be complete when compiler-generated special  
13 functions are employed.

14 For the Pimpl Idiom, there's not really a trade-off between the characteristics of  
15 `std::unique_ptr` and `std::shared_ptr`, because the relationship between  
16 classes like `Widget` and classes like `Widget::Impl` is exclusive ownership, and  
17 that makes `std::unique_ptr` the proper tool for the job. Nevertheless, it's worth  
18 knowing that in other situations—situations where shared ownership exists (and  
19 `std::shared_ptr` is hence a fitting design choice), there's no need to jump  
20 through the function-definition hoops that use of `std::unique_ptr` entails.

## 21 **Things to Remember**

- 22   ♦ The Pimpl Idiom decreases build times by reducing compilation dependencies  
23     between class clients and class implementations.
- 24   ♦ For `std::unique_ptr` `pImpl` pointers, declare special member functions in  
25     the class header, but implement them in the implementation file. Do this even  
26     if the default function implementations are acceptable.
- 27   ♦ 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. Move semantics also enables the creation of  
10   move-only types, such as `std::unique_ptr`, `std::future`, and  
11   `std::thread`.
- 12   • **Perfect forwarding** makes it possible to write function templates that take  
13   arbitrary arguments and forward them to other functions such that the target  
14   functions receive exactly the same arguments as were passed to the forward-  
15   ing functions.

16   Rvalue references are the glue that ties these two rather disparate features together.  
17   They're the underlying language mechanism that makes both move semantics  
18   and perfect forwarding possible.

19   The more experience you have with these features, the more you realize that your  
20   initial impression was based on only the metaphorical tip of the proverbial ice-  
21   berg. The world of move semantics, perfect forwarding, and rvalue references is  
22   more nuanced than it appears. `std::move` doesn't move anything, for example,  
23   and perfect forwarding is imperfect. Move operations aren't always cheaper than  
24   copying; when they are, they're not always as cheap as you'd expect; and they're  
25   not always called in a context where moving is valid. The construct "`type&&`"  
26   doesn't always represent an rvalue reference.

27   No matter how far you dig into these features, it can seem that there's always more  
28   to uncover. Fortunately, there is a limit to their depths. This chapter will take you  
29   to the bedrock. Once you arrive, this part of C++11 will make a lot more sense.

1 You'll know the usage conventions for `std::move` and `std::forward`, for example. You'll be comfortable with the ambiguous nature of "`type&&`". You'll understand the reasons for the surprisingly varied behavioral profiles of move operations. All those pieces will come together and fall into place. At that point, you'll be back where you started, because move semantics, perfect forwarding, and rvalue references will once again seem pretty straightforward. But this time, they'll stay that way.

8 In the Items in this chapter, it's especially important to bear in mind that a parameter is always an lvalue, even if its type is an rvalue reference. That is, given

10 `void f(Widget&& w);`

11 the parameter `w` is an lvalue, even though its type is rvalue-reference-to-`Widget`.  
12 (If this surprises you, please review the overview of lvalues and rvalues that begins on page 5.)

#### 14 **Item 23: Understand `std::move` and `std::forward`.**

15 It's useful to approach `std::move` and `std::forward` in terms of what they *don't* do. `std::move` doesn't move anything. `std::forward` doesn't forward anything.  
17 At run time, neither does anything at all. They generate no executable code. Not a single byte.

19 `std::move` and `std::forward` are merely functions (actually function templates) that perform casts. `std::move` unconditionally casts its argument to an rvalue, while `std::forward` performs this cast only if a particular condition is fulfilled.  
22 That's it. The explanation leads to a new set of questions, but, fundamentally, that's the complete story.

24 To make the story more concrete, here's a sample implementation of `std::move` in C++11. It's not fully conforming to the details of the Standard, but it's very close.

```
26 template<typename T> // in namespace std
27 typename remove_reference<T>::type&&
28 move(T&& param)
29 {
30     using ReturnType = // alias declaration;
31     typename remove_reference<T>::type&&; // see Item 9
```

```

1     return static_cast<ReturnType>(param);
2 }

3 I've highlighted two parts of the code for you. One is the name of the function, be-
4 cause the return type specification is rather noisy, and I don't want you to lose
5 your bearings in the din. The other thing is the cast that comprises the essence of
6 the function. As you can see, std::move takes a reference to an object (a universal
7 reference, to be precise (see Item 24)) and it returns a reference to the same ob-
8 ject.

9 The "&&" part of the function's return type implies that std::move returns an
10 rvalue reference, but, as Item 28 explains, if the type T happens to be an lvalue ref-
11 erence, T&& would become an lvalue reference. To prevent this from happening,
12 the type trait (see Item 9) std::remove_reference is applied to T, thus ensuring
13 that "&&" is applied to a type that isn't a reference. That guarantees that
14 std::move truly returns an rvalue reference, and that's important, because rvalue
15 references returned from functions are rvalues. Thus, std::move casts its argu-
16 ment to an rvalue, and that's all it does.

17 As an aside, std::move can be implemented with less fuss in C++14. Thanks to
18 function return type deduction (see Item 3) and to the Standard Library's alias
19 template std::remove_reference_t (see Item 9), std::move can be written
20 this way:

21 template<typename T>   // C++14; still in
22 decltype(auto) move(T&& param)                                // namespace std
23 {
24     using ReturnType = remove_reference_t<T>&&;
25     return static_cast<ReturnType>(param);
26 }

```

27 Easier on the eyes, no?

28 Because std::move does nothing but cast its argument to an rvalue, there have
29 been suggestions that a better name for it might have been something like rval-
30 ue\_cast. Be that as it may, the name we have is std::move, so it's important to
31 remember what std::move does and doesn't do. It does cast. It doesn't move.

1 Of course, rvalues are candidates for moving, so applying `std::move` to an object  
2 tells the compiler that that object is eligible to be moved from. That's why  
3 `std::move` has the name it does: to make it easy to designate objects that may be  
4 moved from.

5 In truth, rvalues are only *usually* candidates for moving. Suppose you're writing a  
6 function taking a `std::string` parameter, and you know that inside your func-  
7 tion, you'll copy that parameter. Flush with the advice proffered by Item 41, you  
8 declare a by-value parameter:

```
9 void f(std::string s);           // s to be copied, so per Item 41,  
10                           // pass by value
```

11 But suppose you also know that inside `f`, you need only to read `s`'s value; you'll  
12 never need to modify it. In accord with the time-honored tradition of using `const`  
13 whenever possible, you revise your declaration such that `s` is `const`:

```
14 void f(const std::string s);
```

15 Finally, assume that at the end of `f`, `s` will be copied into some data structure. Ex-  
16 cept that you don't want to pay for the copy, so, again in accord with Item 41, you  
17 apply `std::move` to `s` to turn it into an rvalue:

```
18 struct SomeDataStructure {  
19     std::string name;  
20     ...  
21 };  
22 SomeDataStructure sds;  
23 void f(const std::string s)  
24 {  
25     ...                                // read operations on s  
26     sds.name = std::move(s); // "move" s into sds.name; this code  
27 }   // doesn't do what it seems to!
```

28 This code compiles. This code links. This code runs. This code sets `sds.name` to  
29 the value you expect. The only thing separating this code from a perfect realization  
30 of your vision is that `s` is not moved into `sds.name`, it's *copied*. Sure, `s` is cast to an  
31 rvalue by `std::move`, but `s` is declared as a `const std::string`, so before the

1 cast, `s` is an lvalue `const std::string`, and the result of the cast is an rvalue  
2 `const std::string`, but throughout it all, the `constness` remains.

3 Consider the effect that has when compilers have to determine which  
4 `std::string` assignment operator to call. There are two possibilities:

```
5 class string {           // std::string is actually a
6 public:                 // typedef for std::basic_string<char>
7 ...
8     string& operator=(const string& rhs); // copy assignment
9     string& operator=(string&& rhs);      // move assignment
10 ...
11 };
```

12 In our function `f`, the result of `std::move(s)` is an rvalue of type `const`  
13 `std::string`. That rvalue can't be passed to `std::string`'s move assignment  
14 operator, because the move assignment operator takes an rvalue reference to a  
15 *non-const* `std::string`. The rvalue can, however, be passed to the copy assign-  
16 ment operator, because an lvalue reference-to-`const` is permitted to bind to a  
17 `const` rvalue. The statement

```
18 sds.name = std::move(s);
```

19 therefore invokes the *copy* assignment operator in `std::string`, even though `s`  
20 has been cast to an rvalue! Such behavior is essential to maintaining `const-`  
21 correctness. Moving a value out of an object generally modifies the object, so the  
22 language should not permit `const` objects to be passed to functions (such as move  
23 assignment operators) that could modify them.

24 There are two lessons to be drawn from this example. First, don't declare objects  
25 `const` if you want to be able to move from them. Move requests on `const` objects  
26 are silently transformed into copy operations. Second, `std::move` not only doesn't  
27 actually move anything, it doesn't even guarantee that the object it's casting will be  
28 eligible to be moved. The only thing you know for sure about the result of applying  
29 `std::move` to an object is that it's an rvalue.

30 The story for `std::forward` is similar to that for `std::move`, but whereas  
31 `std::move` *unconditionally* casts its argument to an rvalue, `std::forward` does it  
32 only under certain conditions. `std::forward` is a *conditional* cast. To understand

1 when it casts and when it doesn't, recall how `std::forward` is typically used. The  
2 most common scenario is a function template taking a universal reference parameter  
3 that is to be passed to another function:

```
4 void process(const Widget& lvalArg);      // process lvalues
5 void process(Widget&& rvalArg);           // process rvalues
6 template<typename T>                     // template that passes
7 void logAndProcess(T&& param)            // param to process
8 {
9     auto now =                                // get current time
10    std::chrono::system_clock::now();
11
12    makeLogEntry("Calling 'process'", now);
13    process(std::forward<T>(param));
14 }
```

15 Consider two calls to `logAndProcess`, one with an lvalue, the other with an rvalue:

```
17 Widget w;
18 logAndProcess(w);                         // call with lvalue
19 logAndProcess(std::move(w));               // call with rvalue
```

20 Inside `logAndProcess`, the parameter `param` is passed to the function `process`.  
21 `process` is overloaded for lvalues and rvalues. When we call `logAndProcess`  
22 with an lvalue, we naturally expect that lvalue to be forwarded to `process` as an  
23 lvalue, and when we call `logAndProcess` with an rvalue, we expect the rvalue  
24 overload of `process` to be invoked.

25 But `param`, like all function parameters, is an lvalue. Every call to `process` inside  
26 `logAndProcess` will thus want to invoke the lvalue overload for `process`. To prevent  
27 this, we need a mechanism for `param` to be cast to an rvalue if and only if the  
28 argument with which `param` was initialized—the argument passed to `logAndProcess`—was an rvalue.  
29 This is precisely what `std::forward` does. That's why `std::forward` is a *conditional* cast: it casts to an rvalue only if its argument  
30 was initialized with an rvalue.  
31

32 You may wonder how `std::forward` can know whether its argument was initialized  
33 with an rvalue. In the code above, for example, how can `std::forward` tell

1 whether `param` was initialized with an lvalue or an rvalue? The brief answer is that  
2 that information is encoded in `logAndProcess`'s template parameter `T`. That pa-  
3 rameter is passed to `std::forward`, which recovers the encoded information. For  
4 details on exactly how that works, consult Item 28.

5 Given that both `std::move` and `std::forward` boil down to casts, the only differ-  
6 ence being that `std::move` always casts, while `std::forward` only sometimes  
7 does, you might ask whether we can dispense with `std::move` and just use  
8 `std::forward` everywhere. From a purely technical perspective, the answer is  
9 yes: `std::forward` can do it all. `std::move` isn't necessary. Of course, neither  
10 function is really *necessary*, because we could write casts everywhere, but I hope  
11 we agree that that would be, well, yucky.

12 `std::move`'s attractions are convenience, reduced likelihood of error, and greater  
13 clarity. Consider a class where we want to track how many times the move  
14 constructor is called. A `static` counter that's incremented during move construction  
15 is all we need. Assuming the only non-static data in the class is a `std::string`,  
16 here's the conventional way (i.e., using `std::move`) to implement the move con-  
17 structor:

```
18 class Widget {
19 public:
20     Widget(Widget&& rhs)
21     : s(std::move(rhs.s))
22     { ++moveCtorCalls; }
23 ...
24 private:
25     static std::size_t moveCtorCalls;
26     std::string s;
27 };
```

28 To implement the same behavior with `std::forward`, the code would look like  
29 this:

```
30 class Widget {
31 public:
32     Widget(Widget&& rhs)           // unconventional,
33     : s(std::forward<std::string>(rhs.s)) // undesirable
34     { ++moveCtorCalls; }                // implementation
```

1        ...

2        };

3 Note first that `std::move` requires only a function argument (i.e., `rhs.s`), while  
4 `std::forward` requires both a function argument (`rhs.s`) and a template type  
5 argument (`std::string`). Then note that the type we pass to `std::forward`  
6 should be a non-reference, because that's the convention for encoding that the ar-  
7 gument being passed is an rvalue (see Item 28). Together, this means that  
8 `std::move` requires less typing than `std::forward`, and it spares us the trouble  
9 of passing a type argument that encodes that the argument we're passing is an  
10 rvalue. It also eliminates the possibility of our passing an incorrect type (e.g.  
11 `std::string&`, which would result in the data member `s` being copy-constructed  
12 instead of move-constructed).

13 More importantly, the use of `std::move` conveys an unconditional cast to an rval-  
14 ue, while the use of `std::forward` indicates a cast to an rvalue only for references  
15 to which rvalues have been bound. Those are two very different actions. The first  
16 one typically sets up a move, while the second one just passes—*forwards*—an ob-  
17 ject to another function in a way that retains its original lvalueness or rvalueness.  
18 Because these actions are so different, it's good that we have two different func-  
19 tions (and function names) to distinguish them.

## 20 Things to Remember

- 21     ♦ `std::move` performs an unconditional cast to an rvalue. In and of itself, it  
22        doesn't move anything.
- 23     ♦ `std::forward` casts its argument to an rvalue only if that argument is bound  
24        to an rvalue.
- 25     ♦ Neither `std::move` nor `std::forward` do anything at run time.

## 26 Item 24: Distinguish universal references from rvalue ref- 27 erences.

28 It's been said that the truth shall set you free, but under the right circumstances, a  
29 well-chosen lie can be equally liberating. This Item is such a lie. Because we're

1 dealing with software, however, let's eschew the word "lie" and instead say that  
2 this Item comprises an *abstraction*.

3 To declare an rvalue reference to some type T, you write T&&. It thus seems reasonable  
4 to assume that if you see "T&&" in source code, you're looking at an rvalue  
5 reference. Alas, it's not quite that simple:

```
6 void f(Widget&& param);           // rvalue reference
7 Widget&& var1 = Widget();          // rvalue reference
8 auto&& var2 = var1;               // not rvalue reference
9 template<typename T>
10 void f(std::vector<T>&& param); // rvalue reference
11 template<typename T>
12 void f(T&& param);            // not rvalue reference
```

13 In fact, "T&&" has two different meanings. One is rvalue reference, of course. Such  
14 references behave exactly the way you expect: they bind only to rvalues, and their  
15 primary *raison d'être* is to identify objects that may be moved from.

16 The other possible meaning for "T&&" is *either* rvalue reference *or* lvalue reference.  
17 Such references look like rvalue references in the source code (i.e., "T&&"), but they  
18 can behave as if they were lvalue references (i.e., "T&"). Their dual nature permits  
19 them to bind to rvalues (like rvalue references) as well as lvalues (like lvalue references). Furthermore,  
20 they can bind to const or non-const objects, to volatile or non-volatile objects, even to objects that are both const and volatile. They  
21 can bind to virtually *anything*. Such unprecedently flexible references deserve a  
22 name of their own. I call them *universal references*.

24 Universal references arise in two contexts. The most common is function template  
25 parameters, such as this example from the sample code above:

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

28 The second context is auto declarations, including this one from the sample code  
29 above:

```
30 auto&& var2 = var1;          // var2 is a universal reference
```

1 What these contexts have in common is the presence of *type deduction*. In the tem-  
2 plate `f`, the type of `param` is being deduced, and in the declaration for `var2`, `var2`'s  
3 type is being deduced. Compare that with the following examples (also from the  
4 sample code above), where type deduction is missing. If you see “`T&&`” without  
5 type deduction, you’re looking at an rvalue reference:

```
6 void f(Widget&& param);           // no type deduction;  
7                                     // param is an rvalue reference  
8 Widget&& var1 = Widget();          // no type deduction;  
9                                     // var1 is an rvalue reference
```

10 Because universal references are references, they must be initialized. The initializ-  
11 er for a universal reference determines whether it represents an rvalue reference  
12 or an lvalue reference. If the initializer is an rvalue, the universal reference corre-  
13 sponds to an rvalue reference. If the initializer is an lvalue, the universal reference  
14 corresponds to an lvalue reference. For universal references that are function pa-  
15 rameters, the initializer is provided at the call site:

```
16 template<typename T>  
17 void f(T&& param);      // param is a universal reference  
18 Widget w;  
19 f(w);                  // lvalue passed to f; param's type is  
20                         // Widget& (i.e., an lvalue reference)  
21 f(std::move(w));        // rvalue passed to f; param's type is  
22                         // Widget&& (i.e., an rvalue reference)
```

23 For a reference to be universal, type deduction is necessary, but it’s not sufficient.  
24 The *form* of the reference declaration must also be correct, and that form is quite  
25 constrained. It must be precisely “`T&&`”. Look again at this example from the sam-  
26 ple code we saw earlier:

```
27 template<typename T>  
28 void f(std::vector<T>&& param); // param is an rvalue reference
```

29 When `f` is invoked, the type `T` will be deduced (unless the caller explicitly specifies  
30 it, an edge case we’ll not concern ourselves with). But the form of `param`’s type  
31 declaration isn’t “`T&&`,” it’s “`std::vector<T>&&`.” That rules out the possibility  
32 that `param` is a universal reference. `param` is therefore an rvalue reference, some-

1 thing that your compilers will be happy to confirm for you if you try to pass an  
2 lvalue to `f`:

```
3 std::vector<int> v;
4 f(v);                                // error! can't bind lvalue to
5                               // rvalue reference
```

6 Even the simple presence of a `const` qualifier is enough to disqualify a reference  
7 from being universal:

```
8 template<typename T>
9 void f(const T&& param);           // param is an rvalue reference
```

10 If you're in a template and you see a function parameter of type "`T&&`," you might  
11 think you can assume that it's a universal reference. You can't. That's because being  
12 in a template doesn't guarantee the presence of type deduction. Consider this  
13 `push_back` member function in `std::vector`:

```
14 template<class T, class Allocator = allocator<T>> // from C++
15 class vector {                                     // Standards
16 public:
17     void push_back(T&& x);
18     ...
19 };
```

20 `push_back`'s parameter certainly has the right form for a universal reference, but  
21 there's no type deduction in this case. That's because `push_back` can't exist without  
22 a particular `vector` instantiation for it to be part of, and the type of that instantia-  
23 tion fully determines the declaration for `push_back`. That is, saying

```
24 std::vector<Widget> v;
```

25 causes the `std::vector` template to be instantiated as follows:

```
26 class vector<Widget, allocator<Widget>> {
27 public:
28     void push_back(Widget&& x);           // rvalue reference
29     ...
30 };
```

31 Now you can see clearly that `push_back` employs no type deduction. This  
32 `push_back` for `vector<T>` (there are two—the function is overloaded) always  
33 declares a parameter of type rvalue-reference-to-`T`.

1 In contrast, the conceptually similar `emplace_back` member function in  
2 `std::vector` *does* employ type deduction:

```
3 template<class T, class Allocator = allocator<T>> // still from
4 class vector { // C++
5 public: // Standards
6     template <class... Args>
7     void emplace_back(Args&&... args);
8     ...
9 }
```

10 Here, the type parameter `Args` is independent of `vector`'s type parameter `T`, so  
11 `Args` must be deduced each time `emplace_back` is called. (Okay, `Args` is really a  
12 parameter pack, not a type parameter, but for purposes of this discussion, we can  
13 treat it as if it were a type parameter.)

14 The fact that `emplace_back`'s type parameter is named `Args`, yet it's still a uni-  
15 versal reference, reinforces my earlier comment that it's the *form* of a universal  
16 reference that must be "`T&&`". There's no requirement that you use the name `T`. For  
17 example, the following template takes a universal reference, because the form  
18 ("`type&&`") is right, and `param`'s type will be deduced (again, excluding the corner  
19 case where the caller explicitly specifies the type):

```
20 template<typename MyTemplateName> // param is a
21 void someFunc(MyTemplateName&& param); // universal reference
```

22 I remarked earlier that `auto` variables can also be universal references. To be  
23 more precise, variables declared with the type `auto&&` are universal references,  
24 because type deduction takes place and they have the correct form ("`T&&`"). `auto`  
25 universal references are not as common as universal references used for function  
26 template parameters, but they do crop up from time to time in C++11. They crop  
27 up a lot more in C++14, because C++14 lambda expressions may declare `auto&&`  
28 parameters. For example, if you wanted to write a C++14 lambda to record the  
29 time taken in an arbitrary function invocation, you could do this:

```
30 auto timeFuncInvocation =
31     [](&func, &params)
32     {
33         start timer;
34         std::forward<decltype(func)>(func)( // invoke func
35             std::forward<decltype(params)>(params)... // on params in
```

```

1      );
2      stop timer and record elapsed time;
3  };

4 If your reaction to the “std::forward<decltype(bLah bLah bLah)>” code in-
5 side the lambda is, “What the...?!”, that probably just means you haven’t yet read
6 Item 33. Don’t worry about it. The important thing in this Item is the auto&&
7 parameters that the lambda declares. func is a universal reference that can be bound
8 to any callable object, lvalue or rvalue. args is zero or more universal references
9 (i.e., a universal reference parameter pack) that can be bound to any number of
10 objects of arbitrary types. The result, thanks to auto universal references, is that
11 timeFuncInvocation can time pretty much any function execution. (For infor-
12 mation on the difference between “any” and “pretty much any,” turn to Item 30.)

13 Bear in mind that this entire Item—the foundation of universal references—is a
14 lie...er, an abstraction. The underlying truth is known as reference-collapsing, a
15 topic to which Item 28 is dedicated. But the truth doesn’t make the abstraction any
16 less useful. Distinguishing between rvalue references and universal references will
17 help you read source code more accurately (“Does that T&& I’m looking at bind to
18 rvalues only or to everything?”), and it will avoid ambiguities when you communi-
19 cate with your colleagues (“I’m using a universal reference here, not an rvalue ref-
20 erence...”). It will also allow you to make sense of Items 25 and 26, which rely on
21 the distinction. So embrace the abstraction. Revel in it. Just as Newton’s laws of
22 motion (which are technically incorrect) are typically just as useful as and easier to
23 apply than Einstein’s theory of general relativity (“the truth”), so is the notion of
24 universal references normally preferable to working through the details of refer-
25 ence-collapsing.

26 Things to Remember
27   • If a function template parameter has type T&& for a deduced type T, or if an
28     object is declared using auto&&, the parameter or object is a universal refer-
29     ence.
30   • If the form of the type declaration isn’t precisely type&&, or if type deduction
31     does not occur, type&& denotes an rvalue reference.

```

- 1   ♦ Universal references correspond to rvalue references if they're initialized with  
2   rvalues. They correspond to lvalue references if they're initialized with lvalues.

3   **Item 25: Use `std::move` on rvalue references, `std::forward`  
4   on universal references.**

5   Rvalue references bind only to objects that are candidates for moving. If you have  
6   an rvalue reference parameter, you *know* that the object it's bound to may be  
7   moved:

```
8   class Widget {  
9     Widget(Widget&& rhs);           // rhs definitely refers to an  
10  ...                           // object eligible for moving  
11 };
```

12 That being the case, you'll want to pass such objects to other functions in a way  
13 that permits those functions to take advantage of the object's rvaluueness. The way  
14 to do that is to cast parameters bound to such objects to rvalues. As Item 23 ex-  
15 plains, that's not only what `std::move` does, it's what it was created for:

```
16 class Widget {  
17 public:  
18   Widget(Widget&& rhs)           // rhs is rvalue reference  
19   : name(std::move(rhs.name)),  
20     p(std::move(rhs.p))  
21   { ... }  
22   ...  
23 private:  
24   std::string name;  
25   std::shared_ptr<SomeDataStructure> p;  
26 };
```

27 A universal reference, on the other hand (see Item 24), *might* be bound to an ob-  
28 ject that's eligible for moving. Universal references should be cast to rvalues only if  
29 they were initialized with rvalues. Item 23 explains that this is precisely what  
30 `std::forward` does:

```
31 class Widget {  
32 public:  
33   template<typename T>  
34   void setName(T& newName)        // newName is  
35   { name = std::forward<T>(newName); } // universal reference
```

```
1     ...
2 };
3 In short, rvalue references should be unconditionally cast to rvalues (via
4 std::move) when forwarding them to other functions, because they're always
5 bound to rvalues, and universal references should be conditionally cast to rvalues
6 (via std::forward) when forwarding them, because they're only sometimes
7 bound to rvalues.
```

```
8 Item 23 explains that using std::forward on rvalue references can be made to
9 exhibit the proper behavior, but the source code is wordy, error-prone, and unidi-
10 omatic, so you should avoid using std::forward with rvalue references. Even
11 worse is the idea of using std::move with universal references, because that can
12 have the effect of unexpectedly modifying lvalues (e.g., local variables):
```

```
13 class Widget {
14 public:
15     template<typename T>
16     void setName(T&& newName)           // universal reference
17     { name = std::move(newName); }        // compiles, but is
18     ...                                    // bad, bad, bad!
19
20 private:
21     std::string name;
22     std::shared_ptr<SomeDataStructure> p;
23 };
24
25 std::string getWidgetName();           // factory function
26
27 Widget w;
28
29 auto n = getWidgetName();            // n is local variable
30 w.setName(n);                      // moves n into w!
31
32 ...                                // n's value now unknown
```

```
33 Here, the local variable n is passed to w.setName, which the caller can be forgiven
for assuming is a read-only operation on n. But because setName internally uses
std::move to unconditionally cast its reference parameter to an rvalue, n's value
will be moved into w.name, and n will come back from the call to setName with an
unspecified value. That's the kind of behavior that can drive callers to despair—
possibly to violence.
```

1 You might argue that `setName` shouldn't have declared its parameter to be a universal reference. Such references can't be `const` (see Item 24), yet `setName` surely shouldn't modify its parameter. You might point out that if `setName` had simply been overloaded for `const` lvalues and for rvalues, the whole problem could have been avoided. Like this:

```
6 class Widget {  
7 public:  
8     void setName(const std::string& newName)           // set from  
9     { name = newName; }                                // const lvalue  
10    void setName(std::string&& newName)                // set from  
11    { name = std::move(newName); }                      // rvalue  
12    ...  
13};
```

14 That would certainly work in this case, but there are drawbacks. First, it's more  
15 source code to write and maintain (two functions instead of a single template).  
16 Second, it can be less efficient. For example, consider this use of `setName`:

```
17 w.setName("Adela Novak");
```

18 With the version of `setName` taking a universal reference, the string literal "Adela  
19 Novak" would be passed to `setName`, where it would be conveyed to the assignment  
20 operator for the `std::string` inside `w`. `w`'s `name` data member would thus be  
21 assigned directly from the string literal; no temporary `std::string` objects  
22 would arise. With the overloaded versions of `setName`, however, a temporary  
23 `std::string` object would be created for `setName`'s parameter to bind to, and  
24 this temporary `std::string` would then be moved into `w`'s data member. A call to  
25 `setName` would thus entail execution of one `std::string` constructor (to create  
26 the temporary), one `std::string` move assignment operator (to move `newName`  
27 into `w.name`), and one `std::string` destructor (to destroy the temporary). That's  
28 almost certainly a more expensive execution sequence than invoking only the  
29 `std::string` assignment operator taking a `const char*` pointer. The additional  
30 cost is likely to vary from implementation to implementation, and whether that  
31 cost is worth worrying about will vary from application to application and library  
32 to library, but the fact is that replacing a template taking a universal reference with  
33 a pair of functions overloaded on lvalue references and rvalue references is likely

1 to incur a runtime cost in some cases. If we generalize the example such that  
2 `Widget`'s data member may be of an arbitrary type (rather than knowing that it's  
3 `std::string`), the performance gap can widen considerably, because not all types  
4 are as cheap to move as `std::string` (see Item 29).

5 The most serious problem with overloading on lvalues and rvalues, however, isn't  
6 the volume or idiomacity of the source code, nor is it the code's runtime perfor-  
7 mance. It's the poor scalability of the design. `Widget::setName` takes only one  
8 parameter, so only two overloads are necessary in this example, but for functions  
9 taking more parameters, each of which could be an lvalue or an rvalue, the number  
10 of overloads grows geometrically:  $n$  parameters necessitates  $2^n$  overloads. And  
11 that's not the worst of it. Some functions—function templates, actually—take an  
12 *unlimited* number of parameters, each of which could be an lvalue or rvalue. The  
13 poster children for such functions are `std::make_shared`, and, as of C++14,  
14 `std::make_unique` (see Item 21). Check out the declarations of their most com-  
15 monly-used overloads:

```
16 template<class T, class... Args>           // from C++11
17 shared_ptr<T> make_shared(Args&&... args); // Standard
18 template<class T, class... Args>           // from C++14
19 unique_ptr<T> make_unique(Args&&... args); // Standard
```

20 For functions like this, overloading on lvalues and rvalues is not an option: univer-  
21 sal references are the only way to go. And inside such functions, I assure you,  
22 `std::forward` is applied to the universal reference parameters when they're  
23 passed to other functions. Which is exactly what you should do.

24 Well, usually. Eventually. But not necessarily initially. In some cases, you'll want to  
25 use the object bound to an rvalue reference or a universal reference more than  
26 once in a single function, and you'll want to make sure that it's not moved from  
27 until you're otherwise done with it. In that case, you'll want to apply `std::move`  
28 (for rvalue references) or `std::forward` (for universal references) to only the  
29 *final* use of the reference. For example:

```
30 template<typename T>                   // text is
31 void setSignText(T&& text)             // univ. reference
32 {
```

```
1 sign.setText(text); // use text, but  
2 // don't modify it  
3 auto now = // get current time  
4     std::chrono::system_clock::now();  
5 signHistory.add(now,  
6                  std::forward<T>(text)); // conditionally cast  
7 } // text to rvalue
```

8 Here, we want to make sure that `text`'s value doesn't get changed by  
9 `sign.setText`, because we want to use that value when we call `signHistory.`  
10 `add`. Ergo the use of `std::forward` on only the final use of the universal ref-  
11 erence.

12 For `std::move`, the same thinking applies (i.e., apply `std::move` to an rvalue ref-  
13 erence the last time it's used), but it's important to note that in rare cases, you'll  
14 want to call `std::move_if_noexcept` instead of `std::move`. To learn when and  
15 why, consult Item 14.

16 If you're in a function that returns *by value*, and you're returning an object bound  
17 to an rvalue reference or a universal reference, you'll want to apply `std::move` or  
18 `std::forward` when you return the reference. To see why, consider an opera-  
19 `tor+` function to add two matrices together, where the left-hand matrix is known  
20 to be an rvalue (and can hence have its storage reused to hold the sum of the ma-  
21 trices):

```
22 Matrix // by-value return  
23 operator+(Matrix&& lhs, const Matrix& rhs)  
24 {  
25     lhs += rhs;  
26     return std::move(lhs); // move lhs into  
27 } // return value
```

28 By casting `lhs` to an rvalue in the `return` statement (via `std::move`), `lhs` will be  
29 moved into the function's return value location. If the call to `std::move` were  
30 omitted,

```
31 Matrix // as above  
32 operator+(Matrix&& lhs, const Matrix& rhs)  
33 {  
34     lhs += rhs;
```

```
1     return lhs;                                // copy lhs into
2 }  // return value
3 the fact that lhs is an lvalue would force compilers to instead copy it into the re-
4 turn value location. Assuming that the Matrix type supports move construction
5 which is more efficient than copy construction, using std::move in the return
6 statement yields more efficient code.
```

7 If Matrix does not support moving, casting it to an rvalue won't hurt, because the
8 rvalue will simply be copied by Matrix's copy constructor (see Item 23). If Matrix
9 is later revised to support moving, operator+ will automatically benefit the next
10 time it is compiled. That being the case, there's nothing to be lost (and possibly
11 much to be gained) by applying std::move to rvalue references being returned
12 from functions that return by value.

13 The situation is similar for universal references and std::forward. Consider a
14 function template reduceAndCopy that takes a possibly-unreduced Fraction ob-
15 ject, reduces it, and then returns a copy of the reduced value. If the original object
16 is an rvalue, its value should be moved into the return value (thus avoiding the ex-
17 pense of making a copy), but if the original is an lvalue, an actual copy must be cre-
18 ated. Hence:

```
19 template<typename T>
20 Fraction reduceAndCopy(T&& frac)           // by-value return
21 {   // universal reference param
22     frac.reduce();
23     return std::forward<T>(frac);             // move rvalue into return
24 }  // value, copy lvalue
```

26 If the call to std::forward were omitted, frac would be unconditionally copied
27 into reduceAndCopy's return value.

28 Some programmers take the information above and try to extend it to situations
29 where it doesn't apply. "If using std::move on an rvalue reference parameter be-
30 ing copied into a return value turns a copy construction into a move construction,"
31 they reason, "I can perform the same optimization on local variables that I'm re-
32 turning." In other words, they figure that given a function returning a local varia-
33 ble by value, such as this,

```
1 Widget makeWidget()           // "Copying" version of makeWidget
2 {
3     Widget w;                // local variable
4     ...
5     return w;                // "copy" w into return value
6 }
```

7 they can “optimize” it by turning the “copy” into a move:

```
8 Widget makeWidget()           // Moving version of makeWidget
9 {
10    Widget w;
11    ...
12    return std::move(w);      // move w into return value
13 }
```

14 My liberal use of quotation marks should tip you off that this line of reasoning is  
15 flawed. But why is it flawed?

16 It’s flawed, because the Standardization Committee is way ahead of such pro-  
17 grammers when it comes to this kind of optimization. It was recognized long ago  
18 that the “copying” version of `makeWidget` can avoid the need to copy the local var-  
19 iable `w` by constructing it in the memory allotted for the function’s return value.  
20 This is known as the *return value optimization* (RVO), and it’s been expressly  
21 blessed by the C++ Standard for as long as there’s been one.

22 Wording such a blessing is finicky business, because you want to permit such *copy*  
23 *elision* only in places where it won’t affect the observable behavior of the software.  
24 Paraphrasing the legalistic (arguably toxic) prose of the Standard, this particular  
25 blessing says that compilers may elide the copying (or moving) of a local object<sup>†</sup> in  
26 a function that returns by value if (1) the type of the local object is the same as that

---

<sup>†</sup> Eligible local objects include most local variables (such as `w` inside `makeWidget`) as well as temporary objects created as part of a `return` statement. Function parameters don’t qualify. Some people draw a distinction between application of the RVO to named and unnamed (i.e., temporary) local objects, limiting the term RVO to unnamed objects and calling its application to named objects the *named return value optimization* (NRVO).

1 returned by the function and (2) the local object is what's being returned. With  
2 that in mind, look again at the "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 for this  
10 code, every decent C++ compiler will employ the RVO to avoid copying `w`. That  
11 means that the "copying" version of `makeWidget` doesn't, in fact, copy anything.

12 The moving version of `makeWidget` does just what its name says it does (assuming  
13 `Widget` offers a move constructor): it moves the contents of `w` into `makeWidget`'s  
14 return value location. But why don't compilers use the RVO to eliminate the move,  
15 again constructing `w` in the memory allotted for the function's return value? The  
16 answer is simple: they can't. Condition (2) stipulates that the RVO may be per-  
17 formed only if what's being returned is a local object, but that's not what the mov-  
18 ing version of `makeWidget` is doing. Look again at its `return` statement:

```
19 return std::move(w);
```

20 What's being returned here isn't the local object `w`, it's *a reference to w*—the result  
21 of `std::move(w)`. Returning a reference to a local object doesn't satisfy the condi-  
22 tions required for the RVO, so compilers must move `w` into the function's return  
23 value location. Developers trying to help their compilers optimize by applying  
24 `std::move` to a local variable that's being returned are actually limiting the opti-  
25 mization options available to their compilers!

26 But the RVO is an optimization. Compilers aren't *required* to elide copy and move  
27 operations, even when they're permitted to. Maybe you're paranoid, and you wor-  
28 ry that your compilers will punish you with copy operations, just because they can.  
29 Or perhaps you're insightful enough to recognize that there are cases where the  
30 RVO is difficult for compilers to implement, e.g., when different control paths in a  
31 function return different local variables. (Compilers would have to generate code  
32 to construct the appropriate local variable in the memory allotted for the func-

1 tion's return value, but how could compilers determine which local variable would  
2 be appropriate?) If so, you might be willing to pay the price of a move as insurance  
3 against the cost of a copy. That is, you might still think it's reasonable to apply  
4 `std::move` to a local object you're returning, simply because you'd rest easy  
5 knowing you'd never pay for a copy.

6 In that case, applying `std::move` to a local object would *still* be a bad idea. The  
7 part of the Standard blessing the RVO goes on to say that if the conditions for the  
8 RVO are met, but compilers choose not to perform copy elision, the object being  
9 returned *must be treated as an rvalue*. In effect, the Standard requires that when  
10 the RVO is permitted, either copy elision takes place or `std::move` is implicitly  
11 applied to local objects being returned. So in the "copying" version of `makeWidget`,

```
12 Widget makeWidget()          // as before
13 {
14     Widget w;
15     ...
16     return w;
17 }
```

18 compilers must either elide the copying of `w` or they must treat the function as if it  
19 were written like this:

```
20 Widget makeWidget()
21 {
22     Widget w;
23     ...
24     return std::move(w);      // treat w as rvalue, because
25 }                           // no copy elision was performed
```

26 The situation is similar for by-value function parameters. They're not eligible for  
27 copy elision with respect to their function's return value, but compilers must treat  
28 them as rvalues if they're returned. As a result, if your source code looks like this,

```
29 Widget makeWidget(Widget w)      // by-value parameter of same
30 {                                // type as function's return
31     ...
32     return w;
33 }
```

34 compilers must treat it as if it had been written this way:

```
1 Widget makeWidget(Widget w)
2 {
3     ...
4     return std::move(w);           // treat w as rvalue
5 }
```

6 This means that if you use `std::move` on a local object being returned from a  
7 function that's returning by value, you can't help your compilers (they have to  
8 treat the local object as an rvalue if they don't perform copy elision), but you can  
9 certainly hinder them (by precluding the RVO). There are situations where apply-  
10 ing `std::move` to a local variable can be a reasonable thing to do (i.e., when you're  
11 passing it to a function and you know you won't be using the variable any longer),  
12 but as part of a `return` statement that would otherwise qualify for the RVO or that  
13 returns a by-value parameter is not one of them.

#### 14 **Things to remember**

- 15 ♦ Apply `std::move` to rvalue references and `std::forward` to universal refer-  
16 ences the last time each is used.
- 17 ♦ Do the same thing for rvalue references and universal references being re-  
18 turned from functions that return by value.
- 19 ♦ Never apply `std::move` or `std::forward` to local objects if they would oth-  
20 erwise be eligible for the return value optimization.

#### 21 **Item 26: Avoid overloading on universal references.**

22 Suppose you need to write a function that takes a name as a parameter, logs the  
23 current date and time, then adds the name to a global data structure. That is, you  
24 need to write a function that looks something like this:

```
25 std::set<std::string> names;           // global data structure
26 void logAndAdd(const std::string& name)
27 {
28     auto now =                         // get current time
29         std::chrono::system_clock::now();
30     log(now, "logAndAdd");            // make log entry
```

```
1     names.emplace(name);           // add name to global
2 }                                // structure; Item 42
3                                     // has info on emplace
```

4 This isn't unreasonable code, but it's not as efficient as it could be. Consider three  
5 potential calls:

```
6 std::string petName("Darla");
7 logAndAdd(petName);               // pass lvalue std::string
8 logAndAdd(std::string("Persephone")); // pass rvalue std::string
9 logAndAdd("Patty Dog");          // pass string literal
```

10 In the first call, `logAndAdd`'s parameter `name` is bound to the variable `petName`.  
11 Within `logAndAdd`, `name` is ultimately passed to `names.emplace`. Because `name` is  
12 an lvalue, it is copied into `names`. There's no way to avoid that copy, because an  
13 lvalue (`petName`) was passed into `logAndAdd` in the first place.

14 In the second call, the parameter `name` is bound to an rvalue (the temporary  
15 `std::string` explicitly created from "Persephone"). `name` itself is an lvalue, so  
16 it's copied into `names`, but we recognize that, in principle, its value could be moved  
17 into `names`. In this call, we pay for a copy, but we should be able to get by with only  
18 a move.

19 In the third call, the parameter `name` is again bound to an rvalue, but this time it's  
20 to a temporary `std::string` that's implicitly created from "Patty Dog". As in the  
21 second call, `name` is copied into `names`, but in this case, the argument originally  
22 passed to `logAndAdd` was a string literal. Had that string literal been passed di-  
23 rectly to the call to `emplace`, there would have been no need to create a temporary  
24 `std::string` at all. Instead, `emplace` would have used the string literal to create  
25 the `std::string` object directly inside the `std::set`. In this third call, then,  
26 we're paying to copy a `std::string`, yet there's really no reason to pay even for a  
27 move, much less a copy.

28 We can eliminate the inefficiencies in the second and third calls by rewriting `lo-`  
29 `gAndAdd` to take a universal reference (see Item 24) and, in accord with Item 25,  
30 `std::forwarding` this reference to `emplace`. The results speak for themselves:

```

1 template<typename T>
2 void logAndAdd(T&& name)
3 {
4     auto now = std::chrono::system_clock::now();
5     log(now, "logAndAdd");
6     names.emplace(std::forward<T>(name));
7 }

8 std::string petName("Darla");           // as before
9 logAndAdd(name);                     // as before, copy
10                         // lvalue into set
11 logAndAdd(std::string("Persephone")); // move rvalue instead
12                         // of copying it
13 logAndAdd("Patty Dog");             // create std::string
14                         // in set instead of
15                         // copying a temporary
16                         // std::string

```

17 Hurray, optimal efficiency!

18 Were this the end of the story, we could stop here and go home happy, but I ha-  
 19 ven't told you that clients don't always have direct access to the names that lo-  
 20 gAndAdd requires. Some clients have only an index that logAndAdd uses to look  
 21 up the corresponding name in a table. To support such clients, logAndAdd is over-  
 22 loaded:

```

23 std::string nameFromIdx(int idx);           // return name
24                         // corresponding to idx

25 void logAndAdd(int idx)
26 {
27     auto now = std::chrono::system_clock::now();
28     log(now, "logAndAdd");
29     names.emplace(nameFromIdx(idx));
30 }

```

31 Resolution of calls to the two overloads works as expected:

```

32 std::string petName("Darla");           // as before
33 logAndAdd(petName);                   // as before, these
34 logAndAdd(std::string("Persephone")); // calls all invoke
35 logAndAdd("Patty Dog");             // the T&& overload
36 logAndAdd(22);                      // calls int overload

```

1 Actually, resolution works as expected only if you don't expect too much. Suppose  
2 a client has a `short` holding an index and passes that to `logAndAdd`:

```
3 short nameIdx;  
4 ... // give nameIdx a value  
5 logAndAdd(nameIdx); // error!
```

6 The comment on the last line isn't terribly illuminating, so let me explain what  
7 happens here.

8 There are two `logAndAdd` overloads. The one taking a universal reference can de-  
9 duce `T` to be `short`, thus yielding an exact match. The overload with an `int` pa-  
10 rameter can match the `short` argument only with a promotion. Per the normal  
11 overload resolution rules, an exact match beats a match with a promotion, so the  
12 universal reference overload is invoked.

13 Within that overload, the parameter `name` is bound to the `short` that's passed in.  
14 `name` is then `std::forwarded` to the `emplace` member function on `names` (a  
15 `std::set<std::string>`), which, in turn, dutifully forwards it to the  
16 `std::string` constructor. There is no constructor for `std::string` that takes a  
17 `short`, so the `std::string` constructor call inside the call to `set::emplace` in-  
18 side the call to `logAndAdd` fails. All because the universal reference overload was a  
19 better match for a `short` argument than an `int`.

20 Functions taking universal references are the greediest functions in C++. They in-  
21 stantiate to create exact matches for almost any type of argument. (The few kinds  
22 of arguments where this isn't the case are described in Item 30.) This is why com-  
23 bining overloading and universal references is almost always a bad idea: the uni-  
24 versal reference overload vacuums up far more argument types than the developer  
25 doing the overloading generally expects.

26 An easy way to topple into this pit is to write a perfect forwarding constructor. A  
27 small modification to the `logAndAdd` example demonstrates the problem. Instead  
28 of writing a function that can take either a `std::string` or an index that can be  
29 used to look up a `std::string`, imagine a class `Person` to which we can pass ei-  
30 ther a `std::string` or an index:

```
1 class Person {
2 public:
3     template<typename T>
4     explicit Person(T&& n)           // perfect forwarding ctor;
5     : name(std::forward<T>(n)) {}    // initializes data member
6
7     explicit Person(int idx)          // int ctor
8     : name(nameFromIdx(idx)) {}
9     ...
10
11 private:
12     std::string name;
13 };

```

12 As was the case with `logAndAdd`, passing an integral type other than `int` (e.g.,  
13 `std::size_t`, `short`, `long`, etc.) will call the universal reference constructor  
14 overload instead of the `int` overload, and that will lead to compilation failures.  
15 The problem is much worse in this case, however, because there's more overloading  
16 present in `Person` that meets the eye. Item 17 explains that, under the appropriate  
17 conditions, C++ will generate both copy and move constructors, and this is  
18 true even if the class contains a templated constructor that could be instantiated  
19 to produce the signature of the copy or move constructor. If the copy and move  
20 constructors for `Person` are thus generated, `Person` will effectively look like this:

```
21 class Person {
22 public:
23     template<typename T>           // perfect forwarding ctor
24     explicit Person(T&& n);
25
26     explicit Person(int idx);      // int ctor
27
28     Person(const Person& rhs);   // copy ctor (compiler-generated)
29     Person(Person&& rhs);       // move ctor (compiler-generated)
30     ...
31 };

```

30 This leads to behavior that's intuitive only if you've spent so much time around  
31 compilers and compiler-writers, you've forgotten what it's like to be human:

```
32 Person p("Nancy");
33 auto clone(p);                  // create new Person from p;
34   // this won't compile!
```

1 Here we're trying to create a `Person` from another `Person`, which seems like  
2 about as obvious a case for copy construction as one can get. (`p`'s an lvalue, so we  
3 can banish any thoughts we might have about the "copying" being accomplished  
4 through a move operation.) But this code won't call the copy constructor. It will  
5 call the perfect-forwarding constructor. That function will then try to initialize  
6 `Person`'s `std::string` data member with a `Person` object (`p`). `std::string`  
7 having no constructor taking a `Person`, your compilers will throw up their hands  
8 in exasperation, possibly punishing you with long and incomprehensible error  
9 messages as an expression of their displeasure.

10 "Why," you might wonder, "does the perfect-forwarding constructor get called in-  
11 stead of the copy constructor? We're initializing a `Person` with another `Person`!"  
12 Indeed we are, but compilers are sworn to uphold the rules of C++, and the rules of  
13 relevance here are the ones governing the resolution of calls to overloaded func-  
14 tions.

15 Compilers reason as follows. `clone` is being initialized with a `non-const` lvalue  
16 (`p`), and that means that the templated constructor can be instantiated to take a  
17 `non-const` lvalue of type `Person`. After such instantiation, the `Person` class looks  
18 like this:

```
19 class Person {  
20 public:  
21     explicit Person(Person& n); // instantiated from perfect-  
22                             // forwarding template  
23     explicit Person(int idx); // as before  
24     Person(const Person& rhs); // copy ctor (compiler-generated)  
25     ...  
26 };
```

27 In the statement,

```
28 auto clone(p);  
29 p could be passed to either the copy constructor or the instantiated template. Call-  
30 ing the copy constructor would require adding const to p to match the copy con-  
31 structor's parameter's type, but calling the instantiated template requires no such
```

1 addition. The overload generated from the template is thus a better match, so  
2 compilers do what they're designed to do: generate a call to the better-matching  
3 function. "Copying" non-`const` lvalues of type `Person` is thus handled by the per-  
4 fect-forwarding constructor, not the copy constructor.

5 If we change the example slightly so that the object to be copied is `const`, we hear  
6 an entirely different tune:

```
7 const Person cp("Nancy");           // object is now const
8 auto clone(cp);                   // calls copy constructor!
```

9 Because the object to be copied is now `const`, it's an exact match for the parame-  
10 ter taken by the copy constructor. The templated constructor can be instantiated  
11 to have the same signature,

```
12 class Person {
13 public:
14     explicit Person(const Person& n);      // instantiated from
15   // template
16     Person(const Person& rhs);             // copy ctor
17   // (compiler-generated)
18     ...
19 };
```

20 but this doesn't matter, because one of the overload-resolution rules in C++ is that  
21 in situations where a template instantiation and a non-template function (i.e., a  
22 "normal" function) are equally good matches for a function call, the normal func-  
23 tion is preferred. The copy constructor (a normal function) thereby trumps an in-  
24 stantiated template with the same signature.

25 (If you're wondering why compilers generate a copy constructor when they could  
26 instantiate a templated constructor to get the signature that the copy constructor  
27 would have, review Item 17.)

28 The interaction among perfect-forwarding constructors and compiler-generated  
29 copy and move operations develops even more wrinkles when inheritance enters  
30 the picture. In particular, the conventional implementations of derived class copy  
31 and move operations behave quite surprisingly. Here, take a look:

```
1 class SpecialPerson: public Person {  
2 public:  
3     SpecialPerson(const SpecialPerson& rhs) // copy ctor; calls  
4         : Person(rhs) // base class  
5     { ... } // forwarding ctor!  
6     SpecialPerson(SpecialPerson&& rhs) // move ctor; calls  
7         : Person(std::move(rhs)) // base class  
8     { ... } // forwarding ctor!  
9 };
```

10 As the comments indicate, the derived class copy and move constructors don't call  
11 their base class's copy and move constructors, they call the base class's perfect-  
12 forwarding constructor! To understand why, note that the derived class functions  
13 are using arguments of type `SpecialPerson` to pass to their base class, then work  
14 through the template instantiation and overload-resolution consequences for the  
15 constructors in class `Person`. Ultimately, the code won't compile, because there's  
16 no `std::string` constructor taking a `SpecialPerson`.

17 I hope that by now I've convinced you that overloading on universal reference pa-  
18 rameters is something you should avoid if at all possible. But if overloading on  
19 universal references is a bad idea, what do you do if you need a function that for-  
20 wards most argument types, yet needs to treat some argument types in a special  
21 fashion? That egg can be unscrambled in a number of ways. So many, in fact, that  
22 I've devoted an entire Item to them. It's Item 27. The next Item. Keep reading,  
23 you'll bump right into it.

## 24 Things to Remember

- 25 • Overloading on universal references almost always leads to the universal ref-  
26 erence overload being called more frequently than expected.
- 27 • Perfect-forwarding constructors are especially problematic, because they're  
28 typically better matches than copy constructors for non-const lvalues, and they  
29 can hijack derived class calls to base class copy and move constructors.

1   **Item 27: Familiarize yourself with alternatives to overload-**  
2   **ing on universal references.**

3   Item 26 explains that overloading on universal references can lead to a variety of  
4   problems, both for freestanding and for member functions (especially construc-  
5   tors). Yet it also gives examples where such overloading could be useful. If only it  
6   would behave the way we'd like! This Item explores ways to achieve the desired  
7   behavior, either through designs that avoid overloading on universal references or  
8   by employing them in ways that constrain the types of arguments they can match.

9   The discussion that follows builds on the examples introduced in Item 26. If you  
10   haven't read that Item recently, you'll want to review it before continuing.

11   **Abandon overloading**

12   The first example in Item 26, `logAndAdd`, is representative of the many functions  
13   that can avoid the drawbacks of overloading on universal references by simply us-  
14   ing different names for the would-be overloads. The two `logAndAdd` overloads, for  
15   example, could be broken into `logAndAddName` and `logAndAddNameIdx`. Alas, this  
16   approach won't work for the second example we considered, the `Person` construc-  
17   tor, because constructor names are fixed by the language. Besides, who wants to  
18   give up overloading?

19   **Pass by const T&**

20   An alternative is to revert to C++98 and replace pass-by-universal-reference with  
21   pass-by-(lvalue)-reference-to-const. In fact, that's the first approach Item 26 con-  
22   siders (on page 194). The drawback is that the design isn't as efficient as we'd pre-  
23   fer. Knowing what we now know about the interaction of universal references and  
24   overloading, giving up some efficiency to keep things simple might be a more at-  
25   tractive trade-off than it initially appeared.

26   **Pass by value**

27   An approach that often allows you to dial up performance without any increase in  
28   complexity is to replace pass-by-reference parameters with, counterintuitively,  
29   pass-by-value. The design adheres to the advice in Item 41 to consider passing ob-

1 jects by value when you know you'll copy them, so I'll defer to that Item for a de-  
2 tailed discussion of how things work and how efficient they are. Here, I'll just show  
3 how the technique could be used in the `Person` example:

```
4 class Person {  
5 public:  
6     explicit Person(std::string n) // replaces T&& ctor; see  
7     : name(std::move(n)) {}      // Item 41 for use of std::move  
8     explicit Person(int idx)      // as before  
9     : name(nameFromIdx(idx)) {}  
10    ...  
11 private:  
12     std::string name;  
13 };
```

14 Because there's no `std::string` constructor taking only an integer, all `int` and  
15 `int`-like arguments to a `Person` constructor (e.g., `std::size_t`, `short`, `long`) get  
16 funneled to the `int` overload. Similarly, all arguments of type `std::string` (and  
17 things from which `std::strings` can be created, e.g., literals such as "Ruth") get  
18 passed to the constructor taking a `std::string`. There are thus no surprises for  
19 callers. You could argue, I suppose, that some people might be surprised that using  
20 `0` or `NULL` to indicate a null pointer would invoke the `int` overload, but such peo-  
21 ple should be referred to Item 8 and required to read it repeatedly until the  
22 thought of using `0` or `NULL` as a null pointer makes them recoil.

### 23 Use Tag dispatch

24 Neither pass by lvalue-reference-to-`const` nor pass by value offer support for per-  
25 fect forwarding. If your motivation for the use of a universal reference is perfect  
26 forwarding, you have to use a universal reference; there's no other choice. Yet we  
27 don't want to abandon overloading. So if we don't give up overloading and we  
28 don't give up universal references, how can we avoid overloading on universal ref-  
29 erences?

30 It's actually not that hard. Calls to overloaded functions are resolved by looking at  
31 all the parameters of all the overloads as well as all the arguments at the call site,  
32 then choosing the function with the best overall match—taking into account all  
33 parameter/argument combinations. A universal reference parameter generally

1 provides an exact match for whatever's passed in, but if the universal reference is  
2 part of a parameter list containing other parameters that are *not* universal refer-  
3 ences, sufficiently poor matches on the non-universal reference parameters can  
4 knock an overload with a universal reference out of the running. That's the basis  
5 behind the *tag dispatch* approach, and an example will make the foregoing descrip-  
6 tion easier to understand.

7 We'll apply tag dispatch to the `logAndAdd` example from page 196. Here's that  
8 code, to save you the trouble of looking it up:

```
9 std::set<std::string> names;           // global data structure
10 template<typename T>                  // make log entry and add
11 void logAndAdd(T&& name)              // name to data structure
12 {
13     auto now = std::chrono::system_clock::now();
14     log(now, "logAndAdd");
15     names.emplace(std::forward<T>(name));
16 }
```

17 By itself, this function works fine, but were we to introduce the overload taking an  
18 `int` that's used to look up objects by index, we'd be back in the troubled land of  
19 Item 26. The goal of this Item is to avoid that. Rather than adding the overload,  
20 we'll reimplement `logAndAdd` to delegate to two other functions, one for integral  
21 values and one for everything else. `logAndAdd` itself will accept all argument  
22 types, both integral and non-integral.

23 The two functions doing the real work will be named `logAndAddImpl`, i.e., we'll  
24 use overloading. One of the functions will take a universal reference. So we'll have  
25 both overloading and universal references. But each function will also take a sec-  
26 ond parameter, one that indicates whether the argument being passed is integral.  
27 This second parameter is what will prevent us from tumbling into the morass de-  
28 scribed in Item 26, because we'll arrange it so that the second parameter will be  
29 the factor that determines which overload is selected.

30 Yes, I know, "Blah, blah, blah. Stop talking and show me the code!" No problem.  
31 Here's an almost-correct version of the updated `logAndAdd`:

```
32 template<typename T>
33 void logAndAdd(T&& name)
```

```
1 {  
2     logAndAddImpl(std::forward<T>(name),  
3                     std::is_integral<T>());      // not quite correct  
4 }
```

5 This function forwards its parameter to `logAndAddImpl`, but it also passes an ar-  
6 gument indicating whether the type it received (`T`) is integral. At least, that's what  
7 it's supposed to do. For integral arguments that are rvalues, it's also what it does.  
8 But, as Item 28 explains, if an lvalue argument is passed to the universal reference  
9 `name`, the type deduced for `T` will be an lvalue reference. So if an lvalue of type `int`  
10 is passed to `logAndAdd`, `T` will be deduced to be `int&`. That's not an integral type,  
11 because references aren't integral types. That means that `std::is_integral<T>`  
12 will be false for any lvalue argument, even if the argument really does represent an  
13 integral value.

14 Recognizing the problem is tantamount to identifying the solution, because the  
15 ever-handy Standard C++ Library has a type trait, `std::remove_reference`, that  
16 does both what its name suggests and what we need: remove any reference quali-  
17 fiers from a type. The proper way to write `logAndAdd` is therefore:

```
18 template<typename T>  
19 void logAndAdd(T&& name)  
20 {  
21     logAndAddImpl(  
22         std::forward<T>(name),  
23         std::is_integral<typename std::remove_reference<T>::type>()  
24     );  
25 }
```

26 This does the trick. (In C++14, you can save a few keystrokes by using  
27 `std::remove_reference_t<T>` in place of the highlighted text. For details, see  
28 Item 9.)

29 With that taken care of, we can shift our attention to the function being called, `lo-`  
30 `gAndAddImpl`. There are two overloads, and the first is applicable only to non-  
31 integral types (i.e., to types where `std::is_integral<typename`  
32 `std::remove_reference<T>::type>` is false):

```
33 template<typename T>                      // non-integral  
34 void logAndAddImpl(T&& name, std::false_type) // argument:  
35 {  // add it to
```

```
1 auto now = std::chrono::system_clock::now(); // global data
2 log(now, "logAndAdd"); // structure
3 names.emplace_back(std::forward<T>(name));
4 }
```

5 This is straightforward code, once you understand the mechanics behind the high-  
6 lighted parameter. Conceptually, `logAndAdd` passes a boolean to `logAndAddImpl`  
7 indicating whether an integral type was passed to `logAndAdd`, but `true` and  
8 `false` are *runtime* values, and we need to use overload resolution—a *compile-time*  
9 phenomenon—to choose the correct `logAndAddImpl` overload. That means we  
10 need a *type* that corresponds to `true` and a different type that corresponds to  
11 `false`. This need is common enough that the Standard Library provides what is  
12 required under the names `std::true_type` and `std::false_type`. The argu-  
13 ment passed to `logAndAddImpl` by `logAndAdd` is an object of a type that inherits  
14 from `std::true_type` if `T` is integral and from `std::false_type` if `T` is not inte-  
15 gral. The net result is that this `logAndAddImpl` overload is a viable candidate for  
16 the call in `logAndAdd` only if `T` is not an integral type.

17 The second overload covers the opposite case: when `T` is an integral type. In that  
18 event, `logAndAddImpl` simply finds the name corresponding to the passed-in in-  
19 dex and passes that name back to `logAndAdd`:

```
20 std::string nameFromIdx(int idx); // as in Item 26
21 void logAndAddImpl(int idx, std::true_type) // integral
22 { // argument: look
23     logAndAdd(anyFromIdx(idx)); // up name and
24 } // call logAndAdd
25 // with it
```

26 By having `logAndAddImpl` for an index look up the corresponding name and pass  
27 it to `logAndAdd` (from where it will be `std::forwarded` to the other `logAn-`  
28 `dAddImpl` overload), we avoid the need to put the logging code in both `logAn-`  
29 `dAddImpl` overloads.

30 In this design, the types `std::true_type` and `std::false_type` are “tags”  
31 whose only purpose is to force overload resolution to go the way we want. Notice  
32 that we don’t even name those parameters. They serve no purpose at run time, and  
33 in fact we hope that compilers will recognize that tag parameters are unused and

1 will optimize them out of the program’s execution image. (Some compilers do, at  
2 least some of the time.) The call to the overloaded implementation functions inside  
3 `logAndAdd` “dispatches” the work to the correct overload by causing the proper  
4 tag object to be created. Hence the name for this design: *tag dispatch*. It’s a stand-  
5 ard building block of template metaprogramming, and the more you look at code  
6 inside contemporary C++ libraries, the more often you’ll encounter it.

7 For our purposes, what’s important about tag dispatch is less how it works and  
8 more how it permits us to combine universal references and overloading without  
9 the problems described in Item 26. The dispatching function—`logAndAdd`—takes  
10 an unconstrained universal reference parameter, but this function is not overload-  
11 ed. The implementation functions—`logAndAddImpl`—are overloaded, and one  
12 takes a universal reference parameter, but resolution of calls to these functions  
13 depends not just on the universal reference parameter, but also on the tag parame-  
14 ter, and the tag values are designed so that no more than one overload will ever be  
15 a viable match. As a result, it’s the tag that determines which overload gets called.  
16 The fact that the universal reference parameter will always generate an exact  
17 match for its argument is immaterial.

## 18 **Constraining templates that take universal references**

19 A keystone of tag dispatch is the existence of a single (unoverloaded) function as  
20 the client API. This single function dispatches the work to be done to the imple-  
21 mentation functions. Creating an unoverloaded dispatch function is usually easy,  
22 but the second problem case Item 26 considers, that of a perfect-forwarding con-  
23 structor for the `Person` class (see page 198), is an exception. Compilers may gen-  
24 erate copy and move constructors themselves, so even if you write only one con-  
25 structor and use tag dispatch within it, some constructor calls may be handled by  
26 compiler-generated functions that bypass the tag dispatch system.

27 In truth, the real problem is not that the compiler-generated functions sometimes  
28 bypass the tag dispatch design, it’s that they don’t *always* pass it by. You virtually  
29 always want the copy constructor for a class to handle requests to copy lvalues of  
30 that type, but, as Item 26 demonstrates, providing a constructor taking a universal  
31 reference causes the universal reference constructor (rather than the copy con-

1 structor) to be called when copying non-`const` lvalues. That Item also explains  
2 that when a base class declares a perfect-forwarding constructor, that constructor  
3 will typically be called when derived classes implement their copy and move con-  
4 structors in the conventional fashion, even though the correct behavior is for the  
5 base class's copy and move constructors to be invoked.

6 For situations like these, where an overloaded function taking a universal refer-  
7 ence is greedier than you want, yet not greedy enough to act as a single dispatch  
8 function, tag dispatch is not the droid you're looking for. You need a different tech-  
9 nique, one that lets you ratchet down the conditions under which the function tem-  
10 plate that the universal reference is part of is permitted to be employed. What you  
11 need, my friend, is `std::enable_if`.

12 `std::enable_if` gives you a way to force compilers to behave as if a particular  
13 template didn't exist. Such templates are said to be *disabled*. By default, all tem-  
14 plates are *enabled*, but a template using `std::enable_if` is enabled only if the  
15 condition specified by `std::enable_if` is satisfied. In our case, we'd like to ena-  
16 ble the `Person` perfect-forwarding constructor only if the type being passed isn't  
17 `Person`. If the type being passed is `Person`, we want to disable the perfect-  
18 forwarding constructor (i.e., cause compilers to ignore it), because that will cause  
19 the class's copy or move constructor to handle the call, which is what we want  
20 when a `Person` object is initialized with another `Person`.

21 The way to express that idea isn't particularly difficult, but the syntax is off-  
22 putting, especially if you've never seen it before, so I'll ease you into it. There's  
23 some boilerplate that goes around the condition part of `std::enable_if`, so we'll  
24 start with that. Here's the declaration for the perfect-forwarding constructor in  
25 `Person`, showing only as much of the `std::enable_if` as is required simply to  
26 use it. I'm showing only the declaration for this constructor, because the use of  
27 `std::enable_if` has no effect on the function's implementation. (The implemen-  
28 tation remains the same as in Item 26.)

```
29 class Person {  
30 public:  
31     template<typename T,
```

```
1     typename = typename std::enable_if<condition>::type>
2     explicit Person(T&& n);
3
4     ...
5
6     };
```

5 To understand exactly what's going on in the highlighted text, I must regretfully  
6 suggest that you consult other sources, because the details take a while to explain,  
7 and there's just not enough space for it in this book. (During your research, look  
8 into "SFINAE" as well as `std::enable_if`, because SFINAE is the technology that  
9 makes `std::enable_if` work.) Here, I want to focus on expression of the condi-  
10 tion that will control whether this constructor is enabled.

11 The condition we want to specify is that `T` isn't `Person`, i.e., that the templated  
12 constructor should be enabled only if `T` is a type other than `Person`. Thanks to a  
13 type trait (see Item 9) that determines whether two types are the same  
14 (`std::is_same`), it would seem that the condition we want is  
15 `!std::is_same<Person, T>::value`. (Notice the "!" at the beginning of the  
16 expression. We want for `Person` and `T` to *not* be the same.) This is close to what  
17 we need, but it's not quite correct, because, as Item 28 explains, the type deduced  
18 for a universal reference initialized with an lvalue is always an lvalue reference.  
19 That means that for code like this,

```
20 Person p("Nancy");
21 Person clone = p;           // initialize from lvalue
```

22 the type `T` in the universal constructor will be deduced to be `Person&`. The types  
23 `Person` and `Person&` are not the same, and the result of `std::is_same` will re-  
24 flect that: `std::is_same<Person, Person&>::value` is false.

25 If we think more precisely about what we mean when we say that the templated  
26 constructor in `Person` should be enabled only if `T` isn't `Person`, we'll realize that  
27 when we're looking at `T`, we want to ignore

- 28 • **Whether it's a reference.** For the purpose of determining whether the univer-  
29 sal reference constructor should be enabled, the types `Person`, `Person&`, and  
30 `Person&&` are all the same as `Person`.

1 • **Whether it's `const` or `volatile`.** As far as we're concerned, a `const Person`  
2 and a `volatile Person` and a `const volatile Person` are all the same as a  
3 `Person`.

4 This means we need a way to strip any references, `consts`, and `volatiles` from `T`  
5 before checking to see if that type is the same as `Person`. Once again, the Standard  
6 Library gives us what we need in the form of a type trait. That trait is `std::decay`.  
7 `std::decay<T>::type` is the same as `T`, except that references and *cv-qualifiers*  
8 (i.e., `const` or `volatile` qualifiers) are removed. (I'm fudging the truth here, be-  
9 cause `std::decay`, as its name suggests, also turns array and function types into  
10 pointers (see Item 1), but for purposes of this discussion, `std::decay` behaves as  
11 I've described.) The condition we want to control whether our constructor is ena-  
12 bled, then, is

13     `!std::is_same<Person, typename std::decay<T>::type>::value`  
14 i.e., `Person` is not the same type as `T`, ignoring any references or cv-qualifiers. (As  
15 Item 9 explains, the “`typename`” in front of `std::decay` is required, because the  
16 type `std::decay<T>::type` depends on the template parameter `T`.)

17 Inserting this condition into the `std::enable_if` boilerplate above, plus format-  
18 ting the result to make it easier to see how the pieces fit together, yields this decla-  
19 ration for `Person`'s perfect-forwarding constructor:

```
20 class Person {  
21 public:  
22     template<  
23         typename T,  
24         typename = typename std::enable_if<  
25             !std::is_same<Person,  
26                         typename std::decay<T>::type  
27                     >::value  
28                 >  
29     explicit Person(T&& n);  
30  
31     ...  
32 };
```

33 If you've never seen anything like this before, count your blessings. There's a rea-  
34 son I saved this design for last. When you can use one of the other mechanisms to

1 avoid mixing universal references and overloading (and you almost always can),  
2 you should. Still, once you get used to the functional syntax and the proliferation of  
3 angle brackets, it's really not that bad. Furthermore, this gives you the behavior  
4 you've been striving for. Given the declaration above, constructing a `Person` from  
5 another `Person`—lvalue or rvalue, `const` or non-`const`, `volatile` or non-  
6 `volatile`—will never invoke the constructor taking a universal reference.

7 Success, right? We're done!

8 Um, no. Belay that celebration. There's still one loose end from Item 26 that con-  
9 tinues to flap about. We need to tie it down.

10 Suppose a class derived from `Person` implements the copy and move operations in  
11 the conventional manner:

```
12 class SpecialPerson: public Person {  
13 public:  
14     SpecialPerson(const SpecialPerson& rhs) // copy ctor; calls  
15         : Person(rhs) // base class  
16         { ... } // forwarding ctor!  
17     SpecialPerson(SpecialPerson&& rhs) // move ctor; calls  
18         : Person(std::move(rhs)) // base class  
19         { ... } // forwarding ctor!  
20     ...  
21 };
```

22 This is the same code I showed in Item 26 (on page 201), including the comments,  
23 which, alas, remain accurate. When we copy or move a `SpecialPerson` object, we  
24 expect to copy or move its base class parts using the base class's copy and move  
25 constructors, but in these functions, we're passing `SpecialPerson` objects to the  
26 base class's constructors, and because `SpecialPerson` isn't the same as `Person`  
27 (not even after application of `std::decay`), the universal reference constructor in  
28 the base class is enabled, and it happily instantiates to perform an exact match for  
29 a `SpecialPerson` argument. This exact match is better than the derived-to-base  
30 conversions that would be necessary to bind the `SpecialPerson` objects to the  
31 `Person` parameters in `Person`'s copy and move constructors, so with the code we  
32 have now, copying and moving `SpecialPerson` objects would use the `Person`

1 perfect-forwarding constructor to copy or move their base class parts! It's déjà  
2 Item 26 all over again.

3 The derived class is just following the normal rules for implementing derived class  
4 copy and move constructors, so the fix for this problem is in the base class and, in  
5 particular, in the condition that controls whether `Person`'s universal reference  
6 constructor is enabled. We now realize that we don't want to enable the tem-  
7 platized constructor for any argument type other than `Person`, we want to enable  
8 it for any argument type other than `Person` or a type derived from `Person`. Pesky  
9 inheritance!

10 You should not be surprised to hear that among the standard type traits is one that  
11 determines whether one type is derived from another. It's called  
12 `std::is_base_of`. `std::is_base_of<T1, T2>::value` is true if `T2` is derived  
13 from `T1`. Types are considered to be derived from themselves, so  
14 `std::is_base_of<T, T>::value` is true. This is handy, because we want to re-  
15 vise our condition controlling `Person`'s perfect-forwarding constructor such that  
16 the constructor is enabled only if the type `T`, after stripping it of references and cv-  
17 qualifiers, is neither `Person` nor a class derived from `Person`. Using  
18 `std::is_base_of` instead of `std::is_same` gives us what we need:

```
19 class Person {
20 public:
21     template<
22         typename T,
23         typename = typename std::enable_if<
24             !std::is_base_of<Person,
25                         typename std::decay<T>::type
26                     >::value
27                 >>
28     explicit Person(T&& n);
29
30     ...
31 };
```

32 Now we're finally done. Provided we're writing the code in C++11, that is. If we're  
33 using C++14, this code will still work, but we can employ alias templates for

1 std::enable\_if and std::decay to get rid of the “typename” and “::type”  
2 cruft (see Item 9), thus yielding this somewhat more palatable code:

```
3 class Person {                                     // C++14 version
4 public:
5     template<
6         typename T,
7         typename = std::enable_if_t<           // less code here
8             !std::is_base_of<Person,
9                 std::decay_t<T> // and here
10            >::value
11        >                                // and here
12     explicit Person(T&& n);
13
14 ...
15 };
```

16 Okay, I admit it: I lied. We’re still not done. But we’re close. Tantalizingly close.  
17 Honest.

18 We’ve seen how to use std::enable\_if to selectively disable Person’s universal  
19 reference constructor for argument types we want to have handled by the class’s  
20 copy and move constructors, but we haven’t yet seen how to apply it to distinguish  
21 integral and non-integral arguments. That was, after all, our original goal; the con-  
22 structor ambiguity problem was just something we got dragged into along the way.

23 All we need to do—and I really do mean that this is everything—is (1) add a Per-  
24 son constructor overload to handle integral arguments and (2) further constrain  
25 the templated constructor so that it’s disabled for such arguments. Pour these  
26 ingredients into the pot with everything else we’ve discussed, simmer over a low  
27 flame, and savor the aroma of success:

```
28 class Person {
29 public:
30     template<
31         typename T,
32         typename = std::enable_if_t<
33             !std::is_base_of<Person, std::decay_t<T>>::value
34             &&
35             !std::is_integral<std::remove_reference_t<T>>::value
36         >
37     >
```

```

1   explicit Person(T&& n)           // ctor for std::strings and
2     : name(std::forward<T>(n))      // args convertible to
3   { ... }                           // std::strings
4   explicit Person(int idx)          // ctor for integral args
5     : name(nameFromIdx(idx))
6   { ... }
7   ...
8 private:
9   std::string name;
10 };

```

11 *Voilà!* A thing of beauty! Well, okay, the beauty is perhaps most pronounced for  
 12 those with something of a template metaprogramming fetish, but the fact remains  
 13 that this approach not only gets the job done, it does it with unique aplomb. Be-  
 14 cause it uses perfect forwarding, it offers maximal efficiency, and because it con-  
 15 trols the combination of universal references and overloading rather than forbid-  
 16 ding it, this technique can be applied in circumstances (such as constructors)  
 17 where overloading is unavoidable.

## 18 Trade-offs

19 The first three techniques considered in this Item—abandoning overloading, pass-  
 20 ing by `const T&`, and passing by value—specify a type for each parameter in the  
 21 function(s) to be called. The last two techniques—tag dispatch and constraining  
 22 template eligibility—use perfect forwarding, hence don’t specify types for the pa-  
 23 rameters. This fundamental decision—to specify a type or not—has consequences.

24 As a rule, perfect forwarding is more efficient, because it avoids the creation of  
 25 temporary objects solely for the purpose of conforming to the type of a parameter  
 26 declaration. In the case of the `Person` constructor, perfect forwarding permits a  
 27 string literal such as "Nancy" to be forwarded to the constructor for the  
 28 `std::string` inside `Person`, whereas techniques not using perfect forwarding  
 29 must create a temporary `std::string` object from the string literal to satisfy the  
 30 parameter specification for the `Person` constructor.

1 But perfect forwarding has drawbacks. One is that some kinds of arguments can't  
2 be perfect-forwarded, even though they can be passed to functions taking specific  
3 types. Item 30 explores these perfect forwarding failure cases.

4 A second issue is the comprehensibility of error messages when clients pass invalid  
5 arguments. Suppose, for example, a client creating a `Person` object passes a  
6 string literal made up of `char16_ts` (a type introduced in C++11 to represent 16-  
7 bit characters) instead of `chars` (which is what a `std::string` consists of):

```
8 Person p(u"Konrad Zuse"); // "Konrad Zuse" consists of
9 // characters of type const char16_t
```

10 With the first three approaches examined in this Item, compilers will see that the  
11 available constructors take either `int` or `std::string`, and they'll produce a  
12 more or less straightforward error message explaining that there's no conversion  
13 from `const char16_t[12]` to `int` or `std::string`.

14 With an approach based on perfect forwarding, however, the array of `const`  
15 `char16_ts` gets bound to the constructor's parameter without complaint. From  
16 there it's forwarded to the constructor of `Person`'s `std::string` data member,  
17 and it's only at that point that the mismatch between what the caller passed in (a  
18 `const char16_t` array) and what's required (any type acceptable to the  
19 `std::string` constructor) is discovered. The resulting error message is likely to  
20 be, er, impressive. With one of the compilers I use, it's more than 160 lines long.

21 In this example, the universal reference is forwarded only once (from the `Person`  
22 constructor to the `std::string` constructor), but the more complex the system,  
23 the more likely that a universal reference is forwarded through several layers of  
24 function calls before finally arriving at a site that determines whether the argument  
25 type(s) are acceptable. The more times the universal reference is forwarded,  
26 the more baffling the error message may be when something goes wrong. Many  
27 developers find that this issue alone is grounds to reserve universal reference  
28 parameters for interfaces where performance is a foremost concern.

29 In the case of `Person`, we know that the forwarding function's universal reference  
30 parameter is supposed to be an initializer for a `std::string`, so we can use a  
31 `static_assert` to verify that it can play that role. The `std::is_constructible`

1 type trait (see Item 9) performs a compile-time test to determine whether an ob-  
2 ject of one type can be constructed from an object (or set of objects) of a different  
3 type (or set of types), so the assertion is easy to write:

```
4 class Person {  
5 public:  
6     template<   // as before  
7         typename T,  
8         typename = std::enable_if_t<  
9             !std::is_base_of<Person, std::decay_t<T>>::value  
10            &&  
11            !std::is_integral<std::remove_reference_t<T>>::value  
12        >  
13     explicit Person(T&& n)  
14     : name(std::forward<T>(n))  
15     {  
16         // assert that a std::string can be created from a T object  
17         static_assert(  
18             std::is_constructible<std::string, T>::value,  
19             "Parameter n can't be used to construct a std::string"  
20         );  
21     }  
22     ...   // the usual ctor work goes here  
23 }  
24 ...   // remainder of Person class (as before)  
25 };
```

26 This causes the specified error message to be produced if client code tries to create  
27 a `Person` from a type that can't be used to construct a `std::string`. Unfortunate-  
28 ly, in this example the `static_assert` is in the body of the constructor, but the  
29 forwarding code, being part of the member initialization list, precedes it. With the  
30 compilers I use, the result is that the nice, readable message arising from the  
31 `static_assert` appears only *after* the usual error messages (up to 160-plus lines  
32 of them) have been emitted.

33 **Things to Remember**

- 34 • Alternatives to the combination of universal references and overloading in-  
35 clude the use of distinct function names, passing parameters by `(lvalue)-`  
36 `reference-to-const`, passing parameters by value, and using tag dispatch.

- ◆ Constraining templates via `std::enable_if` permits the use of universal references and overloading together, but it controls the conditions under which compilers may use the universal reference overloads.
  - ◆ Universal reference parameters often have efficiency advantages, but they typically have usability disadvantages.

## 6 Item 28: Understand reference collapsing.

7 Item 23 remarks that when an argument is passed to a template function, the type  
8 deduced for the template parameter encodes whether the argument is an lvalue or  
9 an rvalue. The Item fails to mention that this happens only when the argument is  
10 used to initialize a parameter that's a universal reference, but there's a good rea-  
11 son for the omission: universal references aren't introduced until Item 24. Togeth-  
12 er, these observations about universal references and lvalue/rvalue encoding  
13 mean that for this template,

```
14     template<typename T>
15     void func(T&& param);
```

the deduced template parameter T will encode whether the argument passed to param was an lvalue or an rvalue.

18 The encoding mechanism is simple. When an lvalue is passed as an argument, T is  
19 deduced to be an lvalue reference. When an rvalue is passed, T is deduced to be a  
20 non-reference. (Note the asymmetry: lvalues are encoded as lvalue references, but  
21 rvalues are encoded as *non-references*.) Hence:

28 In both calls to `func`, a `Widget` is passed, yet because one `Widget` is an lvalue and  
29 one is an rvalue, different types are deduced for the template parameter `T`. This, as  
30 we shall soon see, is what determines whether universal references become rvalue

1 references or lvalue references, and it's also the underlying mechanism through  
2 which `std::forward` does its work.

3 Before we can look more closely at `std::forward` and universal references, we  
4 must note that references to references are illegal in C++. Should you try to declare  
5 one, your compilers will reprimand you:

```
6 int x;  
7 ...  
8 auto& & rx = x; // error! can't declare reference to reference
```

9 But consider what happens when an lvalue is passed to a function template taking  
10 a universal reference:

```
11 template<typename T>  
12 void func(T&& param); // as before  
13 func(w); // invoke func with lvalue;  
14 // T deduced as Widget&
```

15 If we take the type deduced for `T` (i.e., `Widget&`) and use it to instantiate the tem-  
16 plate, we get this:

```
17 void func(Widget& && param);
```

18 A reference to a reference! And yet compilers issue no protest. We know from Item  
19 24 that because the universal reference `param` is being initialized with an lvalue,  
20 `param`'s type is supposed to be an lvalue reference, but how does the compiler get  
21 from the result of taking the deduced type for `T` and substituting it into the tem-  
22 plate to the following, which is the ultimate function signature?

```
23 void func(Widget& param);
```

24 The answer is *reference collapsing*. Yes, *you* are forbidden from declaring refer-  
25 ences to references, but *compilers* may produce them in particular contexts, tem-  
26 plate instantiation being among them. When compilers generate references to ref-  
27 erences, reference collapsing dictates what happens next.

28 There are two kinds of references (lvalue and rvalue), so there are four possible  
29 reference-reference combinations (lvalue to lvalue, lvalue to rvalue, rvalue to  
30 lvalue, and rvalue to rvalue). If a reference to a reference arises in a context where

1 this is permitted (e.g., during template instantiation), the references *collapse* to a  
2 single reference according to this rule:

3 If either reference is an lvalue reference, the result is an lvalue reference.  
4 Otherwise (i.e., if both are rvalue references) the result is an rvalue refer-  
5 ence.

6 In our example above, substitution of the deduced type `Widget&` into the template  
7 `func` yields an rvalue reference to an lvalue reference, and the reference-  
8 collapsing rule tells us that the result is an lvalue reference.

9 Reference collapsing is a key part of what makes `std::forward` work. As ex-  
10 plained in Item 25, `std::forward` is applied to universal reference parameters, so  
11 a common use case looks like this:

```
12 template<typename T>
13 void f(T&& fParam)
14 {
15     ...
16     someFunc(std::forward<T>(fParam)); // forward fParam to
17 }
```

18 Because `fParam` is a universal reference, we know that the type parameter `T` will  
19 encode whether the argument passed to `f` (i.e., the expression used to initialize  
20 `fParam`) was an lvalue or an rvalue. `std::forward`'s job is to cast `fParam` (an  
21 lvalue) to an rvalue if and only if `T` encodes that the argument passed to `f` was an  
22 rvalue, i.e., if `T` is a non-reference type.

23 Here's how `std::forward` can be implemented to do that:

```
24 template<typename T> // in
25 T&& forward(typename // namespace
26             remove_reference<T>::type& param) // std
27 {
28     return static_cast<T&&>(param);
29 }
```

30 This isn't quite standards-conformant (I've omitted a few interface details), but the  
31 differences are irrelevant for the purpose of understanding how `std::forward`  
32 behaves.

1 Suppose that the argument passed to `f` is an lvalue of type `Widget`. `T` will be deduced as `Widget&`, and the call to `std::forward` will instantiate as  
2 `std::forward<Widget&>`. Plugging `Widget&` into the `std::forward` implementation yields this:  
3

```
5 Widget& && forward(typename
6           remove_reference<Widget&>::type& param)
7 { return static_cast<Widget& &&>(param); } 
```

8 The type trait `std::remove_reference<Widget&>::type` yields `Widget` (see  
9 Item 9), so `std::forward` becomes:

```
10 Widget& && forward(Widget& param)
11 { return static_cast<Widget& &&>(param); } 
```

12 Reference collapsing is then applied to the return type and the cast, and the result  
13 is the final version of `std::forward` for the call:

```
14 Widget& forward(Widget& param)  // still in
15 { return static_cast<Widget&>(param); }  // namespace std
```

16 As you can see, when an lvalue argument is passed to the function template `f`,  
17 `std::forward` is instantiated to take and return an lvalue reference. The cast inside  
18 `std::forward` does nothing, because `param`'s type is already `Widget&`, so  
19 casting it to `Widget&` has no effect. An lvalue argument passed to `std::forward`  
20 will thus return an lvalue reference. By definition, lvalue references are lvalues, so  
21 passing an lvalue to `std::forward` causes an lvalue to be returned, just like it's  
22 supposed to.

23 Now suppose that the argument passed to `f` is an rvalue of type `Widget`. In this  
24 case, the deduced type for `f`'s type parameter `T` will simply be `Widget`. The call  
25 inside `f` to `std::forward` will thus be to `std::forward<Widget>`. Substituting  
26 `Widget` for `T` in the `std::forward` implementation gives this:

```
27 Widget&& forward(typename
28           remove_reference<Widget>::type& param)
29 { return static_cast<Widget&&>(param); } 
```

30 Applying `std::remove_reference` to the non-reference type `Widget` yields the  
31 same type it started with (`Widget`), so `std::forward` becomes this:

```
1 Widget&& forward(Widget& fwdParam)
2 { return static_cast<Widget&&>(param); }
```

3 There are no references to references here, so there's no reference collapsing, and  
4 this is the final instantiated version of `std::forward` for the call.

5 Rvalue references returned from functions are defined to be rvalues, so in this  
6 case, `std::forward` will turn `f`'s parameter `fParm` (an lvalue) into an rvalue. The  
7 end result is that an rvalue argument passed to `f` will be forwarded to `someFunc`  
8 as an rvalue, which is precisely what is supposed to happen.

9 In C++14, the existence of `std::remove_reference_t` (see Item 9) makes it pos-  
10 sible to implement `std::forward` a bit more concisely:

```
11 template<typename T> // C++14; still in
12 T&& forward(remove_reference_t<T>& param) // namespace std
13 {
14     return static_cast<T&&>(param);
15 }
```

16 Reference collapsing occurs in four contexts. The first and most common is tem-  
17 plate instantiation. The second is type generation for `auto` variables. The details  
18 are essentially the same as for templates, because type deduction for `auto` varia-  
19 bles is essentially the same as type deduction for templates (see Item 2). Consider  
20 again this example from earlier in the Item:

```
21 template<typename T>
22 void func(T&& param);

23 Widget widgetFactory(); // function returning rvalue
24 Widget w; // a variable (an lvalue)
25 func(w); // call func with lvalue; T deduced
26 // to be Widget&
27 func(widgetFactory()); // call func with rvalue; T deduced
28 // to be Widget
```

29 This example can be mimicked in `auto` form. The declaration

```
30 auto&& w1 = w;
```

1 initializes `w1` with an lvalue, thus deducing the type `Widget&` for `auto`. Plugging  
2 `Widget&` in for `auto` in the declaration for `w1` yields this reference-to-reference  
3 code,

4 `Widget& && w1 = w;`

5 which, after reference collapsing, becomes

6 `Widget& w1 = w;`

7 As a result, `w1` is an lvalue reference.

8 On the other hand, this declaration,

9 `auto&& w2 = widgetFactory();`

10 initializes `w2` with an rvalue, causing the non-reference type `Widget` to be deduced  
11 for `auto`. Substituting `Widget` for `auto` gives us this:

12 `Widget&& w2 = widgetFactory();`

13 There are no references to references here, so we're done; `w2` is an rvalue refer-  
14 ence.

15 We're now in a position to truly understand the universal references introduced in  
16 Item 24. A universal reference isn't a new kind of reference, it's actually an rvalue  
17 reference in a context where two conditions are satisfied:

18 • **Type deduction distinguishes lvalues from rvalues.** Lvalues of type `T` are  
19 deduced to have type `T&`, while rvalues of type `T` yield `T` as their deduced type.

20 • **Reference collapsing occurs.**

21 The concept of universal references is useful, because it frees you from having to  
22 recognize the existence of reference collapsing contexts, to mentally deduce differ-  
23 ent types for lvalues and rvalues, and to apply the reference collapsing rule after  
24 mentally substituting the deduced types into the contexts in which they occur.

25 I said there were four such contexts, but we've discussed only two: template in-  
26 stantiation and `auto` type generation. The third is the generation and use of  
27 `typedefs` and alias declarations (see Item 9). If, during creation or evaluation of a

1    **typedef**, references to references arise, reference collapsing intervenes to eliminate them. For example, suppose we have a `Widget` class template with an embedded **typedef** for an rvalue reference type,

```
4  template<typename T>
5  class Widget {
6  public:
7      typedef T&& RvalueRefToT;
8      ...
9  };
```

10 and suppose we instantiate `Widget` with an lvalue reference type:

```
11 Widget<int&> w;
```

12 Substituting `int&` for `T` in the `Widget` template gives us the following **typedef**:

```
13 typedef int& && RvalueRefToT;
```

14 Reference collapsing reduces it to this,

```
15 typedef int& RvalueRefToT;
```

16 which makes clear that the name we chose for the **typedef** is perhaps not as descriptive as we'd hoped: `RvalueRefToT` is a **typedef** for an *lvalue reference* when `Widget` is instantiated with an lvalue reference type.

19 The final context in which reference collapsing takes place is uses of `decltype`. If, 20 during analysis of a type involving `decltype`, a reference to reference arises, reference collapsing will kick in to eliminate it.

## 22 **Things to Remember**

- 23    ♦ Reference collapsing occurs in four contexts: template instantiation, `auto` type 24    generation, creation and use of **typedefs** and alias declarations, and 25    `decltype`.
- 26    ♦ When compilers generate a reference to a reference in a reference collapsing 27    context, the result becomes a single reference. If either of the original references 28    is an lvalue reference, the result is an lvalue reference. Otherwise it's an 29    rvalue reference.

- 1     ♦ Universal references are rvalue references in contexts where type deduction  
2       distinguishes lvalues from rvalues and where reference-collapsing occurs.

3     **Item 29: Assume that move operations are not present, not  
4       cheap, and not used.**

5     Move semantics is arguably *the* premier feature of C++11. “Moving containers is  
6     now as cheap as copying pointers!,” you’re likely to hear, and “Copying temporary  
7     objects is now so efficient, coding to avoid it is tantamount to premature optimiza-  
8     tion!” Such sentiments are easy to understand. Move semantics is truly an im-  
9     portant feature. It doesn’t just allow compilers to replace expensive copy opera-  
10    tions with comparatively cheap moves, it actually *requires* that they do so (when  
11    the proper conditions are fulfilled). Take your C++98 code base, recompile with a  
12    C++11-conformant compiler and Standard Library, and—*shazam!*—your software  
13    runs faster!

14    Move semantics can really pull that off, and that grants the feature an aura worthy  
15    of legend. Legends, however, are generally the result of exaggeration. The purpose  
16    of this Item is to keep your expectations grounded.

17    Let’s begin with the observation that many types fail to support move semantics.  
18    The entire C++98 Standard Library was overhauled for C++11 to add move opera-  
19    tions for types where moving could be implemented faster than copying, and the  
20    implementation of the library components was revised to take advantage of these  
21    operations, but chances are that you’re working with a code base that has not been  
22    completely revised to take advantage of C++11. For types in your applications (or  
23    in the libraries you use) where no modifications for C++11 have been made, the  
24    existence of move support in your compilers is likely to do you little good. True,  
25    C++11 is willing to generate move operations for classes that lack them, but that  
26    happens only for classes declaring no copy operations, move operations, or de-  
27    structors (see Item 17). Data members or base classes of types that have disabled  
28    moving (e.g., by declaring the move operations `=delete`) will also suppress com-  
29    piler-generated move operations. For types without explicit support for moving  
30    and that don’t qualify for compiler-generated move operations, there is no reason  
31    to expect C++11 to deliver any kind of performance improvement over C++98.

1 Even types with explicit move support may not benefit as much as you'd hope. All  
2 containers in the standard C++11 library support moving, for example, but it  
3 would be a mistake to assume that moving all containers is cheap. For some con-  
4 tainers, this is because there's no truly cheap way to move their contents. For oth-  
5 ers, it's because the truly cheap move operations the containers offer come with  
6 caveats the container elements can't satisfy.

7 Consider `std::array`, a new container in C++11. `std::array` is essentially a  
8 built-in array with an STL interface. This is fundamentally different from the other  
9 standard containers, each of which stores its contents on the heap. Objects of such  
10 container types hold (as data members), conceptually, only a pointer to the heap  
11 memory storing the contents of the container. (The reality is more complex, but for  
12 purposes of this analysis, the differences are not important.) The existence of this  
13 pointer makes it possible to move the contents of an entire container in constant  
14 time: just copy the pointer to the container's contents from the source container to  
15 the target, and set the source's pointer to null:

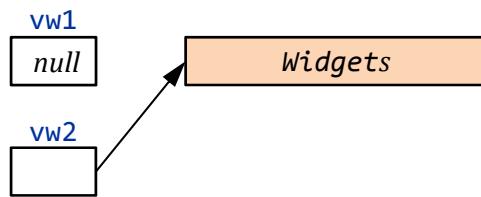
```
std::vector<Widget> vw1;  
// put data into vw1
```

...

```
// move vw1 into vw2. Runs in  
// constant time. Only ptrs  
// in vw1 and vw2 are modified  
auto vw2 = std::move(vw1);
```

16

17 `std::array` objects lack such a pointer, because the data for a `std::array`'s con-  
18 tents are stored directly in the `std::array` object:



```

std::array<Widget, 10000> aw1; aw1  

// put data into aw1  

...  

// move aw1 into aw2. Runs in Widgets  

// linear time. All elements in Widgets (moved from)  

// aw1 are moved into aw2  

auto aw2 = std::move(aw1); aw2  

Widgets (moved to)  

1  

2 Note that the elements in aw1 are moved into aw2. Assuming that Widget is a type  

3 where moving is faster than copying, moving a std::array of Widget will be  

4 faster than copying the same std::array. So std::array certainly offers move  

5 support. Yet both moving and copying a std::array have linear-time computa-  

6 tional complexity, because each element in the container must be copied or moved.  

7 This is far from the “moving a container is now as cheap as assigning a couple of  

8 pointers” claim that one sometimes hears.  

9 On the other hand, std::string offers constant-time moves and linear-time cop-  

10 ies. That makes it sound like moving is faster than copying, but that may not be the  

11 case. Many string implementations employ the small string optimization (SSO).  

12 With the SSO, “small” strings (e.g., those with a capacity of no more than 15 charac-  

13 ters) are stored in a buffer within the std::string object; no heap-allocated  

14 storage is used. Moving small strings using an SSO-based implementation is no  

15 faster than copying them, because the copy-only-a-pointer trick that generally un-  

16 derlies the performance advantage of moves over copies isn’t applicable.  

17 The motivation for the SSO is extensive evidence that short strings are the norm  

18 for many applications. Using an internal buffer to store the contents of such strings  

19 eliminates the need to dynamically allocate memory for them, and that’s typically  

20 an efficiency win. An implication of the win, however, is that moves are no faster  

21 than copies, though one could just as well take a glass-half-full approach and say  

22 that for such strings, copying is no slower than moving.

```

1 Even for types supporting speedy move operations, some seemingly sure-fire  
2 move situations can end up making copies. Item 14 explains that some container  
3 operations in the Standard Library offer the strong exception safety guarantee and  
4 that to ensure that legacy C++98 code dependent on that guarantee isn't broken  
5 when upgrading to C++11, the underlying copy operations may be replaced with  
6 move operations only if the move operations are known to not throw. A conse-  
7 quence is that even if a type offers move operations that are more efficient than  
8 the corresponding copy operations, and even if, at a particular point in the code, a  
9 move operation would generally be appropriate (e.g., if the source object is an  
10 rvalue), compilers might still be forced to invoke a copy operation because the cor-  
11 responding move operation isn't declared `noexcept`.

12 There are thus several scenarios in which C++11's move semantics do you no  
13 good:

14 • **No move operations:** The object to be moved from fails to offer move opera-  
15 tions. The move request therefore becomes a copy request.

16 • **Move not faster:** The object to be moved from has move operations that are  
17 no faster than its copy operations.

18 • **Move not usable:** The context in which the moving would take place requires  
19 a `nothrow` move operation, but that operation isn't declared `noexcept`.

20 It's worth mentioning, too, another scenario where move semantics offers no effi-  
21 ciency gain:

22 • **Source object is lvalue:** With very few exceptions (see e.g., Item 25) only  
23 rvalues may be used as the source of a move operation.

24 But the title of this Item is to *assume* that move operations are not present, not  
25 cheap, and not used. This is typically the case in generic code, e.g., when writing  
26 templates, because you don't know all the types you're working with. In such cir-  
27 cumstances, you must be as conservative about copying objects as you were in  
28 C++98—before move semantics existed. This is also the case for “unstable” code,  
29 i.e., code where the characteristics of the types being used are subject to relatively  
30 frequent modification.

Often, however, you know the types your code uses, and you can rely on their characteristics not changing (e.g., whether they support inexpensive move operations). When that's the case, you don't need to make assumptions. You can simply look up the move support details for the types you're using. If those types offer cheap move operations, and if you're using objects in contexts where those move operations will be invoked, you can safely rely on move semantics to replace copy operations with their less expensive move counterparts.

### Things to remember

- Assume that move operations are not present, not cheap, and not used.
- In code with known types or support for move semantics, there is no need for assumptions.

## Item 30: Familiarize yourself with perfect forwarding failure cases.

One of the features most prominently emblazoned on the C++11 box is perfect forwarding. *Perfect* forwarding. It's *perfect!* Alas, tear the box open, and you'll find that there's "perfect" (the ideal), and then there's "perfect" (the reality). C++11's perfect forwarding is very good, but it achieves true perfection only if you're willing to overlook an epsilon or two. This Item is devoted to familiarizing you with the epsilons.

Before embarking on our epsilon exploration, it's worthwhile to review what's meant by "perfect forwarding." "Forwarding" just means that one function passes—*forwards*—its parameters to another function. The goal is for the second function (the one being forwarded to) to receive the same objects that the first function (the one doing the forwarding) received. That rules out by-value parameters, because they're *copies* of what the original caller passed in; we want the forwarded-to function to be able to work with the originally-passed-in objects. Pointer parameters are also ruled out, because we don't want to force callers to pass pointers. When it comes to general-purpose forwarding, we'll be dealing with parameters that are references.

1 *Perfect forwarding* means we don't just forward objects, we also forward their sa-  
2 lient characteristics: their types, whether they're lvalues or rvalues, and whether  
3 they're `const` or `volatile`. In conjunction with the observation that we'll be deal-  
4 ing with reference parameters, this implies that we'll be using universal references  
5 (see Item 24), because only universal reference parameters encode information  
6 about the lvalueness and rvalueness of the arguments that are passed to them.

7 Let's assume we have some function `f`, and we'd like to write a function (in truth, a  
8 function template) that forwards to it. The core of what we need looks like this:

```
9 template<typename T>
10 void fwd(T&& param) // accept any argument
11 {
12     f(std::forward<T>(param)); // forward it to f
13 }
```

14 Forwarding functions are, by their nature, generic. The `fwd` template, for example,  
15 accepts any type of argument, and it forwards whatever it gets. A logical extension  
16 of this genericity is for forwarding functions to be not just templates, but *variadic*  
17 templates, thus accepting any number of arguments. The variadic form for `fwd`  
18 looks like this:

```
19 template<typename... Ts>
20 void fwd(Ts&&... params) // accept any arguments
21 {
22     f(std::forward<Ts>(params)...); // forward them to f
23 }
```

24 This is the form you'll see in, among other places, the standard containers' em-  
25 placement functions (see Item 42) and the smart pointer factory functions,  
26 `std::make_shared` and `std::make_unique` (see Item 21).

27 Given our target function `f` and our forwarding function `fwd`, perfect forwarding  
28 *fails* if calling `f` with a particular argument does one thing, but calling `fwd` with the  
29 same argument does something different:

```
30 f( expression ); // if this does one thing,
31 fwd( expression ); // but this does something else, fwd fails
32 // to perfectly forward expression to f
```

1 Several kinds of arguments lead to this kind of failure. Knowing what they are and  
2 how to work around them is important, so let's tour the arguments that can't be  
3 perfect-forwarded.

4 **Braced initializers**

5 Suppose `f` is declared like this:

6 `void f(const std::vector<int>& v);`

7 In that case, calling `f` with a braced initializer compiles,

8 `f({ 1, 2, 3 });` // fine, "{1, 2, 3}" implicitly  
9 // converted to std::vector<int>

10 but passing the same braced initializer to `fwd` doesn't compile:

11 `fwd({ 1, 2, 3 });` // error! doesn't compile

12 That's because the use of a braced initializer is a perfect forwarding failure case.

13 All such failure cases have the same cause. In a direct call to `f` (such as  
14 `f({ 1, 2, 3 })`), compilers see the arguments passed at the call site, and they  
15 see the types of the parameters declared by `f`. They compare the arguments at the  
16 call site to the parameter declarations to see if they're compatible, and, if neces-  
17 sary, they perform implicit conversions to make the call succeed. In the example  
18 above, they generate a temporary `std::vector<int>` object from { 1, 2, 3 }  
19 so that `f`'s parameter `v` has a `std::vector<int>` object to bind to.

20 When calling `f` indirectly through the forwarding function template `fwd`, compilers  
21 no longer compare the arguments passed at `fwd`'s call site to the parameter decla-  
22 rations in `f`. Instead, they *deduce* the types of the arguments being passed to `fwd`,  
23 and they compare the deduced types to `f`'s parameter declarations. Perfect for-  
24 warding fails when either of the following occurs:

- 25 • **Compilers are unable to deduce a type** for one or more of `fwd`'s parameters.

26 In this case, the code fails to compile.

- 27 • **Compilers deduce the “wrong” type** for one or more of `fwd`'s parameters.

28 Here, “wrong” could mean that `fwd`'s instantiation won't compile with the

1 types that were deduced, but it could also mean that the call to `f` using `fwd`'s  
2 deduced types behaves differently from a direct call to `f` with the arguments  
3 that were passed to `fwd`. One source of such divergent behavior would be if `f`  
4 were an overloaded function name, and, due to “incorrect” type deduction, the  
5 overload of `f` called inside `fwd` were different from the overload that would be  
6 invoked if `f` were called directly.

7 In the “`fwd({ 1, 2, 3 })`” call above, the problem is that passing a braced ini-  
8 tializer to a function template parameter that's not declared to be a  
9 `std::initializer_list` is decreed to be, as the Standard puts it, a “non-  
10 deduced context.” In plain English, that means that compilers are forbidden from  
11 deducing a type for the expression `{ 1, 2, 3 }` in the call to `fwd`, because `fwd`'s  
12 parameter isn't declared to be a `std::initializer_list`. Being prevented from  
13 deducing a type for `fwd`'s parameter, compilers must understandably reject the  
14 call.

15 Interestingly, Item 2 explains that type deduction succeeds for `auto` variables ini-  
16 tialized with a braced initializer. Such variables are deemed to be  
17 `std::initializer_list` objects, and this affords a simple workaround for cases  
18 where the type the forwarding function should deduce is a  
19 `std::initializer_list`: declare a local variable using `auto`, then pass the local  
20 variable to the forwarding function:

```
21 auto il = { 1, 2, 3 };      // il's type deduced to be
22                           // std::initializer_list<int>
23 fwd(il);                  // fine, perfect-forwards il to f
```

## 24 **0 or NULL as null pointers**

25 Item 8 explains that when you try to pass `0` or `NULL` as a null pointer to a template,  
26 type deduction goes awry, deducing an integral type (typically `int`) instead of a  
27 pointer type for the argument you pass. The result is that neither `0` nor `NULL` can  
28 be perfect-forwarded as a null pointer. The fix is easy, however: pass `nullptr` in-  
29 stead of `0` or `NULL`. For details, consult Item 8.

1   **Declaration-only integral static const data members**

2   As a general rule, there's no need to define integral `static const` data members  
3   in classes; declarations alone suffice. That's because compilers perform *const-*  
4   *propagation* on such members' values, thus eliminating the need to set aside  
5   memory for them. For example, consider this code:

```
6  class Widget {  
7  public:  
8      static const std::size_t MinVals = 28; // MinVals' declaration  
9      ...  
10 };  
11 ... // no defn. for MinVals  
  
12 std::vector<int> widgetData;  
13 widgetData.reserve(Widget::MinVals); // use of MinVals
```

14 Here, we're using `Widget::MinVals` (henceforth simply `MinVals`) to specify  
15 `widgetData`'s initial capacity, even though `MinVals` lacks a definition. Compilers  
16 work around the missing definition (as they are required to do) by plopping the  
17 value `28` into all places where `MinVals` is mentioned; the fact that no storage has  
18 been set aside for `MinVals`' value is unproblematic. If `MinVals`' address were to  
19 be taken (e.g., if somebody created a pointer to `MinVals`), then `MinVals` would  
20 require storage (so that the pointer had something to point to), and the code  
21 above, though it would compile, would fail at link-time until a definition for `Min-`  
22 `Vals` was provided.

23 With that in mind, imagine that `f` (the function `fwd` forwards its argument to) is  
24 declared like this:

```
25 void f(std::size_t val);
```

26 Calling `f` with `MinVals` is fine, because compilers will just replace `MinVals` with  
27 its value:

```
28 f(Widget::MinVals); // fine, treated as "f(28)"
```

29 Alas, things may not go so smoothly if we try to call `f` through `fwd`:

```
30 fwd(Widget::MinVals); // error! shouldn't link
```

1 This code will compile, but it shouldn't link. If that reminds you of what happens if  
2 we write code that takes `MinVals`' address, that's good, because the underlying  
3 problem is the same.

4 Although nothing in the source code takes `MinVals`' address, `fwd`'s parameter is a  
5 universal reference, and references, in the code generated by compilers, are usual-  
6 ly treated like pointers. In the program's underlying binary code (and on the  
7 hardware), pointers and references are essentially the same thing. At this level,  
8 there's truth to the adage that references are simply pointers that are automatical-  
9 ly dereferenced. That being the case, passing `MinVals` by reference is effectively  
10 the same as passing it by pointer, and as such, there has to be some memory for  
11 the pointer to point to. Passing integral `static const` data members by reference,  
12 then, generally requires that they be defined, and that requirement can cause code  
13 using perfect forwarding to fail where the equivalent code without perfect for-  
14 warding succeeds.

15 But perhaps you noticed the weasel words I sprinkled through the preceding dis-  
16 cussion. The code "shouldn't" link. References are "usually" treated like pointers.  
17 Passing integral `static const` data member by reference "generally" requires  
18 that they be defined. It's almost like I know something I don't really want to tell  
19 you...

20 That's because I do. According to the Standard, passing `MinVals` by reference re-  
21 quires that it be defined. But not all implementations enforce this requirement. So,  
22 depending on your compilers and linkers, you may find that you can perfect-  
23 forward integral `static const` data members that haven't been defined. If you do,  
24 congratulations, but there is no reason to expect such code to port. To make it  
25 portable, simply provide a definition for the integral `static const` data member  
26 in question. For `MinVals`, that'd look like this:

27 `const std::size_t Widget::MinVals; // in Widget's .cpp file`

28 Note that the definition doesn't repeat the initializer (28, in the case of `MinVals`).  
29 Don't stress over this detail, however. If you forget and provide the initializer in  
30 both places, your compilers will complain, thus reminding you to specify it only  
31 once.

## 1 Overloaded function names and template names

2 Suppose our function `f` (the one we keep wanting to forward arguments to via  
3 `fwd`) can have its behavior customized by passing it a function that does some of  
4 its work. Assuming this function takes and returns `ints`, `f` could be declared like  
5 this:

```
6 void f(int (*pf)(int)); // pf = "processing function"
```

7 It's worth noting that `f` could also be declared using a simpler non-pointer syntax.  
8 Such a declaration would look like this, though it'd have the same meaning as the  
9 declaration above:

```
10 void f(int pf(int)); // declares same f as above
```

11 Either way, now suppose we have an overloaded function, `processVal`:

```
12     int processVal(int value);
13     int processVal(int value, int priority);
```

14 We can pass processVal to f,

```
15    f(processVal); // fine
```

16 but it's something of a surprise that we can. `f` demands a pointer to a function as  
17 its argument, but `processVal` isn't a function pointer or even a function, it's the  
18 name of two different functions. However, compilers know which `processVal`  
19 they need: the one matching `f`'s parameter type. They thus choose the pro-  
20 `cessVal` taking one `int`, and they pass that function's address to `f`.

21 What makes this work is that `f`'s declaration lets compilers figure out which ver-  
22 sion of `processVal` is required. `fwd`, however, being a function template, doesn't  
23 have any information about what type it needs, and that makes it impossible for  
24 compilers to determine which overload should be passed:

```
25   fwd(processVal);           // error! which processVal?
```

26 processVal alone has no type. Without a type, there can be no type deduction,  
27 and without type deduction, we're left with another perfect forwarding failure  
28 case.

1 The same problem arises if we try to use a function template instead of (or in addition to) an overloaded function name. A function template doesn't represent one  
2 function, it represents *many* functions:

```
4 template<typename T>
5 T workOnVal(T param)           // template for processing values
6 { ... }

7 fwd(workOnVal);              // error! which workOnVal
8                         // instantiation?
```

9 The way to get a perfect-forwarding function like `fwd` to accept an overloaded  
10 function name or a template name is to manually specify the overload or instantiation  
11 you want to have forwarded. For example, you can create a function pointer of  
12 the same type as `f`'s parameter, initialize that pointer with `processVal` or `wor-`  
13 `kOnVal` (thus causing the proper version of `processVal` to be selected or the  
14 proper instantiation of `workOnVal` to be generated), and pass the pointer to `fwd`:

```
15 using ProcessFuncType =           // make typedef;
16     int (*)(int);                // see Item 9

17 ProcessFuncType processValPtr = processVal; // specify needed
18                                // signature for
19                                // processVal

20 fwd(processValPtr);            // fine

21 fwd(static_cast<ProcessFuncType>(workOnVal)); // also fine
```

22 Of course, this requires that you know the type of function pointer that `fwd` is for-  
23 warding to. It's not unreasonable to assume that a perfect-forwarding function will  
24 document that. After all, perfect-forwarding functions are designed to accept *any-*  
25 *thing*, so if there's no documentation telling you what to pass, how would you  
26 know?

1    **Bitfields**

2    The final failure case for perfect forwarding is when a bitfield is used as a function  
3    argument. To see what this means in practice, observe that an IPv4 header can be  
4    modeled as follows:<sup>†</sup>

```
5    struct IPv4Header {  
6       std::uint32_t version:4,  
7              IHL:4,  
8              DSCP:6,  
9              ECN:2,  
10          totalLength:16;  
11       ...  
12   };
```

13   If our long-suffering function `f` (the perennial target of our forwarding function  
14   `fwd`) is declared to take a `std::size_t` parameter, calling it with, say, the `total-`  
15   `Length` field of an `IPv4Header` object compiles without fuss:

```
16   void f(std::size_t sz);                    // function to call  
17   IPv4Header h;  
18   ...  
19   f(h.totalLength);                          // fine
```

20   Trying to forward `h.totalLength` to `f` via `fwd`, however, is a different story:

```
21   fwd(h.totalLength);                        // error!
```

22   The problem is that `fwd`'s parameter is a reference, and `h.totalLength` is a non-  
23   `const` bitfield. That may not sound so bad, but the C++ Standard condemns the  
24   combination in unusually clear prose: “A `non-const` reference shall not be bound  
25   to a bit-field.” There’s an excellent reason for the prohibition. Bitfields may consist  
26   of arbitrary parts of machine words (e.g., bits 3-5 of a 32-bit `int`), but there’s no  
27   way to directly address such things. I mentioned earlier that references and point-  
28   ers are the same thing at the hardware level, and just as there’s no way to create a

---

<sup>†</sup> 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 pointer to arbitrary bits (C++ dictates that the smallest thing you can point to is a  
2 `char`), there's no way to bind a reference to arbitrary bits, either.

3 Working around the impossibility of perfect-forwarding a bitfield is easy, once you  
4 realize that any function that accepts a bitfield as an argument will receive a *copy*  
5 of the bitfield's value. After all, no function can bind a reference to a bitfield, nor  
6 can any function accept pointers to bitfields, because pointers to bitfields don't  
7 exist. The only kinds of parameters to which a bitfield can be passed are by-value  
8 parameters and, counterintuitively, references-to-`const`. In the case of by-value  
9 parameters, the called function obviously receives a copy of the value in the bit-  
10 field, and it turns out that in the case of a reference-to-`const` parameter, the  
11 Standard requires that the reference actually bind to a *copy* of the bitfield's value  
12 that's stored in an object of some standard integral type (e.g., `int`). References-to-  
13 `const` don't bind to bitfields, they bind to "normal" objects into which the values  
14 of the bitfields have been copied.

15 The key to passing a bitfield into a perfect-forwarding function, then, is to take ad-  
16 vantage of the fact that the forwarded-to function will always receive a copy of the  
17 bitfield's value. You can thus make a copy yourself and call the forwarding function  
18 with the copy. In the case of our example with `IPv4Header`, this code would do the  
19 trick:

```
20 // copy bitfield value; see Item 6 for info on init. form
21 auto length = static_cast<std::uint16_t>(h.totalLength);
22 fwd(length);                                // forward the copy
```

### 23 **Upshot**

24 In most cases, perfect forwarding works exactly as advertised. You rarely have to  
25 think about it. But when it doesn't work—when reasonable-looking code fails to  
26 compile or, worse, compiles but doesn't behave the way you anticipate—it's im-  
27 portant to know about perfect forwarding's imperfections. Equally important is  
28 knowing how to work around them. In most cases, this is straightforward.

1    **Things to remember**

- 2    ♦ Perfect forwarding fails when template type deduction fails or when it deduces  
3    the wrong type.
- 4    ♦ The kinds of arguments that lead to perfect forwarding failure are braced ini-  
5    tializers, null pointers expressed as `0` or `NULL`, declaration-only integral `const`  
6    `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 18 and 19), and they make  
14   the specification of predicates for condition variables in the threading API equally  
15   straightforward (see Item 39). Beyond the Standard Library, lambdas facilitate the  
16   on-the-fly specification of callback functions, interface adaption functions, and  
17   context-specific functions for one-off calls. Lambdas really make C++ a more pleas-  
18   ant programming 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                   [] (int val) { return 0 < val && val < 10; });
```

- 1       the highlighted expression is the lambda.
- 2       • A *closure* is the runtime object created by a lambda. Depending on the capture  
3       mode, closures hold copies of or references to the captured data. In the call to  
4       `std::find_if` above, the closure is the object that's passed at run time as the  
5       third argument to `std::find_if`.
- 6       • A *closure class* is a class from which a closure is instantiated. Each lambda  
7       causes compilers to generate a unique closure class. The statements inside a  
8       lambda become executable instructions in the member functions of its closure  
9       class.
- 10      A lambda is often used to create a closure that's used only as an argument to a  
11     function. That's the case in the call to `std::find_if` above. However, closures  
12     may generally be copied, so it's usually possible to have multiple closures of a clo-  
13     sure type corresponding to a single lambda. For example, in the following code,

```
14 {                                     // begin a scope
15   int x;                           // x is local variable
16   ...
17   auto c1 =                         // c1 is copy of the
18     [x](int y) { return x * y > 55; }; // closure produced
19                               // by the lambda
20   auto c2 = c1;                     // c2 is copy of c1
21   auto c3 = c2;                     // c3 is copy of c2
22   ...
23 }                                     // end the scope
```

24 `c1, c2, and c3` are all copies of the closure produced by the lambda.

25 Informally, it's perfectly acceptable to blur the lines between lambdas, closures,  
26 and closure classes. But in the Items that follow, it's often important to distinguish  
27 what exists during compilation (lambdas and closure classes), what exists at run  
28 time (closures), and how they relate to one another.

## 1 Item 31: 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 a closure to refer to a local variable or a parameter  
10 that's available in the scope where the lambda is defined. If the lifetime of a closure  
11 created from that lambda exceeds the lifetime of the local variable or parameter,  
12 the reference in the closure will dangle. For example, suppose we have a container  
13 of filtering functions, each of which takes an `int` and returns a `bool` indicating  
14 whether a passed-in value satisfies the filter:

```
15 using FilterContainer = std::vector<std::function<bool(int)>>; // typedef;
16 // 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 42 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 run time, 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; } // ref to
31     ); // divisor
32 } // will // 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! ref to
10        { return value % divisor == 0; } // divisor will
11    );                                // still dangle!
```

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 executed immediately (e.g., by being passed to  
18 an STL algorithm), there is no risk that its closure will outlive the local variables  
19 and parameters in the environment where the lambda is created. In that case, you  
20 might argue, there's no risk of dangling references, hence no reason to avoid a de-  
21 fault by-reference capture mode. For example, our filtering lambda might be used  
22 only as an argument to C++11's `std::all_of`, which returns whether all elements  
23 in a 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; }))
```

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.

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 is 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 21 for
12         // std::make_unique
13
14     pw->addFilter(); // add filter that uses
15         // Widget::divisor
16     ...
17 }
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

When a call is made to `doSomeWork`, a filter is created that depends on the `Widget` object produced by `std::make_unique`, i.e., a filter that contains a copy of a 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 `std::unique_ptr` managing its lifetime (see Item 18). From that point on, `filters` contains an entry with a dangling pointer.

This particular problem can be solved by making a local copy of the data member you want to capture and then capturing the copy:

```
void Widget::addFilter() const
{
    auto divisorCopy = divisor; // copy data member

    filters.emplace_back(
        [divisorCopy](int value) // capture the copy
        { return value % divisorCopy == 0; } // use the copy
    );
}
```

To be honest, if you take this approach, default by-value capture will work, too,

```
void Widget::addFilter() const
{
    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 32):

```
10 void Widget::addFilter() const  
11 {  
12   filters.emplace_back( // C++14:  
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   ++divisor; // modify divisor
6 }
```

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`). Practical-  
14 ly 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 dangling references.  
20 • Default by-value capture is susceptible to dangling pointers (especially `this`),  
21 and it misleadingly suggests that lambdas are self-contained.

## 22 Item 32: 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 31 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 21 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 25 explains
9     : pw(std::move(ptr)) {}                      // use of std::move
10    bool operator()() const
11    { return pw->isValidated() && pw->isArchived(); }
12
13 private:
14     DataType pw;
15 }
```

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        [](const 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 a  
17 callable object. Subsequent arguments represent values to be passed to that object.

18 A bind object contains copies of all the arguments passed to `std::bind`. For each  
19 lvalue argument, the corresponding object in the bind object is copy constructed.  
20 For each rvalue, it's move constructed. In this example, the second argument is an  
21 rvalue (the result of `std::move`—see Item 23), so `data` is move-constructed into  
22 the bind object. This move construction is the crux of move capture emulation, be-  
23 cause moving an rvalue into a bind object is how we work around the inability to  
24 move an rvalue into a C++11 closure.

25 When a bind object is “called” (i.e., its function call operator is invoked) the argu-  
26 ments it stores are passed to the callable object originally passed to `std::bind`. In  
27 this example, that means that when `func` (the bind object) is called, the move-  
28 constructed copy of `data` inside `func` is passed as an argument to the lambda that  
29 was passed to `std::bind`.

30 This lambda is the same as the lambda we'd use in C++14, except a parameter, `da-`  
31 `ta`, has been added to correspond to our pseudo-move-captured object. This pa-  
32 rameter is an lvalue reference to the copy of `data` in the bind object. (It's not an

1 rvalue reference, because although the expression used to initialize the copy of  
2 `data ("std::move(data)")` is an rvalue, the copy of `data` itself is an lvalue.) Uses  
3 of `data` inside the lambda will thus operate on the move-constructed copy of `data`  
4 inside the bind object.

5 By default, the `operator()` member function inside the closure class generated  
6 from a lambda is `const`. That has the effect of rendering all data members in the  
7 closure `const` within the body of the lambda. The move-constructed copy of `data`  
8 inside the bind object is not `const`, however, so to prevent that copy of `data` from  
9 being modified inside the lambda, the lambda's parameter is declared reference-to-  
10 `const`. If the lambda were declared `mutable`, `operator()` in its closure class  
11 would not be declared `const`, and it would be appropriate to omit `const` in the  
12 lambda's parameter declaration:

```
13 auto func =
14     std::bind(
15         [](std::vector<double>& data) mutable // of init capture
16         { /* uses of data */ },
17         std::move(data)
18     );
```

19 Because a bind object stores copies of all the arguments passed to `std::bind`, the  
20 bind object in our example contains a copy of the closure produced by the lambda  
21 that is its first argument. The lifetime of the closure is therefore the same as the  
22 lifetime of the bind object. That's important, because it means that as long as the  
23 closure exists, the bind object containing the pseudo-move-captured object exists,  
24 too.

25 If this is your first exposure to `std::bind`, you may need to consult your favorite  
26 C++11 reference before all the details of the foregoing discussion fall into place.  
27 Even if that's the case, these fundamental points should be clear:

- 28 • It's not possible to move-construct an object into a C++11 closure, but it is pos-  
29 sible to move-construct an object into a C++11 bind object.
- 30 • Emulating move-capture in C++11 consists of move-constructing an object into  
31 a bind object, then passing the move-constructed object to the lambda by ref-  
32 erence.

1     • Because the lifetime of the bind object is the same as that of the closure, it's  
2       possible to treat objects in the bind object as if they were in the closure.

3     As a second example of using `std::bind` to emulate move capture, here's the  
4       C++14 code we saw earlier to create a `std::unique_ptr` in a closure:

```
5 auto func = [pw = std::make_unique<Widget>()] // as before,  
6 { return pw->isValidated() // create pw  
7     && pw->isArchived(); }; // in closure
```

8     And here's the C++11 emulation:

```
9 auto func = std::bind(  
10    [](const std::unique_ptr<Widget>& pw)  
11    { return pw->isValidated()  
12        && pw->isArchived(); },  
13    std::make_unique<Widget>()  
14 );
```

15     It's ironic that I'm showing how to use `std::bind` to work around limitations in  
16       C++11 lambdas, because in Item 34, I advocate the use of lambdas over  
17       `std::bind`. However, that Item explains that there are some cases in C++11  
18       where `std::bind` can be useful, and this is one of them. (In C++14, features such  
19       as init capture and `auto` parameters eliminate those cases.)

## 20     **Things to Remember**

- 21       • Use C++14's init capture to move objects into closures.
- 22       • In C++11, emulate init capture via hand-written classes or `std::bind`.

## 23     **Item 33: Use `decltype` on `auto&` parameters to 24            `std::forward` them.**

25     One of the most exciting features of C++14 is *generic lambdas*—lambdas that use  
26       `auto` in their parameter specifications. The implementation of this feature is  
27       straightforward: `operator()` in the lambda's closure class is a template. Given  
28       this lambda, for example,

```
29 auto f = [](auto x){ return func(normalize(x)); };
```

30     the closure class's function call operator looks like this:

```

1 class SomeCompilerGeneratedClassName {
2 public:
3     template<typename T> // see Item 3 for
4         auto operator()(T x) const // auto return type
5             { return func(normalize(x)); }

6     ... // other closure class
7 };

```

8 In this example, the only thing the lambda does with its parameter `x` is forward it  
9 to `normalize`. If `normalize` treats lvalues differently from rvalues, this lambda  
10 isn't written properly, because it always passes an lvalue (the parameter `x`) to  
11 `normalize`, even if the argument that was passed to the lambda was an rvalue.

12 The correct way to write the lambda is to have it perfect-forward `x` to `normalize`.  
13 Doing that requires two changes to the code. First, `x` has to become a universal ref-  
14 erence (see Item 24), and second, it has to be passed to `normalize` via  
15 `std::forward` (see Item 25). In concept, these are trivial modifications:

```

16 auto f = [](auto&& x)
17     { return func(normalize(std::forward<???>(x))); };

```

18 Between concept and realization, however, is the question of what type to pass to  
19 `std::forward`, i.e., to determine what should go where I've written `???` above.

20 Normally, when you employ perfect forwarding, you're in a template function tak-  
21 ing a type parameter `T`, so you just write `std::forward<T>`. In the generic lamb-  
22 da, though, there's no type parameter `T` available to you. There is a `T` in the tem-  
23 platized `operator()` inside the closure class generated by the lambda, but it's not  
24 possible to refer to it from the lambda, so it does you no good.

25 Item 28 explains that if an lvalue argument is passed to a universal reference pa-  
26 rameter, the type of that parameter becomes an lvalue reference. If an rvalue is  
27 passed, the parameter becomes an rvalue reference. This means that in our lamb-  
28 da, we can determine whether the argument passed was an lvalue or an rvalue by  
29 inspecting the type of the parameter `x`. `decltype` gives us a way to do that (see  
30 Item 3). If an lvalue was passed in, `decltype(x)` will produce a type that's an  
31 lvalue reference. If an rvalue was passed, `decltype(x)` will produce an rvalue  
32 reference type.

1 Item 28 also explains that when calling `std::forward`, convention dictates that  
2 the type argument be an lvalue reference to indicate an lvalue and a non-reference  
3 to indicate an rvalue. In our lambda, if `x` is bound to an lvalue, `decltype(x)` will  
4 yield an lvalue reference. That conforms to convention. However, if `x` is bound to  
5 an rvalue, `decltype(x)` will yield an rvalue reference instead of the customary  
6 non-reference.

7 But look at the sample C++14 implementation for `std::forward` from Item 28:

```
8 template<typename T> // in namespace
9     T&& forward(remove_reference_t<T>& param) // std
10    {
11        return static_cast<T&&>(param);
12    }
```

13 If client code wants to perfect-forward an rvalue of type `Widget`, it normally in-  
14stantiates `std::forward` with the type `Widget` (i.e., a non-reference type), and the  
15 `std::forward` template yields this function:

```
16 Widget&& forward(Widget& param) // instantiation of
17 { // std::forward when
18     return static_cast<Widget&&>(param); // T is Widget
19 }
```

20 But consider what would happen if the client code wanted to perfect-forward the  
21 same rvalue of type `Widget`, but instead of following the convention of specifying `T`  
22 to be a non-reference type, it specified it to be an rvalue reference. That is, consid-  
23 er what would happen if `T` were specified to be `Widget&&`. After initial instantia-  
24 tion of `std::forward` and application of `std::remove_reference_t`, but before  
25 reference collapsing (once again, see Item 28), `std::forward` would look like  
26 this:

```
27 Widget&& && forward(Widget& param) // instantiation of
28 { // std::forward when
29     return static_cast<Widget&& &&>(param); // T is Widget&&
30 } // (before reference-
31 // collapsing)
```

32 Applying the reference-collapsing rule that an rvalue reference to an rvalue refer-  
33 ence becomes a single rvalue reference, this instantiation emerges:

```
1 Widget&& forward(Widget& param)           // instantiation of
2 {   // std::forward when
3     return static_cast<Widget&&>(param); // T is Widget&&
4 }   // (after reference-
5   // collapsing)
```

6 If you compare this instantiation with the one that results when `std::forward` is  
7 called with `T` set to `Widget`, you'll see that they're identical. That means that in-  
8 stantiating `std::forward` with an rvalue reference type yields the same result as  
9 instantiating it with a non-reference type.

10 That's wonderful news, because `decltype(x)` yields an rvalue reference type  
11 when an rvalue is passed as an argument to our lambda's parameter `x`. We estab-  
12 lished above that when an lvalue is passed to our lambda, `decltype(x)` yields the  
13 customary type to pass to `std::forward`, and now we realize that for rvalues,  
14 `decltype(x)` yields a type to pass to `std::forward` that's not conventional, but  
15 that nevertheless yields the same outcome as the conventional type. So for both  
16 lvalues and rvalues, passing `decltype(x)` to `std::forward` gives us the result  
17 we want. Our perfect-forwarding lambda can therefore be written like this:

```
18 auto f =
19   [](auto&& param)
20   {
21     return
22       func(normalize(std::forward<decltype(param)>(param)));
23   };
```

24 From there, it's just a hop, skip, and six dots to a perfect-forwarding lambda that  
25 accepts not just a single parameter, but any number of parameters, because C++14  
26 lambdas can also be variadic:

```
27 auto f =
28   [](auto&&... params)
29   {
30     return
31       func(normalize(std::forward<decltype(params)>(params)...));
32   };
```

### 33 Things to Remember

34 ♦ Use `decltype` on `auto&&` parameters to `std::forward` them.

1    **Item 34: Prefer lambdas to std::bind.**

2    `std::bind` is the C++11 successor to C++98's `std::bind1st` and `std::bind2nd`,  
3    but, informally, it's been part of the Standard Library since 2005. That's when the  
4    Standardization Committee adopted a document known as TR1, which included  
5    `bind`'s specification. (In TR1, `bind` was in a different namespace, so it was  
6    `std::tr1::bind`, not `std::bind`, and a few interface details were different.) This  
7    history means that some programmers have a decade or more of experience using  
8    `std::bind`. If you're one of them, you may be reluctant to abandon a tool that's  
9    served you well. That's understandable, but in this case, change is good, because in  
10   C++11, lambdas are almost always a better choice than `std::bind`. As of C++14,  
11   the case for lambdas isn't just stronger, it's downright iron-clad.

12   This Item assumes that you're familiar with `std::bind`. If you're not, you'll want  
13   to acquire a basic understanding before continuing. Such an understanding is  
14   worthwhile in any case, because you never know when you might encounter uses  
15   of `std::bind` in a code base you have to read or maintain.

16   As in Item 32, I refer to the function objects returned from `std::bind` as *bind objects*.

18   The most important reason to prefer lambdas over `std::bind` is that lambdas are  
19   more readable. Suppose, for example, we have a function to set up an audible  
20   alarm:

```
21 // typedef for a point in time (see Item 9 for syntax)
22 using Time = std::chrono::steady_clock::time_point;
23 // see Item 10 for "enum class"
24 enum class Sound { Beep, Siren, Whistle };
25 // typedef for a length of time
26 using Duration = std::chrono::steady_clock::duration;
27 // at time t, make sound s for duration d
28 void setAlarm(Time t, Sound s, Duration d);
```

29   Further suppose that at some point in the program, we've determined we'll want  
30   an alarm that will go off an hour after it's set and that will stay on for 30 seconds.

1 The alarm sound, however, remains undecided. We can write a lambda that revises  
2 `setAlarm`'s interface so that only a sound needs to be specified:

```
3 // setSoundL ("L" for "lambda") is a function object allowing a
4 // sound to be specified for a 30-sec alarm to go off an hour
5 // after it's set
6 auto setSoundL =
7 [](Sound s)
8 {
9     // make std::chrono components available w/o qualification
10    using namespace std::chrono;
11
12    setAlarm(steady_clock::now() + hours(1), // alarm to go off
13              s, // in an hour for
14              seconds(30)); // 30 seconds
15}
```

15 I've highlighted the call to `setAlarm` inside the lambda. This is a normal-looking  
16 function call, and even a reader with little lambda experience can see that the pa-  
17 rameter `s` passed to the lambda is passed as an argument to `setAlarm`.

18 We can streamline this code in C++14 by availing ourselves of the standard suffix-  
19 es for seconds (`s`), milliseconds (`ms`), hours (`h`), etc., that build on C++11's support  
20 for user-defined literals. These suffixes are implemented in the `std::literals`  
21 namespace, so the above code can be rewritten as follows:

```
22 auto setSoundL =
23 [](Sound s)
24 {
25     using namespace std::chrono;
26     using namespace std::literals; // for C++14 suffixes
27
28     setAlarm(steady_clock::now() + 1h, // C++14, but
29              s, // same meaning
30              30s); // as above
31}
```

31 Our first attempt to write the corresponding `std::bind` call is below. It has an  
32 error that we'll fix in a moment, but the correct code is more complicated, and  
33 even this simplified version brings out some important issues:

```
34 using namespace std::chrono; // as above
35 using namespace std::literals;
36 using namespace std::placeholders; // needed for use of "_1"
```

```
1 auto setSoundB = // "B" for "bind"
2     std::bind(setAlarm,
3                 steady_clock::now() + 1h, // incorrect! see below
4                 _1,
5                 30s);
```

6 I'd like to highlight the call to `setAlarm` here as I did in the lambda, but there's no  
7 call to highlight. Readers of this code simply have to know that calling `setSoundB`  
8 invokes `setAlarm` with the time and duration specified in the call to `std::bind`.  
9 To the uninitiated, the placeholder "`_1`" is essentially magic, but even readers in  
10 the know have to mentally map from the number in that placeholder to its position  
11 in the `std::bind` parameter list in order to understand that the first argument in  
12 a call to `setSoundB` is passed as the second argument to `setAlarm`. The type of  
13 this argument is not identified in the call to `std::bind`, so readers have to consult  
14 the `setAlarm` declaration to determine what kind of argument to pass to  
15 `setSoundB`.

16 But, as I said, the code isn't quite right. In the lambda, it's clear that the expression  
17 "`steady_clock::now() + 1h`" is an argument to `setAlarm`; it will be evaluated  
18 when `setAlarm` is called. That makes sense: we want the alarm to go off an hour  
19 after invoking `setAlarm`. In the `std::bind` call, however,  
20 "`steady_clock::now() + 1h`" is passed as an argument to `std::bind`, not to  
21 `setAlarm`. That means that the expression will be evaluated when `std::bind` is  
22 called, and the time resulting from that expression will be stored inside the result-  
23 ing bind object. As a consequence, the alarm will be set to go off an hour *after the*  
24 *call to std::bind*, not an hour after the call to `setAlarm`!

25 Fixing the problem requires telling `std::bind` to defer evaluation of the expres-  
26 sion until `setAlarm` is called, and the way to do that is to nest a second call to  
27 `std::bind` inside the first one:

```
28 auto setSoundB =
29     std::bind(setAlarm,
30               std::bind(std::plus<>(), steady_clock::now(), 1h),
31               _1,
32               30s);
```

33 If you're familiar with the `std::plus` template from C++98, you may be surprised  
34 to see that in this code, no type is specified between the angle brackets, i.e., the

1 code contains “`std::plus<>`”, not “`std::plus<type>`”. In C++14, the template  
2 type argument for the standard operator templates can generally be omitted, so  
3 there’s no need to provide it here. C++11 offers no such feature, so the C++11  
4 `std::bind` equivalent to the lambda is:

```
5 using namespace std::chrono; // as above
6 using namespace std::placeholders;

7 auto setSoundB =
8     std::bind(setAlarm,
9             std::bind(std::plus<steady_clock::time_point>(),
10                 steady_clock::now(),
11                 hours(1)),
12                 _1,
13                 seconds(30));
```

14 If, at this point, the lambda’s not looking a lot more attractive, you should probably  
15 have your eyesight checked.

16 When `setAlarm` is overloaded, a new issue arises. Suppose there’s an overload  
17 taking a fourth parameter specifying the alarm volume:

```
18 enum class Volume { Normal, Loud, LoudPlusPlus };

19 void setAlarm(Time t, Sound s, Duration d, Volume v);
```

20 The lambda continues to work as before, because overload resolution chooses the  
21 three-argument version of `setAlarm`:

```
22 auto setSoundL = // same as before
23 [](Sound s)
24 {
25     using namespace std::chrono;
26     setAlarm(steady_clock::now() + 1h, // fine, calls
27             s, // 3-arg version
28             30s); // of setAlarm
29 };
```

30 The `std::bind` call, on the other hand, now fails to compile:

```
31 auto setSoundB = // error! which
32     std::bind(setAlarm, // setAlarm?
33             std::bind(std::plus<>(),
34                 steady_clock::now(),
35                 1h),
```

```
1     _1,
2     30s);
```

3 The problem is that compilers have no way to determine which of the two  
4 `setAlarm` functions they should pass to `std::bind`. All they have is a function  
5 name, and the name alone is ambiguous.

6 To get the `std::bind` call to compile, `setAlarm` must be cast to the proper func-  
7 tion pointer type:

```
8 using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);
9 auto setSoundB =
10     std::bind(static_cast<SetAlarm3ParamType>(setAlarm), // okay
11             std::bind(std::plus<>(),
12                     steady_clock::now(),
13                     1h),
14             _1,
15             30s);
```

16 But this brings up another difference between lambdas and `std::bind`. Inside the  
17 function call operator for `setSoundL` (i.e., the function call operator of the lamb-  
18 da's closure class), the call to `setAlarm` is a normal function invocation that can be  
19 inlined by compilers in the usual fashion:

```
20 setSoundL(Sound::Siren); // body of setAlarm may
21 // well be inlined here
```

22 The call to `std::bind`, however, passes a function pointer to `setAlarm`, and that  
23 means that inside the function call operator for `setSoundB` (i.e., the function call  
24 operator for the bind object), the call to `setAlarm` takes place through a function  
25 pointer. Compilers are less likely to inline function calls through function pointers,  
26 and that means that calls to `setAlarm` through `setSoundB` are less likely to be  
27 fully inlined than those through `setSoundL`:

```
28 setSoundB(Sound::Siren); // body of setAlarm is less
29 // likely to be inlined here
```

30 It's thus possible 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-  
33 ple, consider this C++14 lambda, which returns whether its argument is between a

1 minimum value (`lowVal`) and a maximum value (`highVal`), where `lowVal` and  
2 `highVal` are local variables:

```
3 auto betweenL =
4     [lowVal, highVal]
5     (const auto& val) // C++14
6     { return lowVal <= val && val <= highVal; };
```

7 `std::bind` can express the same thing, but the construct is an example of job se-  
8 curity through code obscurity:

```
9 using namespace std::placeholders; // as above
10 auto betweenB =
11     std::bind(std::logical_and<>(),
12                 std::bind(std::less_equal<>(), lowVal, _1),
13                 std::bind(std::less_equal<>(), _1, highVal));
```

14 In C++11, we'd have to specify the types we wanted to compare, and the  
15 `std::bind` call would then look like this:

```
16 auto betweenB = // C++11 version
17     std::bind(std::logical_and<bool>(),
18                 std::bind(std::less_equal<int>(), lowVal, _1),
19                 std::bind(std::less_equal<int>(), _1, highVal));
```

20 Of course, in C++11, the lambda couldn't take an `auto` parameter, so it'd have to  
21 commit to a type, too:

```
22 auto betweenL = // C++11 version
23     [lowVal, highVal]
24     (int val)
25     { return lowVal <= val && val <= highVal; };
```

26 Either way, I hope we can agree that the lambda version is not just shorter, but  
27 also more comprehensible and maintainable.

28 Earlier, I remarked that for those with little `std::bind` experience, its placehold-  
29 ers (e.g., `_1`, `_2`, etc.) are essentially magic. But it's not just the behavior of the  
30 placeholders that's opaque. Suppose we have a function to create compressed cop-  
31 ies of `Widgets`,

```
32 enum class Complevel { Low, Normal, High }; // compression
33 // level
```

```
1 Widget compress(const Widget& w,           // make compressed
2                     CompLevel lev);           // copy of w
```

3 and we want to create a function object that allows us to specify how much a par-  
4 ticular `Widget w` should be compressed. This use of `std::bind` will create such an  
5 object:

```
6 Widget w;
7 using namespace std::placeholders;
8 auto compressRateB = std::bind(compress, w, _1);
```

9 Now, when we pass `w` to `std::bind`, it has to be stored for the later call to `com-`  
10 `press`. It's stored inside the object `compressRateB`, but how is it stored—by value  
11 or by reference? It makes a difference, because if `w` is modified between the call to  
12 `std::bind` and a call to `compressRateB`, storing `w` by reference will reflect the  
13 changes, while storing it by value won't.

14 The answer is that it's stored by value, but the only way to know that is to memo-  
15 rize how `std::bind` works; there's no sign of it in the call to `std::bind`.<sup>†</sup> Con-  
16 trast that with a lambda approach, where whether `w` is captured by value or by ref-  
17 erence is explicit:

```
18 auto compressRateL =                               // w is captured by
19   [w](CompLevel lev)                           // value; lev is
20   { return compress(w, lev); };                // passed by value
```

21 Equally explicit is how parameters are passed to the lambda. Here, it's clear that  
22 the parameter `lev` is passed by value. Hence:

```
23 compressRateL(CompLevel::High);                 // arg is passed
24   // by value
```

---

<sup>†</sup> `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.

1 But in the call to the object resulting from `std::bind`, how is the argument  
2 passed?

3 `compressRateB(CompLevel::High);` // how is arg  
4 // passed?

5 Again, the only way to know is to memorize how `std::bind` works. (The answer  
6 is that all arguments passed to objects created by `std::bind` are passed by refer-  
7 ence, because the function call operator for such objects uses perfect forwarding.)

8 Compared to lambdas, then, code using `std::bind` is less readable, less expres-  
9 sive, and possibly less efficient. In C++14, there are no reasonable use cases for  
10 `std::bind`. In C++11, however, `std::bind` can be justified in two constrained  
11 situations:

12 • **Move capture.** C++11 lambdas don't offer move capture, but it can be emulat-  
13 ed through a combination of a lambda and `std::bind`. For details, consult  
14 Item 32, which also explains that in C++14, lambdas' support for init capture  
15 eliminates the need for the emulation.

16 • **Polymorphic function objects.** Because the function call operator on a bind  
17 object uses perfect forwarding, it can accept arguments of any type (modulo  
18 the restrictions on perfect forwarding described in Item 30). This can be useful  
19 when you want to bind an object with a templated function call operator. For  
20 example, given this class,

```
21 class PolyWidget {  
22 public:  
23     template<typename T>  
24     void operator()(const T& param);  
25     ...  
26 };
```

27 `std::bind` can bind a `PolyWidget` as follows:

```
28 PolyWidget pw;  
29 auto boundPW = std::bind(pw, _1);
```

30 `boundPW` can then be called with different types of arguments:

```
1 boundPW(1930);           // pass int to
2   // PolyWidget::operator()
3 boundPW(nullptr);          // pass nullptr to
4   // PolyWidget::operator()
5 boundPW("Rosebud");        // pass string literal to
6   // PolyWidget::operator()
```

7 There is no way to do this with a C++11 lambda. In C++14, however, it's easily  
8 achieved via a lambda with an `auto` parameter:

```
9 auto boundPW = [pw](const auto& param)    // C++14
10 { pw(param); };
```

11 These are edge cases, of course, and they're transient edge cases at that, because  
12 compilers supporting C++14 lambdas are increasingly common.

13 When `bind` was unofficially added to C++ in 2005, it was a big improvement over  
14 its 1998 predecessors. The addition of lambda support to C++11 rendered  
15 `std::bind` all but obsolete, however, and as of C++14, there are just no good use  
16 cases for it.

## 17 Things to Remember

- 18 • Lambdas are more readable, more expressive, and may be more efficient than  
19 using `std::bind`.
- 20 • In C++11 only, `std::bind` may be useful for implementing move capture or  
21 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 issues, 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 35: 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 function object passed to std::async (e.g., doAsyncWork) is  
6 considered 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).  
16 A more fundamental difference between thread-based and task-based program-  
17 ming is the higher level of abstraction that task-based embodies. It frees you from  
18 the details of thread management, an observation that reminds me that I need to  
19 summarize the three meanings of "thread" in concurrent C++ software:  
20 • Hardware threads are the threads that actually perform computation. Contem-  
21 porary machine architectures offer one or more hardware threads per CPU  
22 core.  
23 • Software threads (also known as OS threads or system threads) are the threads  
24 that the operating system† manages across all processes and schedules for ex-  
25 ecution on hardware threads. It's typically possible to create more software  
26 threads than hardware threads, because when a software thread is blocked
```

---

<sup>†</sup> Assuming you have one. Some embedded systems don't.

1 (e.g., on I/O or waiting for a mutex or condition variable), throughput can be  
2 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 14 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 unbal-  
20 anced loads and, if the current thread is a GUI thread, responsiveness issues. An-  
21 other option is to wait for some existing software threads to complete and then try  
22 to create a new `std::thread` again, but it’s possible that the existing threads are  
23 waiting for an action that `doAsyncWork` is supposed to perform (e.g., produce a  
24 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 (i.e., they contain little data and few instructions useful to it) and (2) the  
4 running of the “new” software thread on that core “pollutes” the CPU caches for  
5 “old” threads that had been running on that core and are likely to be scheduled to  
6 run there again.

7 Avoiding oversubscription is difficult, because the optimal ratio of software to  
8 hardware threads depends on how often the software threads are runnable, and  
9 that can change dynamically, e.g., when a program goes from an I/O-heavy region  
10 to a computation-heavy region. The best ratio of software to hardware threads is  
11 also dependent on the cost of context switches and how effectively the software  
12 threads use the CPU caches. Furthermore, the number of hardware threads and  
13 the details of the CPU caches (e.g., how large they are and their relative speeds)  
14 depend on the machine architecture, so even if you tune your application to avoid  
15 oversubscription (while still keeping the hardware busy) on one platform, there’s  
16 no guarantee that your solution will work well on other kinds of machines.

17 Your life will be easier if you dump these problems on somebody else, and using  
18 `std::async` does exactly that.

```
19 auto fut = std::async(doAsyncWork); // onus of thread mgmt is
20 // on implementer of
21 // the Standard Library
```

22 This call shifts the thread management responsibility to the implementer of the  
23 C++ Standard Library. For example, the likelihood of receiving an out-of-threads  
24 exception is significantly reduced, because this call will probably never yield one.  
25 “How can that be?”, you might wonder. “If I ask for more software threads than the  
26 system can provide, why does it matter whether I do it by creating `std::threads`  
27 or by calling `std::async`?“ It matters, because `std::async`, when called in this  
28 form (i.e., with the default launch policy—see Item 36), doesn’t guarantee that it  
29 will create a new software thread. Rather, it permits the scheduler to arrange for  
30 the specified function (in this example, `doAsyncWork`) to be run on the thread re-  
questing `doAsyncWork`’s result (i.e., the thread calling `get` or `wait` on `fut`), and

1 reasonable schedulers take advantage of that freedom if the system is oversub-  
2 scribed or is out of threads.

3 If you pulled this “run it on the thread needing the result” trick yourself, I re-  
4 marked that it could lead to load-balancing issues, and those issues don’t go away  
5 simply because it’s `std::async` and the run time scheduler that confront them  
6 instead of you. When it comes to load-balancing, however, the run time scheduler  
7 is likely to have a more comprehensive picture of what’s happening on the ma-  
8 chine than you do, because it manages the threads from all processes, not just the  
9 one your code is running in.

10 With `std::async`, responsiveness on a GUI thread can still be problematic, be-  
11 cause the scheduler has no way of knowing which of your threads has tight re-  
12 sponsiveness requirements. In that case, you’ll want to pass the  
13 `std::launch::async` launch policy to `std::async`. That will ensure that the  
14 function you want to run really executes on a different thread (see Item 36).

15 State-of-the-art thread schedulers employ system-wide thread pools to avoid  
16 oversubscription, and they improve load balancing across hardware cores through  
17 work-stealing algorithms. The C++ Standard does not require the use of thread  
18 pools or work-stealing, and, to be honest, there are some technical aspects of the  
19 C++11 concurrency specification that make it more difficult to employ them than  
20 we’d like. Nevertheless, some vendors take advantage of this technology in their  
21 Standard Library implementation, and it’s reasonable to expect that progress will  
22 continue in this area. If you take a task-based approach to your concurrent pro-  
23 gramming, you automatically reap the benefits of such technology as it becomes  
24 more widespread. If, on the other hand, you program directly with `std::threads`,  
25 you assume the burden of dealing with thread exhaustion, oversubscription, and  
26 load balancing yourself, not to mention how your solutions to these problems  
27 mesh with the solutions implemented in programs running in other processes on  
28 the same machine.

29 Compared to thread-based programming, a task-based design spares you the trav-  
30 ails of manual thread management, and it provides a natural way to examine the  
31 results of asynchronously executed functions (i.e., return values or exceptions).

1 Nevertheless, there are some situations where using threads directly may be ap-  
2 propiate. They include:

3 • **You need access to the API of the underlying threading implementation.**

4 The C++ concurrency API is typically implemented using a lower-level plat-  
5 form-specific API, usually pthreads or Windows' Threads. Those APIs are cur-  
6 rently richer than what C++ offers. (For example, C++ has no notion of thread  
7 priorities or affinities.) To provide access to the API of the underlying thread-  
8 ing implementation, `std::thread` objects typically offer the `native_handle`  
9 member function. There is no counterpart to this functionality for  
10 `std::futures` (i.e., what `std::async` returns).

11 • **You need to and are able to optimize thread usage for your application.**

12 This could be the case, for example, if you're developing server software with a  
13 known execution profile that will be deployed as the only significant process  
14 on a machine with fixed hardware characteristics.

15 • **You need to implement threading technology beyond the C++ concurren-**  
16 **cy API**, e.g., thread pools on platforms where your C++ implementations don't  
17 offer them.

18 These are uncommon cases, however. Most of the time, you should choose task-  
19 based designs instead of programming with threads.

20 **Things to Remember**

- 21 • The `std::thread` API offers no direct way to get return values from asyn-  
22 chronously-run functions, and if those functions throw, the program is termi-  
23 nated.
- 24 • Thread-based programming calls for manual management of thread exhaus-  
25 tion, oversubscription, load balancing, and adaptation to new platforms.
- 26 • Task-based programming via `std::async` with the default launch policy han-  
27 dles most of these issues for you.

1    **Item 36: Specify `std::launch::async` if asynchronicity is  
2        essential.**

3    When you call `std::async` to execute a function (or other callable object), you're  
4        generally intending to run the function asynchronously. But that's not necessarily  
5        what you're asking `std::async` to do. You're really requesting that the function  
6        be run in accord with a `std::async` *launch policy*. There are two standard poli-  
7        cies, each represented by an enumerator in the `std::launch` scoped enum. (See  
8        Item 10 for information on scoped enums.) Assuming a function `f` is passed to  
9        `std::async` for execution,

- 10    • The `std::launch::async` launch policy means that `f` must be run asynchro-  
11        nously, i.e., on a different thread.

- 12    • The `std::launch::deferred` launch policy means that `f` may run only when  
13        `get` or `wait` is called on the future returned by `std::async`.† That is, `f`'s exe-  
14        cution is *deferred* until such a call is made. When `get` or `wait` is invoked, `f` will  
15        execute synchronously, i.e., on the thread that made the invocation. (The caller  
16        will block until `f` finishes running.) If neither `get` nor `wait` is called, `f` will  
17        never run.

18    Perhaps surprisingly, `std::async`'s default launch policy—the one it uses if you  
19        don't expressly specify one—is neither of these. Rather, it's these or-ed together.

20    The following two calls have exactly the same meaning:

---

† This is not really 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 38 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 auto fut1 = std::async(f);           // run f using
2   // default launch
3   // policy
4 auto fut2 = std::async(std::launch::async |      // run f either
5                         std::launch::deferred, // async or
6                         f);                // deferred
```

7 The default policy thus permits `f` to be run either asynchronously or synchronously (on the thread calling `get` or `wait` on the future returned from `std::async`). As  
8 Item 35 points out, this flexibility permits `std::async` and the thread-  
9 management components of the Standard Library to assume responsibility for  
10 thread creation and destruction, avoidance of oversubscription, and load balanc-  
11 ing. That's among the things that make concurrent programming with  
12 `std::async` so convenient.

14 But using `std::async` with the default launch policy has some interesting impli-  
15 cations. Given a thread `t` executing this statement,

```
16 auto fut = std::async(f); // run f using default launch policy
```

- **It's not possible to predict whether `f` will run concurrently with `t`,** be-  
cause `f` might be scheduled to run deferred.
- **It's not possible to predict whether `f` runs on a thread different from the**  
**thread invoking `get` or `wait` on `fut`.** If that thread is `t`, the implication is that  
it's not possible to predict whether `f` runs on a thread different from `t`.
- **It may not be possible to predict whether `f` runs at all**, because it may not  
be possible to guarantee that `get` or `wait` will be called on `fut` along every  
path through the program.

25 The default launch policy's scheduling flexibility often mixes poorly with the use of  
26 `thread_local` variables, because it means that if `f` reads or writes such *thread-*  
27 *local storage* (TLS), it's not possible to predict which thread's variables will be ac-  
28 cessed:

```
29 auto fut = std::async(f);           // TLS for f possibly for
30   // different thread, but
31   // possibly for thread
32   // invoking get or wait on fut
```

1 It also affects `wait-based` loops using timeouts, because calling `wait_for` or  
2 `wait_until` on a task (see Item 35) that's deferred yields the value  
3 `std::launch::deferred`. This means that the following loop, which looks like it  
4 should eventually terminate, may, in reality, run forever:

```
5 using namespace std::literals;           // for C++14 duration
6   // suffixes; see Item 34
7 void f()                                // f sleeps for 1 second,
8 {   // then returns
9     std::this_thread::sleep_for(1s);
10 }
11 auto fut = std::async(f);                // run f asynchronously
12   // (conceptually)
13 while (fut.wait_for(100ms) !=          // loop until f has
14         std::future_status::ready)       // finished running...
15 {   // which may never happen!
16     ...
17 }
```

18 If `f` runs concurrently with the thread calling `std::async` (i.e., if the launch policy  
19 chosen for `f` is `std::launch::async`), there's no problem here, but if `f` is de-  
20 ferred, `fut.wait_for` will always return `std::future_status::deferred`.  
21 That will never be equal to `std::future_status::ready`, so the loop will never  
22 terminate.

23 This kind of bug is easy to overlook during development and unit testing, because  
24 it may manifest itself only under heavy loads. Those are the conditions that push  
25 the machine towards oversubscription or thread exhaustion, and that's when a  
26 task may be most likely to be deferred. After all, if the hardware isn't threatened by  
27 oversubscription or thread exhaustion, there's no reason for the runtime system  
28 not to schedule the task for concurrent execution.

29 The fix is simple: just check the future corresponding to the `std::async` call to  
30 see whether the task is deferred, and, if so, avoid entering the timeout-based loop.  
31 Unfortunately, there's no direct way to ask a future whether its task is deferred.  
32 Instead, you have to call a timeout-based function—a function such as `wait_for`.  
33 Of course, you don't really want to wait for anything, you just want to see if the re-

1 turn value is `std::future_status::deferred`, so stifle your mild disbelief at  
2 the necessary circumlocution and call `wait_for` with a zero timeout:

```
3 auto fut = std::async(f); // as above
4 if (fut.wait_for(0s) == // if task is
5     std::future_status::deferred) // deferred...
6 {
7     ... // ...use wait or get on fut
8     ... // to call f synchronously
9
10 } else { // task isn't deferred
11     while (fut.wait_for(100ms) != // infinite loop
12             std::future_status::ready) { // not possible
13         ... // task is neither deferred nor ready,
14         ... // so do concurrent work until it's ready
15     }
16     ... // fut is ready
17 }
```

18 The upshot of these various considerations is that using `std::async` with the de-  
19 fault launch policy for a task is fine as long as the following conditions are fulfilled:

- 20 • The task need not run concurrently with the thread calling `get` or `wait`.
- 21 • It doesn't matter which thread's `thread_local` variables are read or written.
- 22 • Either there's a guarantee that `get` or `wait` will be called on the future re-  
23 turned by `std::async` or it's acceptable that the task may never execute.
- 24 • Code using `wait_for` or `wait_until` takes the possibility of deferred status  
25 into account.

26 If any of these conditions fails to hold, you probably want to guarantee that  
27 `std::async` will schedule the task for truly asynchronous execution. The way to  
28 do that is to pass `std::launch::async` as the first argument when you make the  
29 call:

```
30 auto fut = std::async(std::launch::async, f); // launch f
31 // asynchronously
```

1 In fact, having a function that acts like `std::async`, but that automatically uses  
2 `std::launch::async` as the launch policy, is a convenient tool to have around, so  
3 it's nice that it's easy to write. Here's the C++11 version:

```
4 template<typename F, typename... Ts>
5 inline
6 std::future<typename std::result_of<F(Ts...)>::type>
7 reallyAsync(F&& f, Ts&&... params)           // return future
8 {   // for asynchronous
9     return std::async(std::launch::async,          // call to f(params...)
10                    std::forward<F>(f),
11                    std::forward<Ts>(params)...);
12 }
```

13 This function takes an callable object `f` and zero or more arguments `args` and per-  
14 fect-forwards them (see Item 25) to `std::async`, passing `std::launch::async`  
15 as the launch policy. Like `std::async`, it returns a `std::future` for the result of  
16 invoking `f` on `args`. Determining the type of that result is easy, because the type  
17 trait `std::result_of` gives it to you. (See Item 9 for general information on type  
18 traits.)

19 `reallyAsync` is used just like `std::async`:

```
20 auto fut = reallyAsync(f);                  // run f asynchronously;
21   // throw if std::async
22   // would throw
```

23 In C++14, the ability to deduce `reallyAsync`'s return type streamlines the func-  
24 tion declaration:

```
25 template<typename F, typename... Ts>
26 inline
27 auto
28 reallyAsync(F&& f, Ts&&... params)           // C++14
29 {
30     return std::async(std::launch::async,
31                     std::forward<F>(f),
32                     std::forward<Ts>(params)...);
33 }
```

34 This version makes it even clearer that `reallyAsync` does nothing but invoke  
35 `std::async` with the `std::launch::async` launch policy.

1    **Things to Remember**

- 2    ♦ The default launch policy for `std::async` permits both asynchronous and  
3    synchronous task execution.
- 4    ♦ This flexibility leads to uncertainty when accessing `thread_locals`, implies  
5    that the task may never execute, and affects program logic for timeout-based  
6    `wait` calls.
- 7    ♦ Specify `std::launch::async` if asynchronous task execution is essential.

8    **Item 37: Make `std::threads` unjoinable on all paths.**

9    Every `std::thread` object is in one of two states: *joinable* or *unjoinable*. A joinable  
10   `std::thread` corresponds to an underlying asynchronous thread of execution  
11   that is or could be running. A `std::thread` corresponding to an underlying  
12   thread that's blocked or waiting to be scheduled is joinable, for example.  
13   `std::thread` objects corresponding to underlying threads that have run to com-  
14   pletion are also considered joinable.

15   An unjoinable `std::thread` is what you'd expect: a `std::thread` that's not join-  
16   able. Unjoinable `std::thread` objects include:

- 17   • **Default-constructed `std::threads`.** Such `std::threads` have no function to  
18   execute, hence don't correspond to an underlying thread of execution.
- 19   • **`std::thread` objects that have been moved from.** The result of a move is  
20   that the underlying thread of execution a `std::thread` used to correspond to  
21   (if any) now corresponds to a different `std::thread`.
- 22   • **`std::threads` that have been joined.** After a join, the `std::thread` object  
23   no longer corresponds to the underlying thread of execution that has finished  
24   running.
- 25   • **`std::threads` that have been detached.** A `detach` severs the connection  
26   between a `std::thread` object and the underlying thread of execution it cor-  
27   responds to.

1 One reason a `std::thread`'s joinability is important is that if the destructor for a  
2 joinable thread is invoked, execution of the program is terminated. For example,  
3 suppose we have a function `doWork` that takes a filtering function, `filter`, and a  
4 maximum value, `maxVal`, as parameters. `doWork` checks to make sure that all con-  
5 ditions necessary for its computation are satisfied, then performs the computation  
6 with all the values between 0 and `maxVal` that pass the filter. If it's time-  
7 consuming to do the filtering and it's also time-consuming to determine whether  
8 `doWork`'s conditions are satisfied, it would be reasonable to do those two things  
9 concurrently.

10 Our preference would be to employ a task-based design for this (see Item 35), but  
11 let's assume we'd like to set the priority of the thread doing the filtering. Item 35  
12 explains that that requires use of the thread's native handle, and that's accessible  
13 only through the `std::thread` API; the task-based API (i.e., futures) doesn't pro-  
14 vide it. Our approach will therefore be based on threads, not tasks.

15 We could come up with code like this:

```
16 constexpr auto tenMillion = 10000000;           // see Item 15
17   // for constexpr
18 bool doWork(std::function<bool(int)> filter,    // returns whether
19             int maxVal = tenMillion)            // computation was
20 {   // performed
21     std::vector<int> vals;                      // values that
22   // satisfy filter
23     std::thread t([&filter, maxVal, &vals]        // compute vals'
24                 {                                // content
25                     for (auto i = 0; i <= maxVal; ++i)
26                         { if (filter(i)) vals.push_back(i); }
27                 });
28     auto nh = t.native_handle();                  // use t's native
29     ...   // handle to set
30   // t's priority
31     if (conditionsAreSatisfied()) {
32         t.join();                            // let t finish
33         performComputation(vals);
34         return true;                        // computation was
35     }   // performed
```

```
1     return false;                                // computation was
2 };  // not performed
```

3 Before I explain why this code is problematic, I'll remark that `tenMillion`'s initializing value can be made more readable in C++14 by taking advantage of  
4 C++14's ability to use an apostrophe as a digit separator:

```
6 constexpr auto tenMillion = 10'000'000;        // C++14
```

7 I'll also remark that setting `t`'s priority after it has started running is a bit like closing the proverbial barn door after the equally proverbial horse has bolted. A better  
8 design would be to start `t` in a suspended state (thus making it possible to adjust  
9 its priority before it does any computation), but I don't want to distract you with  
10 that code. If you're more distracted by the code's absence, turn to Item 39, because  
11 it shows how to start threads suspended.

13 But back to `doWork`. If `conditionsAreSatisfied()` returns `true`, all is well, but  
14 if it returns `false` or throws an exception, the `std::thread` object `t` will be joinable  
15 when its destructor is called at the end of `doWork`. That would cause program  
16 execution to be terminated.

17 You might wonder why the `std::thread` destructor behaves this way. It's because  
18 the two other obvious options are arguably worse. They are:

19 • **An implicit join.** In this case, a `std::thread`'s destructor would wait for its  
20 underlying asynchronous thread of execution to complete. That sounds reasonable,  
21 but it could lead to performance anomalies that would be difficult to track down.  
22 For example, it would be counterintuitive that `doWork` would wait  
23 for its filter to be applied to all values if `conditionsAreSatisfied()` had already  
24 returned `false`.

25 • **An implicit detach.** In this case, a `std::thread`'s destructor would sever the  
26 connection between the `std::thread` object and its underlying thread of execution.  
27 The underlying thread would continue to run. This sounds no less reasonable  
28 than the `join` approach, but the debugging problems it can lead to are worse.  
29 In `doWork`, for example, `vals` is a local variable that is captured by reference.  
30 It's also modified inside the lambda (via the call to `push_back`). Sup-

1 pose, then, that while the lambda is running asynchronously, *conditionsAreSatisfied()* returns `false`. In that case, `doWork` would return,  
2 and its local variables (including `vals`) would be destroyed. Its stack frame  
3 would be popped, and execution of its thread would continue at `doWork`'s call  
4 site.

5 Statements following that call site would, at some point, make additional func-  
6 tion calls, and at least one such call would probably end up using some or all of  
7 the memory that had once been occupied by the `doWork` stack frame. Let's call  
8 such a function `f`. While `f` was running, the lambda that `doWork` initiated  
9 would still be running asynchronously. That lambda could call `push_back` on  
10 the stack memory that used to be `vals` but that is now somewhere inside `f`'s  
11 stack frame. Such a call would modify the memory that used to be `vals`, and  
12 that means that from `f`'s perspective, the content of memory in its stack frame  
13 could spontaneously change! Imagine the fun you'd have debugging *that*.

14 The Standardization Committee decided that the consequences of destroying a  
15 joinable thread were sufficiently dire that they essentially banned it (by specifying  
16 that destruction of a joinable thread causes program termination).

17 This puts the onus on you to ensure that if you use a `std::thread` object, it's  
18 made unjoinable on every path out of the scope in which it's defined. But covering  
19 every path can be complicated. It includes flowing off the end of the scope as well  
20 as jumping out via a `return`, `continue`, `break`, `goto` or exception. That can be a  
21 lot of paths.

22 Any time you want to perform some action along every path out of a scope, the  
23 normal approach is to put that action in the destructor of a local object. Such ob-  
24 jects are known as *RAII objects*, and the classes they come from are known as *RAII*  
25 *classes*. (*RAII* itself stands for “Resource Acquisition Is Initialization,” although the  
26 crux of the technique is destruction, not initialization). RAII classes are common in  
27 the Standard Library. Examples include the STL containers (each container's de-  
28 structor destroys the container's contents and releases its memory), the standard  
29 smart pointers (Items 18-20 explain that `std::unique_ptr`'s destructor invokes  
30 its deleter on the object it points to, and the destructors in `std::shared_ptr` and

1    `std::weak_ptr` decrement reference counts), `std::fstream` objects (their de-  
2    structors close the files they correspond to), and many more. And yet there is no  
3    standard RAII class for `std::thread` objects, perhaps because the Standardiza-  
4    tion Committee, having rejected both `join` and `detach` as default options, simply  
5    didn't know what such a class should do.

6    Fortunately, it's not difficult to write one yourself. For example, the following class  
7    allows callers to specify whether `join` or `detach` should be called when a  
8    `ThreadRAII` object (an RAII object for a `std::thread`) is destroyed:

```
9  class ThreadRAII {
10 public:
11     enum class DtorAction { join, detach }; // Item 10 has info
12   // on enum class
13     ThreadRAII(std::thread&& t, DtorAction a) // in dtor, take
14     : action(a), t(std::move(t)) {}           // action a on t
15
16     ~ThreadRAII()                           // see below for
17     {                                     // joinability test
18         if (t.joinable()) {
19             if (action == DtorAction::join) {
20                 t.join();
21             } else {
22                 t.detach();
23             }
24         }
25     std::thread& get() { return t; }           // see below
26
27 private:
28     DtorAction action;
29     std::thread t;
30 };
```

30   I hope this code is largely self-explanatory, but the following points may be help-  
31   ful:

- 32   • The constructor accepts only `std::thread` rvalues, because we want to move  
33   the passed-in `std::thread` into the `ThreadRAII` object.

- 1     • The parameter order in the constructor is designed to be intuitive to callers  
2       (specifying the `std::thread` first and the destructor action second makes  
3       more sense than vice-versa), but the member initialization list is designed to  
4       match the order of the data members' declarations. That order puts the  
5       `std::thread` object last. In this class, the order makes no difference, but in  
6       general, it's possible for the initialization of one data member to depend on  
7       another, and because `std::thread` objects may start running a function im-  
8       mediately after they are initialized, it's a good habit to declare them last in a  
9       class. That guarantees that at the time they are constructed, all the data mem-  
10      bers that precede them have already been initialized and can therefore be safe-  
11      ly accessed by the asynchronously running thread that corresponds to the  
12      `std::thread` data member.
- 13     • `ThreadRAII` offers a `get` function to provide access to the underlying  
14       `std::thread` object. This is analogous to the `get` functions offered by the  
15       standard smart pointer classes that give access to their underlying raw point-  
16       ers. Providing `get` avoids the need for `ThreadRAII` to replicate the full  
17       `std::thread` interface, and it also means that `ThreadRAII` objects can be  
18       used in contexts where `std::thread` objects are required, e.g., because an in-  
19       terface requires that a reference to a `std::thread` be passed.
- 20     • Before the `ThreadRAII` destructor invokes a member function on the  
21       `std::thread` object `t`, it checks to make sure that `t` is joinable. This is neces-  
22       sary, because invoking `join` or `detach` on an unjoinable thread yields unde-  
23       fined behavior. It's possible that a client constructed a `std::thread`, created a  
24       `ThreadRAII` object from it, used `get` to acquire access to `t`, and then did a  
25       move from `t` or called `join` or `detach` on it. Each of those actions would ren-  
26       der `t` unjoinable.

27     If you're worried that in this code,

```
28     if (t.joinable()) {  
29       if (action == DtorAction::join) {  
30         t.join();  
31       } else {  
32         t.detach();
```

```
1      }
2
3      a race condition exists, because between execution of t.joinable() and in-
4      vocation of join or detach, another thread could render t unjoinable, your
5      intuition is commendable, but your fears are unfounded. A std::thread ob-
6      ject can change state from joinable to unjoinable only through a member func-
7      tion call, e.g., join, detach, or a move operation. At the time a ThreadRAII
8      object's destructor is invoked, no other thread should be making member func-
9      tion calls on that object. If there are simultaneous calls, there is certainly a race
10     condition, but the race isn't inside the destructor, it's in the client code that is
11     trying to invoke two member functions (the destructor and something else) on
12     one object at the same time. In general, simultaneous member function calls on
13     a single object are safe only if all are to const member functions (see Item 16).
```

14 Employing ThreadRAII in our doWork example would look like this:

```
15 bool doWork(std::function<bool(int)> filter, // as before
16             int maxVal = tenMillion)
17 {
18     std::vector<int> vals; // as before
19
20     ThreadRAII t( // use RAI object
21         std::thread([&filter, maxVal, &vals]
22                     {
23                         for (auto i = 0; i <= maxVal; ++i)
24                             { if (filter(i)) vals.push_back(i); }
25                     }),
26                     ThreadRAII::DtorAction::join // RAI action
27     );
28
29     auto nh = t.get().native_handle();
30     ...
31
32     if (conditionsAreSatisfied()) {
33         t.get().join();
34         performComputation(vals);
35         return true;
36     }
37
38     return false;
39 }
```

36 In this case, we've chosen to do a join on the asynchronously running thread in  
37 the ThreadRAII destructor, because, as we saw earlier, doing a detach could lead

1 to some truly nightmarish debugging. We also saw earlier that doing a `join` could  
2 lead to performance anomalies (that, to be frank, could also be unpleasant to de-  
3 bug), but given a choice between undefined behavior (which `detach` would get  
4 us), program termination (which use of a raw `std::thread` would yield), or per-  
5 formance anomalies, performance anomalies seems like the best of a bad lot.

6 Alas, Item 39 demonstrates that using `ThreadRAII` to perform a `join` on  
7 `std::thread` destruction can sometimes lead not just to a performance anomaly,  
8 but to a hung program. The “proper” solution to these kinds of problems would be  
9 to communicate to the asynchronously running lambda that we no longer need its  
10 work and that it should return early, but there’s no support in C++11 for *inter-*  
11 *ruptible threads*. They can be implemented by hand, but that’s a topic beyond the  
12 scope of this book.<sup>†</sup>

13 Item 17 explains that because `ThreadRAII` declares a destructor, there will be no  
14 compiler-generated move operations, but there is no reason `ThreadRAII` objects  
15 shouldn’t be movable. If compilers were to generate these functions, the functions  
16 would do the right thing, so explicitly requesting their creation is appropriate:

```
17 class ThreadRAII {
18 public:
19     enum class DtorAction { join, detach };           // as before
20     ThreadRAII(std::thread&& t, DtorAction a)        // as before
21     : action(a), t(std::move(t)) {}
22
23     ~ThreadRAII()                                     // as before
24     {
25         ...
26     }
27     ThreadRAII(ThreadRAII&&) = default;           // support
28     ThreadRAII& operator=(ThreadRAII&&) = default; // moving
29     std::thread& get() { return t; }                   // as before
```

---

<sup>†</sup> A nice treatment of this topic is in Anthony Williams’ *C++ Concurrency in Action* (Manning Publications, 2012), section 9.2.

```
1 private: // as before
2     DtorAction action;
3     std::thread t;
4 }
```

## 5 Things to Remember

- 6 • Make `std::threads` unjoinable on all paths.
- 7 • `join-on-destruction` can lead to difficult-to-debug performance anomalies.
- 8 • `detach-on-destruction` can lead to difficult-to-debug undefined behavior.

## 9 Item 38: Be aware of varying thread handle destructor behavior.

10 Item 37 explains that a joinable `std::thread` corresponds to an underlying system thread of execution. A future for a non-deferred task (see Item 36) 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.

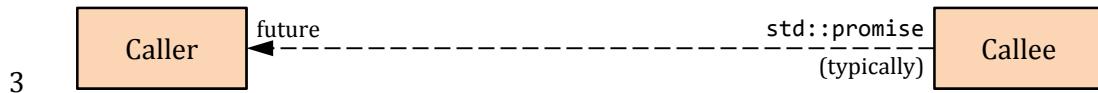
11 From this perspective, it's interesting that `std::threads` and futures have such different behaviors in their destructors. As noted in Item 37, destruction of a joinable `std::thread` terminates your program, because the two obvious alternatives—an implicit `join` and an implicit `detach`—were considered worse choices. Yet the destructor for a future sometimes behaves as if it did an implicit `join`, sometimes as if it did an implicit `detach`, and sometimes neither. It never causes program termination. This thread handle behavioral bouillabaisse deserves closer examination.

12 We'll begin with the observation that a future is one end of a communications channel through which a callee transmits a result to a caller.<sup>†</sup> The callee (usually running asynchronously) writes the result of its computation into the communications channel (typically via a `std::promise` object), and the caller reads that re-

---

<sup>†</sup> Item 39 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   sult using a future. You can think of it as follows, where the dashed arrow shows  
2   the flow of information from callee to caller:



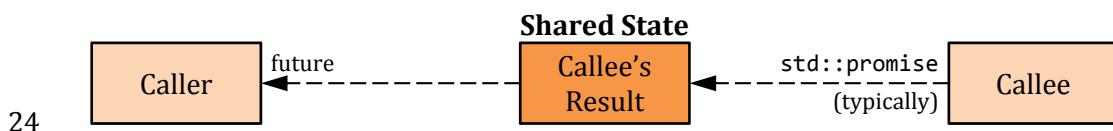
3   

4   But where is the callee's result stored? The callee could finish before the caller in-  
5   vokes `get` on a corresponding future, so the result can't be stored in the callee's  
6   `std::promise`. That object, being local to the callee, would be destroyed when the  
7   callee finished.

8   The result can't be stored in the caller's future, either, because (among other rea-  
9   sons) a `std::future` may be used to create a `std::shared_future` (thus trans-  
10   ferring ownership of the callee's result from the `std::future` to the  
11   `std::shared_future`), which may then be copied many times after the original  
12   `std::future` is destroyed. Given that not all result types can be copied (e.g.,  
13   `std::unique_ptr`—see Item 18) and that the result must live at least as long as  
14   the last future referring to it, which of the potentially many futures corresponding  
15   to the callee should be the one to contain its result?

16   Because neither objects associated with the callee nor objects associated with the  
17   caller are suitable places to store the callee's result, it's stored in a location outside  
18   both. This location is known as the *shared state*. The shared state is typically rep-  
19   resented by a heap-based object, but its type, interface, and implementation are  
20   not specified by the Standard. Standard Library authors are free to implement  
21   shared states in any way they like.

22   We can envision the relationship among the callee, the caller, and the shared state  
23   as follows, where dashed arrows once again represent the flow of information:



1 The existence of the shared state is important, because the behavior of a future's  
2 destructor—the topic of this Item—is determined by the shared state associated  
3 with the future. In particular,

- **The destructor for the last future referring to a shared state for a non-deferred task launched via `std::async` blocks** until the task completes. In essence, the destructor for such a future does an implicit `join` on the thread on which the asynchronously-executing task is running.

- **The destructor for all other futures simply destroys the future object.** For asynchronously running tasks, this is akin to an implicit `detach` on the underlying thread. For deferred tasks for which this is the final future, it means that the deferred task will never run.

These rules sound more complicated than they are. What we're really dealing with is a simple "normal" behavior and one lone exception to it. The normal behavior is that a future's destructor destroys the future object. That's it. It doesn't `join` with anything, it doesn't `detach` from anything, it doesn't run anything. It just destroys the future's data members. (Well, actually, it does one more thing. It also decrements the reference count inside the shared state that's manipulated by both the futures referring to it and the callee's `std::promise`. This reference count makes it possible for the library to know when the shared state can be destroyed. For information about reference counting, see Item 19.)

The exception to this normal behavior arises only for a future for which all of the following apply:

- **It refers to a shared state that was created due to a call to `std::async`.**
- **The task's launch policy is `std::launch::async`** (see Item 36), either because that was chosen by the runtime system or because it was specified in the call to `std::async`.
- **The future is the last future referring to the shared state.** For `std::futures`, this will always be the case. For `std::shared_futures`, if other `std::shared_futures` refer to the same shared state as the future be-

1       ing destroyed, the future being destroyed follows the normal behavior (i.e., it  
2       simply destroys its data members).

3       Only when all of these conditions are fulfilled does a future's destructor exhibit  
4       special behavior, and that behavior is to block until the asynchronously running  
5       task completes. Practically speaking, this amounts to an implicit `join` on the  
6       thread running the `std::async`-created task.

7       It's not uncommon to hear this exception to normal future destructor behavior  
8       summarized as "Futures from `std::async` block in their destructors." To a first  
9       approximation, that's correct, but sometimes you need more than a first approxi-  
10      mation. Now you know the truth in all its glory and wonder.

11      Your wonder may take a different form. It may be of the "I wonder why there's a  
12      special rule for shared states for non-deferred tasks that are launched by  
13      `std::async`" variety. It's a reasonable question. From what I can tell, the Stand-  
14      ardization Committee wanted to avoid the problems associated with an implicit  
15      `detach` (see Item 37), but they didn't want to adopt as radical a policy as manda-  
16      tory program termination (as they did for joinable `std::threads`—again, see  
17      Item 37), so they compromised on an implicit `join`. The decision was not without  
18      controversy, and there was serious talk about abandoning this behavior for C++14.  
19      In the end, however, no change was made, so the behavior of destructors for fu-  
20      tures is consistent in C++11 and C++14.

21      The API for futures offers no way to determine whether a future refers to a shared  
22      state arising from a call to `std::async`, so given an arbitrary future object, it's not  
23      possible to know whether it will block in its destructor waiting for an asynchro-  
24      nously running task to finish. This has some interesting implications:

```
25 // this container might block in its dtor, because one or more
26 // contained futures could refer to a shared state for a non-
27 // deferred task launched via std::async
28 std::vector<std::future<void>> futs; // see Item 39 for info
29 // on std::future<void>
30 class Widget { // Widget objects might
31 public: // block in their dtors
32 ...
```

```
1 private:  
2     std::shared_future<double> fut;  
3 };  
  
4 Of course, if you have a way of knowing that a given future does not satisfy the  
5 conditions that trigger the special destructor behavior (e.g., due to program logic),  
6 you're assured that that future won't block in its destructor. For example, only  
7 shared states arising from calls to std::async qualify for the special behavior, but  
8 there are other ways that shared states get created. One is the use of  
9 std::packaged_task. A std::packaged_task object prepares a function (or  
10 other callable object) for asynchronous execution by wrapping it such that its re-  
11 sult is put into a shared state. A future referring to that shared state can then be  
12 obtained via std::packaged_task's get_future function:
```

```
13 int calcValue();                      // func to run  
14 std::packaged_task<int()> pt(calcValue); // wrap calcValue so it  
15                                // can run asynchronously  
16 auto fut = pt.get_future();           // get future for pt
```

17 At this point, we know that the future `fut` doesn't refer to a shared state created  
18 by a call to `std::async`, so its destructor will behave normally.

19 Once created, the `std::packaged_task` `pt` can be run on a thread. (It could be  
20 run via a call to `std::async`, too, but if you want to run a task using `std::async`,  
21 there's little reason to create a `std::packaged_task`, because `std::async` does  
22 everything `std::packaged_task` does before it schedules the task for execution.)

23 `std::packaged_tasks` aren't copyable, so when `pt` is passed to the  
24 `std::thread` constructor, it must be cast to an rvalue (via `std::move`—see  
25 Item 23), thus ensuring that it will be moved into the data area associated with the  
26 thread, not copied:

```
27 std::thread t(std::move(pt));          // run pt on t
```

28 This example lends some insight into the normal behavior for future destructors,  
29 but it's easier to see if the statements are put together inside a scope:

```
30 {                                     // begin scope
```

```
1 std::packaged_task<int>()
2     pt(calcValue);
3 
4     auto fut = pt.get_future();
5 
6     std::thread t(std::move(pt));
7     ...
8         // see below
9 
10    }
11 
12    // end scope
13 
14    The most interesting code here is the “...” that follows creation of the
15    std::thread object t and precedes the end of the scope. What makes it interest-
16    ing is what can happen to t inside the “...” region. There are three basic possibili-
17    ties:
18 
19    • Nothing happens to t. In this case, t will be joinable at the end of the scope.
20    That will cause the program to be terminated (see Item 37).
21 
22    • A join is done on t. In this case, there would be no need for fut to block in its
23    destructor, because the join is already present in the calling code.
24 
25    • A detach is done on t. In this case, there would be no need for fut to detach
26    in its destructor, because the calling code already does that.
27 
28    In other words, when you have a future corresponding to a shared state that arose
29    due to a std::packaged_task, there's usually no need to adopt a special destruc-
30    tion policy, because the decision among termination, joining, or detaching will be
31    made in the code that manipulates the std::thread on which the
32    std::packaged_task is typically run.
```

## 22 Things to Remember

- 23 • Future destructors normally just destroy the future's data members.
- 24 • The final future referring to a shared state for a non-deferred task launched via  
25 std::async blocks until the task completes.

1   **Item 39: Consider void futures for one-shot event commu-**  
2   **nication.**

3   Sometimes it's useful for a task to tell a second (asynchronously-running) task that  
4   a particular event has occurred, because the second task can't proceed until the  
5   event has taken place. Perhaps a data structure has been initialized, a stage of  
6   computation has been completed, or a significant sensor value has been detected.  
7   When that's the case, what's the best way for this kind of inter-thread communica-  
8   tion to take place?

9   An obvious approach is to use a condition variable. If we call the task that detects  
10   the condition the *detecting task* and the task reacting to the condition the *reacting*  
11   *task*, the strategy is simple: the reacting task waits on a condition variable  
12   (*condvar*), and the detecting thread notifies that condvar when the event occurs.

13   Given

```
14 std::condition_variable cv;           // condvar for event  
15 std::mutex m;                      // mutex for use with cv
```

16   the code in the detecting task is as simple as simple can be:

```
17 ...                                // detect event  
18 cv.notify_one();                  // tell reacting task
```

19   If there were multiple reacting tasks to be notified, it would be appropriate to re-  
20   place *notify\_one* with *notify\_all*, but for now, we'll assume there's only one  
21   reacting task.

22   The code for the reacting task is a bit more complicated, because before calling  
23   *wait* on the condvar, it must lock a mutex through a *std::unique\_lock* object.  
24   (Locking a mutex before waiting on a condition variable is typical for threading  
25   libraries. The need to lock the mutex through a *std::unique\_lock* object is  
26   simply part of the C++11 API.) Here's the conceptual approach:

```
27 ...                                // prepare to react  
28 {                                    // open critical section  
29     std::unique_lock<std::mutex> lk(m); // lock mutex
```

```

1   cv.wait(1k);                                // wait for notify;
2   // this isn't correct!
3   ...                                     // react to event
4   // (m is locked)
5 }   // close crit. section;
6   // unlock m via lk's dtor
7 ...                                     // continue reacting
8   // (m now unlocked)

```

9 The first issue with this approach is what's sometimes termed a *code smell*: even if  
10 the code works, something doesn't seem quite right. In this case, the odor emanates from the need to use a mutex. Mutexes are used to control access to shared  
11 data, but it's entirely possible that the detecting and reacting tasks have no need  
12 for such mediation. For example, the detecting task might be responsible for initializing a global data structure, then turning it over to the reacting task for use. If  
13 the detecting task never accesses the data structure after initializing it, and if the  
14 reacting task never accesses it before the detecting task indicates that it's ready,  
15 the two tasks will stay out of each other's way through program logic. There will  
16 be no need for a mutex. The fact that the condvar approach requires one leaves  
17 behind the unsettling aroma of suspect design.

20 Even if you look past that, there are two other problems you should definitely pay  
21 attention to:

- 22 • **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.
- 27 • **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 truly occurred, and it does this as its first action after waking. The C++ condvar API makes this exception-

1 ally easy, because it permits a lambda (or other function object) that tests for  
2 the waited-for condition to be passed to `wait`. That is, the `wait` call in the re-  
3 acting task could be written like this:

```
4 cv.wait(1k,  
5     []{ return whether the event has occurred; });
```

6 Taking advantage of this capability requires that the reacting task be able to  
7 determine whether the condition it's waiting for is true. But in the scenario  
8 we've been considering, the condition it's waiting for is the occurrence of an  
9 event that the detecting thread is responsible for recognizing. The reacting  
10 thread may have no way of determining whether the event it's waiting for has  
11 taken place. That's why it's waiting on a condition variable!

12 There are many situations where having tasks communicate using a condvar is an  
13 good fit for the problem at hand, but this doesn't seem to be one of them.

14 For many developers, the next trick in their bag is a shared boolean flag. The flag is  
15 initially `false`. When the detecting thread recognizes the event it's looking for, it  
16 sets the flag:

```
17 std::atomic<bool> flag(false);           // shared flag; see  
18   // Item 40 for std::atomic  
19  
20 ...                                     // detect event  
21 flag = true;                            // tell reacting task
```

22 For its part, the reacting thread simply polls the flag. When it sees that the flag is  
23 set, it knows that the event it's been waiting for has occurred:

```
24 ...                                     // prepare to react  
25 while (!flag);                         // wait for event  
26 ...                                     // react to event
```

27 This approach suffers from none of the drawbacks of the condvar-based design.  
28 There's no need for a mutex, no problem if the detecting task sets the flag before  
29 the reacting task starts polling, and nothing akin to a spurious wakeup. Good, good,  
30 good.

1 Less good is the cost of polling in the reacting task. During the time the task is  
2 waiting for the flag to be set, the task is essentially blocked, yet it's still running. As  
3 such, it occupies a hardware thread that another task might be able to make use of,  
4 it incurs the cost of a context switch each time it starts or completes its timeslice,  
5 and it could keep a core running that might otherwise be shut down to save power.  
6 A truly blocked task would do none of these things. That's an advantage of the  
7 condvar-based approach, because a task in a `wait` call is truly blocked.

8 It's common to combine the condvar and flag-based designs. A flag indicates  
9 whether the event of interest has occurred, but access to the flag is synchronized  
10 by a mutex. Because the mutex prevents concurrent access to the flag, there is, as  
11 Item 40 explains, no need for the flag to be `std::atomic`; a simple `bool` will do.  
12 The detecting task would then look like this:

```
13 std::condition_variable cv;           // as before
14 std::mutex m;
15 bool flag(false);                  // not std::atomic
16 ...                                // detect event
17 {
18     std::lock_guard<std::mutex> g(m); // lock m via g's ctor
19     flag = true;                   // tell reacting task
20                         // (part 1)
21 }
22 cv.notify_one();                  // tell reacting task
23                         // (part 2)
```

24 And here's the reacting task:

```
25 ...
26 {
27     std::unique_lock<std::mutex> lk(m); // as before
28     cv.wait(lk, [] { return flag; }); // use lambda to avoid
29                         // spurious wakeups
30 ...
31 }
32 }
```

```
1 ... // continue reacting
2 // (m now unlocked)
```

3 This approach avoids the problems we've discussed. It works regardless of whether the reacting task `waits` before the detecting task notifies, it works in the presence of spurious wakeups, and it doesn't require polling. Yet an odor remains, because the detecting task communicates with the reacting task in a very curious fashion. Notifying the condition variable tells the reacting task that the event it's been waiting for has probably occurred, but the reacting task must check the flag to be sure. Setting the flag tells the reacting task that the event has definitely occurred, but the detecting task still has to notify the condition variable so that the reacting task will awaken and check the flag. The approach works, but it doesn't seem terribly clean.

13 An alternative is to avoid condition variables, mutexes, and flags by having the reacting task `wait` on a future that's set by the detecting task. This may seem like an odd idea. After all, Item 38 explains that a future represents the receiving end of a communications channel from a callee to a (typically asynchronous) caller, and here there's no callee-caller relationship between the detecting and reacting tasks. However, Item 38 also notes that a communications channel whose transmitting end is a `std::promise` and whose receiving end is a future can be used for more than just callee-caller communication. Such a communications channel can be used in any situation where you need to transmit information from one place in your program to another. In this case, we'll use it to transmit information from the detecting task to the reacting task, and the information we'll convey will be that the event of interest has taken place.

25 The design is simple. The detecting task has a `std::promise` object (i.e., the writing end of the communications channel), and the reacting task has a corresponding future. When the detecting task sees that the event it's looking for has occurred, it *sets* the `std::promise` (i.e., write into the communications channel). Meanwhile, the reacting task `waits` on its future. That `wait` blocks the reacting task until the `std::promise` has been set.

31 Now, both `std::promise` and `futures` (i.e., `std::future` and `std::shared_future`) are templates that require a type parameter. That param-

1   eter indicates the type of data to be transmitted through the communications  
2   channel. In our case, however, there's no data to be conveyed. The only thing of  
3   interest to the reacting task is that its future has been set. What we need for the  
4   **std::promise** and future templates is a type that indicates that no data is to be  
5   conveyed across the communications channel. That type is **void**. The detecting  
6   task will thus use a **std::promise<void>**, and the reacting task a  
7   **std::future<void>** or **std::shared\_future<void>**. The detecting task will  
8   set its **std::promise<void>** when the event of interest occurs, and the reacting  
9   task will **wait** on its future. Even though the reacting task won't receive any data  
10   from the detecting task, the communications channel will permit the reacting task  
11   to know when the detecting task has "written" its **void** data by calling **set\_value**  
12   on its **std::promise**.

13 So given

16 the detecting task's code is trivial,

```
17 ... // detect event  
18 p.set_value(); // tell reacting task
```

19 and the reacting task's code is equally simple:

```
20 ... // prepare to react
21 p.get_future().wait(); // wait on future
22 // corresponding to
23 ... // react to event
```

24 Like the approach using a flag, this design requires no mutex, works regardless of  
25 whether the detecting task sets its `std::promise` before the reacting task `waits`,  
26 and is immune to spurious wakeups. (Only condition variables are susceptible to  
27 that problem.) Like the condvar-based approach, the reacting task is truly blocked  
28 after making the `wait` call, so it consumes no system resources while waiting. Per-  
29 fect, right?

1 Not exactly. Sure, a future-based approach skirts those shoals, but there are other  
2 hazards to worry about. For example, Item 38 explains that between a  
3 `std::promise` and a future is a shared state, and shared states are typically dy-  
4 namically allocated. You should therefore assume that this design incurs the cost  
5 of heap-based memory allocation and deallocation.

6 Perhaps more importantly, a `std::promise` may be set only once. The communi-  
7 cations channel between a `std::promise` and a future is a *one-shot* mechanism: it  
8 can't be used repeatedly. This is a notable difference from the condvar- and flag-  
9 based designs, both of which can be used to communicate multiple times. (A  
10 condvar can be repeatedly notified, and a flag can always be cleared and set again.)

11 The one-shot restriction isn't as limiting as you might think. Suppose you'd like to  
12 create a system thread in a suspended state. That is, you'd like to get all the over-  
13 head associated with thread creation out of the way so that when you're ready to  
14 execute something on the thread, the normal thread-creation latency will be  
15 avoided. Or you might want to create a suspended thread so that you could config-  
16 ure it before letting it run. Such configuration might include things like setting its  
17 priority or core affinity. The C++ concurrency API offers no way to do those things,  
18 but `std::thread` objects offer the `native_handle` member function, the result  
19 of which is intended to give you access to the platform's underlying threading API  
20 (usually POSIX threads or Windows threads). The lower-level API often does make  
21 it possible to configure thread characteristics such as priority and affinity.

22 Assuming you want to suspend a thread only once (after creation, but before it's  
23 running its thread function), a design using a `void` future is a reasonable choice.  
24 Here's the essence of the technique:

```
25 std::promise<void> p;
26 void react(); // func for reacting task
27 void detect(); // func for detecting task
28 {
29     std::thread t([] // create thread
30     {
31         p.get_future().wait(); // suspend t until
32         react(); // future is set
33     });
34 }
```

```

1      ...
2                      // here, t is suspended
3      p.set_value();           // prior to call to react
4                      // unsuspend t (and thus
5      ...                   // call react)
6      t.join();              // do additional work
7  };
8      // make t unjoinable
9      // (see Item 37)

```

Because it's important that `t` become unjoinable on all paths out of `detect`, use of an RAII class like Item 37's `ThreadRAII` seems like it would be advisable. Code like this comes to mind:

```

11 void detect()
12 {
13     ThreadRAII tr(                         // use RAII object
14         std::thread[]
15         {
16             p.get_future().wait();
17             react();
18         },
19         ThreadRAII::DtorAction::join          // risky! (see below)
20     );
21
22     ...
23     // thread inside tr
24     // is suspended here
25
26     p.set_value();                        // unsuspend thread
27     // inside tr
28     ...
29
30 };

```

This looks safer than it is. The problem is that if in the first "..." region (the one with the "thread inside `tr` is suspended here" comment), an exception is emitted, `set_value` will never be called on `p`. That means that the call to `wait` inside the lambda will never return. That, in turn, means that the thread running the lambda will never finish, and that's a problem, because the RAII object `tr` has been configured to perform a `join` on that thread in `tr`'s destructor. In other words, if an exception is emitted from the first "..." region of code, this function will hang, because `tr`'s destructor will never complete.

1 There are ways to address this problem, but I'll leave them in the form of the hal-  
2 lowed exercise for the reader.<sup>†</sup> Here, I'd like to show how the original code (i.e., not  
3 using ThreadRAII) can be extended to suspend and then unsuspend not just one  
4 reacting task, but many. It's a simple generalization, because the key is to use  
5 `std::shared_futures` instead of a `std::future` in the `react` code. Once you  
6 know that the `std::future`'s `share` member function transfers ownership of its  
7 shared state to the `std::shared_future` object produced by `share`, the code  
8 nearly writes itself. The only subtlety is that each reacting thread needs its own  
9 copy of the `std::shared_future` that refers to the shared state, so the  
10 `std::shared_future` obtained from `share` is captured by value by the lambdas  
11 running on the reacting threads:

```
12 std::promise<void> p;                                // as before
13 void detect()   // now for multiple
14 {   // reacting tasks
15     auto sf = p.get_future().share(); // sf's type is
16   // std::shared_future<void>
17     std::vector<std::thread> vt;                      // container for
18   // reacting threads
19     for (int i = 0; i < threadsToRun; ++i) {
20         vt.emplace_back([sf]{ sf.wait();                // wait on local
21                               react();}); // copy of sf; see
22     }   // Item 42 for info
23   // on emplace_back
24     ...  // detect hangs if
25   // this "..." code throws!
26     p.set_value();                                     // unsuspend all threads
27     ...
28     for (auto& t : vt) {                            // make all threads
29         t.join();                                     // unjoinable; see Item 2
```

---

<sup>†</sup> A reasonable place to begin researching the matter is my 24 December 2013 blog post at *The View From Aristeia*, “[ThreadRAII + Thread Suspension = Trouble?](#)”

```
1     }                                // for info on "auto&"  
2 };  
  
3 The fact that a design using futures can achieve this effect is noteworthy, and that's  
4 why you should consider it for one-shot event communication.
```

## 5 **Things to Remember**

- 6   ♦ For simple event communication, condvar-based designs require a superfluous  
7   mutex, impose constraints on the relative progress of detecting and reacting  
8   tasks, and require reacting tasks to verify that the event has taken place.
- 9   ♦ Designs employing a flag avoid those problems, but are based on polling, not  
10   blocking.
- 11   ♦ A condvar and flag can be used together, but the resulting communications  
12   mechanism is somewhat stilted.
- 13   ♦ Using `std::promises` and `futures` dodges these issues, but it uses heap  
14   memory for shared states, and it's limited to one-shot communication.

## 15 **Item 40: Use `std::atomic` for concurrency, `volatile` for 16 special memory.**

- 17 Poor `volatile`. So misunderstood. It shouldn't even be in this chapter, because it  
18 has nothing to do with concurrent programming. But in other programming lan-  
19 guages (e.g., Java and C#), it is useful for such programming, and even in C++, some  
20 compilers have imbued `volatile` with semantics that render it applicable to con-  
21 current software (but only when compiled with those compilers). It's thus worth-  
22 while to discuss `volatile` in a chapter on concurrency if for no other reason than  
23 to dispel the confusion surrounding it.
- 24 The C++ feature that programmers sometimes confuse `volatile` with—the fea-  
25 ture that definitely does belong in this chapter—is the `std::atomic` template.  
26 Instantiations of this template (e.g., `std::atomic<int>`, `std::atomic<bool>`,  
27 `std::atomic<Widget*>`, etc.) offer operations that are guaranteed to be seen as  
28 atomic by other threads. Once a `std::atomic` object has been constructed, opera-  
29 tions on it behave as if they were inside a mutex-protected critical section, but the

1 operations are generally implemented using special machine instructions that are  
2 more efficient than would be the case if a mutex were employed.

3 Consider this code using `std::atomic`:

```
4 std::atomic<int> ai(0);      // initialize ai to 0
5 ai = 10;                      // atomically set ai to 10
6 std::cout << ai;              // atomically read ai's value
7 ++ai;                         // atomically increment ai to 11
8 --ai;                         // atomically decrement ai to 10
```

9 During execution of these statements, other threads reading `ai` may see only val-  
10 ues of 0, 10, or 11. No other values are possible (assuming, of course, that this is  
11 the only thread modifying `ai`).

12 Two aspects of this example are worth noting. First, in the “`std::cout << ai;`”  
13 statement, the fact that `ai` is a `std::atomic` guarantees only that the read of `ai` is  
14 atomic. There is no guarantee that the entire statement proceeds atomically. Be-  
15 tween the time `ai`’s value is read and `operator<<` is invoked to write it to the  
16 standard output, another thread—possibly several threads—may have modified  
17 `ai`’s value. That has no effect on the behavior of the statement, because opera-  
18 `tor<<` for `ints` uses a by-value parameter for the `int` to output (the outputted  
19 value will therefore be the one that was read from `ai`), but it’s important to under-  
20 stand that what’s atomic in that statement is nothing more than the read of `ai`.

21 The second noteworthy aspect of the example is the behavior of the last two  
22 statements—the increment and decrement of `ai`. These are each read-modify-  
23 write (RMW) operations, yet they execute atomically. This is one of the nicest  
24 characteristics of the `std::atomic` types: once a `std::atomic` object has been  
25 constructed, all member functions on it, including those comprising RMW opera-  
26 tions, are guaranteed to be seen by other threads as atomic.

27 In contrast, the corresponding code using `volatile` guarantees virtually nothing  
28 in a multithreaded context:

```
29 volatile int vi(0);          // initialize vi to 0
```

```
1  vi = 10;                      // set vi to 10
2  std::cout << vi;              // read vi's value
3  ++vi;                         // increment vi to 11
4  --vi;                          // decrement vi to 10
5
6  During execution of this code, if other threads are reading the value of vi, they
7  may see anything (e.g., -12, 68, 4090727—anything!). Such code would have unde-
8  fined behavior, because these statements modify vi, so if other threads are read-
9  ing vi at the same time, there are simultaneous readers and writers of memory
10 that's neither std::atomic nor protected by a mutex, and that's the definition of a
11 data race.
```

12 As a concrete example of how the behavior of `std::atomics` and `volatiles` can  
13 differ in a multithreaded program, consider a simple counter of each type that's  
incremented by multiple threads. We'll initialize each to 0:

```
14  std::atomic<int> ac(0);      // "atomic counter"
15  volatile int vc(0);          // "volatile counter"
```

16 We'll then increment each counter one time in two simultaneously-running  
17 threads:

```
18  /*----- Thread 1 -----*/    /*----- Thread 2 -----*/
19  ++ac;                        ++ac;
20  ++vc;                        ++vc;
```

21 When both threads have finished, ac's value (i.e., the value of the `std::atomic`)  
22 must be 2, because each increment occurs as an indivisible operation. vc's value,  
23 on the other hand, need not be 2, because its increments may not occur atomically.  
24 Each increment consists of reading vc's value, incrementing the value that was  
25 read, and writing the result back into vc. But these three operations are not guar-
26 anteed to proceed atomically for `volatile` objects, so it's possible that the com-
27 ponent parts of the two increments of vc are interleaved as follows:

- 28 1. Thread 1 reads vc's value, which is 0.
- 29 2. Thread 2 reads vc's value, which is still 0.

1    3. Thread 1 increments the 0 it read to 1, then writes that value into `vc`.  
2    4. Thread 2 increments the 0 it read to 1, then writes that value into `vc`.  
3    `vc`'s final value is therefore 1, even though it was incremented twice.  
4    This is not the only possible outcome. `vc`'s final value is, in general, not predicta-  
5    ble, because `vc` is involved in a data race, and the Standard's decree that data races  
6    cause undefined behavior means that compilers may generate code to do literally  
7    anything. Compilers don't use this leeway to be malicious, of course. Rather, they  
8    perform optimizations that would be valid in programs without data races, and  
9    these optimizations yield unexpected and unpredictable behavior in programs  
10   where races are present.

11   The use of RMW operations isn't the only situation where `std::atomics` com-  
12   prise a concurrency success story and `volatiles` suffer ignominious failure. Sup-  
13   pose one task computes an important value needed by a second task. When the  
14   first task has computed the value, it must communicate this to the second task.  
15   Item 39 explains that one way for the first task to communicate the availability of  
16   the desired value to the second task is by using a `std::atomic<bool>`. Code in  
17   the task computing the value would look something like this:

```
18 std::atomic<bool> valAvailable(false);
19 auto imptValue = computeImportantValue(); // compute value
20 valAvailable = true; // tell other task
21                       // it's available
```

22   As humans reading this code, we know it's crucial that the assignment to `impt-`  
23   `Value` take place before the assignment to `valAvailable`, but all compilers see is  
24   a pair of assignments to independent variables. As a general rule, compilers are  
25   permitted to reorder such unrelated assignments. That is, given this sequence of  
26   assignments (where `a`, `b`, `x`, and `y` correspond to independent variables),

```
27 a = b;
28 x = y;
```

29   compilers may generally reorder them as follows:

```
1  x = y;  
2  a = b;  
3 Even if compilers don't reorder them, the underlying hardware might do it (or  
4 might make it seem to other cores as if it had), because that can sometimes make  
5 the code run faster.
```

```
6 However, the use of std::atomics imposes restrictions on how code can be reor-  
7 dered, and one such restriction is that no code that, in the source code, precedes a  
8 write of a std::atomic variable may take place (or appear to other cores to take  
9 place) afterwards.† That means that in our code,
```

```
10 auto imptValue = computeImportantValue(); // compute value  
11 valAvailable = true; // tell other task  
12 // it's available  
13 not only must compilers retain the order of the assignments to imptValue and  
14 valAvailable, they must generate code that ensures that the underlying hard-  
15 ware does, too. As a result, declaring valAvailable as std::atomic ensures that  
16 our critical ordering requirement—imptValue must be seen by all threads to  
17 change no later than valAvailable does—is maintained.
```

```
18 Declaring valAvailable as volatile doesn't impose the same code reordering  
19 restrictions:
```

```
20 volatile bool valAvailable(false);  
21 auto imptValue = computeImportantValue();
```

---

<sup>†</sup> This is true only for `std::atomics` using *sequential consistency*, which is both the default and the only consistency model for `std::atomic` objects that use the syntax shown in this book. C++11 also supports consistency models with more flexible code-reordering rules. Such *weak* (aka *relaxed*) models make it possible to create software that runs faster on some hardware architectures, but the use of such models yields software that is *much* more difficult to get right, much less understand and maintain. Subtle errors in code using relaxed atomics is not uncommon, even for experts, so you should stick to sequential consistency if at all possible.

```
1 valAvailable = true; // other threads might see this assignment  
2 // before the one to imptValue!
```

3 Here, compilers might flip the order of the assignments to `imptValue` and  
4 `valAvailable`, and even if they don't, they might fail to generate machine code  
5 that would prevent the underlying hardware from making it possible for code on  
6 other cores to see `valAvailable` change before `imptValue`.

7 These two issues—no guarantee of operation atomicity and insufficient re-  
8 strictions on code reordering—explain why `volatile`'s not useful for concurrent  
9 programming, but it doesn't explain what it is useful for. In a nutshell, it's for tell-  
10 ing compilers that they're dealing with memory that doesn't behave normally.

11 “Normal” memory has the characteristic that if you write a value to a memory loca-  
12 tion, the value remains there until something overwrites it. So if I have a normal  
13 `int`,

```
14 int x;
```

15 and a compiler sees the following sequence of operations on it,

```
16 auto y = x;           // read x  
17 y = x;               // read x again
```

18 the compiler can optimize the generated code by eliminating the assignment to `y`,  
19 because it's redundant with `y`'s initialization.

20 Normal memory also has the characteristic that if you write a value to a memory  
21 location, never read it, and then write to that memory location again, the first write  
22 can be eliminated, because it was never used. So given these two adjacent state-  
23 ments,

```
24 x = 10;              // write x  
25 x = 20;              // write x again
```

26 compilers can eliminate the first one. That means that if we have this in the source  
27 code,

```
28 auto y = x;          // read x  
29 y = x;               // read x again
```

```
1  x = 10;           // write x  
2  x = 20;           // write x again
```

3 compilers can treat it as if it had been written like this:

```
4  auto y = x;       // read x  
5  x = 20;           // write x
```

6 Lest you wonder who'd write code that performs these kinds of redundant reads  
7 and superfluous writes (technically known as *redundant loads* and *dead stores*),  
8 the answer is that humans don't write it directly—at least we hope they don't.  
9 However, after compilers take reasonable-looking source code and perform tem-  
10 plate instantiation, inlining, and various common kinds of reordering optimiza-  
11 tions, it's not uncommon for the result to have redundant loads and dead stores  
12 that compilers can get rid of.

13 Such optimizations are valid only if memory behaves normally. "Special" memory  
14 doesn't. Probably the most common kind of special memory is memory used for  
15 *memory-mapped I/O*. Locations in such memory actually communicate with pe-  
16 ripherals, e.g., external sensors or displays, printers, network ports, etc. rather  
17 than reading or writing normal memory (i.e., RAM). In such a context, consider  
18 again the code with seemingly redundant reads:

```
19 auto y = x;         // read x  
20 y = x;             // read x again
```

21 If *x* corresponds to, say, the value reported by a temperature sensor, the second  
22 read of *x* is not redundant, because the temperature may have changed between  
23 the first and second reads.

24 It's a similar situation for seemingly superfluous writes. In this code, for example,

```
25 x = 10;            // write x  
26 x = 20;            // write x again
```

27 if *x* corresponds to the control port for a radio transmitter, it could be that the  
28 code is issuing commands to the radio, and the value 10 corresponds to a different  
29 command from the value 20. Optimizing out the first assignment would change the  
30 sequence of commands sent to the radio.

1    `volatile` is the way we tell compilers that we're dealing with special memory. Its  
2    meaning to compilers is "Don't perform any optimizations on operations on this  
3    memory." So if `x` corresponds to special memory, it'd be declared `volatile`:

4    `volatile int x;`

5    Consider the effect that has on our original code sequence:

6    `auto y = x;                      // read x`  
7    `y = x;                          // read x again (can't be optimized away)`  
8    `x = 10;                        // write x (can't be optimized away)`  
9    `x = 20;                        // write x again`

10   This is precisely what we want if `x` is memory-mapped (or has been mapped to a  
11   memory location shared across processes, etc.).

12   Pop quiz! In that last piece of code, what is `y`'s type: `int` or `volatile int`?†

13   The fact that seemingly redundant loads and dead stores must be preserved when  
14   dealing with special memory explains, by the way, why `std::atomics` are unsuit-  
15   able for this kind of work. Compilers are permitted to eliminate such redundant  
16   operations on `std::atomics`. The code isn't written quite the same way it is for  
17   `volatiles`, but if we overlook that for a moment and focus on what compilers are  
18   permitted to do, we can say that, conceptually, compilers may take this,

19   `std::atomic<int> x;`

20   `auto y = x;                      // conceptually read x (see below)`  
21   `y = x;                          // conceptually read x again (see below)`  
22   `x = 10;                        // write x`  
23   `x = 20;                        // write x again`

---

† `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 and optimize it to this:

```
2 auto y = x;           // conceptually read x (see below)
3 x = 20;              // write x
```

4 For special memory, this is clearly unacceptable behavior.

5 Now, as it happens, neither of these two statements will compile when `x` is  
6 `std::atomic`:

```
7 auto y = x;           // error!
8 y = x;                // error!
```

9 That's because the copy operations for `std::atomic` are deleted (see Item 11).  
10 And with good reason. Consider what would happen if the initialization of `y` with `x`  
11 compiled. Because `x` is `std::atomic`, `y`'s type would be deduced to be  
12 `std::atomic`, too (see Item 2.) I remarked earlier that one of the best things  
13 about `std::atomics` is that all their operations are atomic, but in order for the  
14 copy construction of `y` from `x` to be atomic, compilers would have to generate code  
15 to read `x` and write `y` in a single atomic operation. Hardware generally can't do  
16 that, so copy construction isn't supported for `std::atomic` types. Copy assign-  
17 ment is deleted for the same reason, which is why the assignment from `x` to `y`  
18 won't compile. (The move operations aren't explicitly declared in `std::atomic`,  
19 so, per the rules for compiler-generated special functions described in Item 17,  
20 `std::atomic` offers neither move construction nor move assignment.)

21 It's possible, of course, to get the value of `x` into `y`, but it requires use of  
22 `std::atomic`'s member functions `load` and `store`. The `load` member function  
23 reads a `std::atomic`'s value atomically, while the `store` member function writes  
24 it atomically. To initialize `y` with `x`, then, followed by putting `x`'s value in `y`, the  
25 code must be written like this:

```
26 std::atomic<int> y(x.load());      // read x
27 y.store(x.load());                 // read x again
```

28 This compiles, but the fact that reading `x` (via `x.load()`) is a separate function call  
29 from initializing or storing to `y` makes clear that there is no reason to expect either  
30 statement as a whole to execute as a single atomic operation.

1 Given that code, compilers could “optimize” it by storing `x`’s value in a register instead of reading it twice:

```
3 register = x.load();           // read x into register
4 std::atomic<int> y(register);    // init y with register value
5 y.store(register);             // store register value into y
```

6 The result, as you can see, reads from `x` only once, and that’s the kind of optimization that must be avoided when dealing with special memory. (The optimization isn’t permitted for `volatile` variables.)

9 The situation should thus be clear:

- 10 • `std::atomic` is useful for concurrent programming, but not for accessing special memory.
- 12 • `volatile` is useful for accessing special memory, but not for concurrent programming.

14 Because `std::atomic` and `volatile` serve different purposes, they can even be used together:

```
16 volatile std::atomic<int> vai;      // operations on vai are
17                                // atomic and can't be
18                                // optimized away
```

19 This could be useful if `vai` corresponded to a memory-mapped I/O location that was concurrently accessed by multiple threads.

21 As a final note, some developers prefer to use `std::atomic`’s `load` and `store` member functions even when they’re not required, because it makes explicit in the source code that the variables involved aren’t “normal.” Emphasizing that fact isn’t unreasonable. Accessing a `std::atomic` is typically much slower than accessing a non-`std::atomic`, and we’ve already seen that the use of `std::atomics` prevents compilers from performing certain kinds of code reorderings that would otherwise be permitted. Calling out loads and stores of `std::atomics` can therefore help identify potential scalability chokepoints. From a correctness perspective, *not* seeing a call to `store` on a variable meant to communicate information to other

1 threads (e.g., a flag indicating the availability of data) could mean that the variable  
2 wasn't declared `std::atomic` when it should have been.

3 This is largely a stylistic issue, however, and as such is quite different from the  
4 choice between `std::atomic` and `volatile`.

5 **Things to Remember**

6 ♦ `std::atomic` is for data accessed from multiple threads without using mutex-  
7 es. It's a tool for writing concurrent software.

8 ♦ `volatile` is for memory where reads and writes should not be optimized  
9 away. It's a tool for working with special memory.

## 1    **Chapter 8   Tweaks**

2    For every general technique or feature in C++, there are circumstances where it's  
3    reasonable to use it, and there are other circumstances where it's not. Describing  
4    when it can make sense to use a general technique or feature is usually fairly  
5    straightforward, but this chapter covers two exceptions. The general technique is  
6    pass by value, and the general feature is emplacement, and the decision about  
7    when to employ them will be influenced by so many factors, the best advice I can  
8    give is to *consider* their use. Nevertheless, both are important players in effective  
9    modern C++ programming, and the Items that follow provide the information  
10   you'll need to determine when using them is appropriate for your software.

### 11    **Item 41: Consider pass by value for copyable parameters 12        that are cheap to move and always copied.<sup>†</sup>**

13   Some function parameters are intended to be copied. Setters, for example, store  
14   the value of their parameter in a data member. For efficiency, such functions  
15   should copy lvalue arguments and move rvalue arguments:

```
16   class Widget {  
17   public:  
18       void setName(const std::string& newName)      // take lvalue;  
19       { name = newName; }                            // copy it  
20       void setName(std::string&& newName)           // take rvalue;  
21       { name = std::move(newName); }                  // move it  
22       ...  
23   private:  
24       std::string name;  
25   };
```

---

<sup>†</sup> In this Item, “always copied” means “always copied or moved from.” Recall from page 10 that C++ has no terminology to distinguish a copy made by a copy operation from one made by a move operation.

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.

9 An alternative approach is to make `setName` a function template taking a universal  
10 reference (see Item 24):

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

19 This reduces the source code you have to deal with, but the use of universal refer-  
20 ences leads to other complications. As a template, `setName`’s implementation must  
21 typically be in a header file. It may yield several functions in object code, because it  
22 not only instantiates differently for lvalues and rvalues, it also instantiates differ-  
23 ently for `std::string` and types *convertible* to `std::string` (see Item 25). Yet at  
24 the same time, there are argument types that can’t be passed by universal refer-  
25 ence (see Item 30), and if clients pass improper argument types, compiler error  
26 messages can be intimidating (see Item 27).

27 Wouldn’t it be nice if there were a way to write functions like `setName` such that  
28 lvalues were copied, rvalues were moved, there was only one function to deal with  
29 (in both source and object code), and the idiosyncrasies of universal references  
30 were avoided? As it happens, there is. All you have to do is abandon one of the first  
31 rules you probably learned as a C++ programmer. That rule was to avoid passing  
32 objects of user-defined types by value. For parameters like `newName` in functions  
33 like `setName`, pass by value may be exactly what you want.

1 Before we discuss *why* pass-by-value may be a good fit for `newName` and `setName`,  
2 let's see how it would be implemented.

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

9 The only non-obvious part of this code is the application of `std::move` to the pa-  
10 rameter `newName`. Typically, `std::move` is used with rvalue references (see  
11 Item 25), but in this case, we know that (1) `newName` is a completely independent  
12 object from whatever the caller passed in, so changing `newName` won't affect call-  
13 ers and (2) this is the final use of `newName`, so moving from it won't have any im-  
14 pact on the rest of the function.

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

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

```
24 Widget w;  
25 ...  
26 std::string widgetID("Bart");  
27 w.setName(widgetID);           // call setName with lvalue  
28 ...  
29 w.setName(widgetID + "Jenne"); // call setName with rvalue  
30                                // (see below)
```

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

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

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

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

34 And here are the two calling scenarios we've examined:

```
35 Widget w;  
36 ...  
37 std::string widgetID("Bart");
```

```

1 w.setName(widgetID);                                // pass lvalue
2 ...
3 w.setName(widgetID + "Jenne");                   // pass rvalue
4 Now consider the cost, in terms of copy and move operations, of setting a Widget's
5 name for the two calling scenarios and each of the three setName implementations
6 we've discussed. Our accounting will largely ignore the possibility of compilers
7 finding ways to optimize copy and move operations away, because such optimiza-
8 tions are context- and compiler-dependent and, in practice, don't change the es-
9 sence of the analysis.
10 • Overloading: Regardless of whether an lvalue or an rvalue is passed, the call-
11 er's argument is bound to a reference called newName. That costs nothing, in
12 terms of copy and move operations. In the lvalue overload, newName is copied
13 into Widget::name. In the rvalue overload, it's moved. Cost summary: 1 copy
14 for lvalues, 1 move for rvalues.
15 • Using a universal reference: As with overloading, the caller's argument is
16 bound to the reference newName. This is a no-cost operation. Due to the use of
17 std::forward (see Item 23), lvalue std::string arguments are copied into
18 Widget::name, while rvalue std::string arguments are moved. The cost
19 summary for std::string arguments is the same as with overloading: 1 copy
20 for lvalues, 1 move for rvalues.
21 Item 25 explains that if a caller passes an argument of a type other than
22 std::string, it will be forwarded to a std::string assignment operator,
23 and that could cause as few as zero std::string copy or move operations to
24 be performed. Functions taking universal references can thus be uniquely effi-
25 cient. However, that doesn't affect the analysis in this Item, so we'll keep things
26 simple by assuming that callers always pass std::string arguments.
27 • Passing by value: Regardless of whether an lvalue or an rvalue is passed, the
28 parameter newName must be constructed. If an lvalue is passed, this costs a
29 copy operation. If an rvalue is passed, it costs a move. In the body of the func-
30 tion, newName is then unconditionally moved into Widget::name. The cost
31 summary is thus 1 copy plus 1 move for lvalues, and 2 moves for rvalues.

```

- 1 Look again at this Item's title:
- 2 Consider pass by value for copyable parameters that are cheap to move
- 3 and always copied.
- 4 It's worded the way it is for a reason. Four reasons, in fact:
- 5 1. You should only *consider* using pass by value. Yes, it requires writing only one
- 6 function. Yes, it generates only one function in the object code. Yes, it avoids
- 7 the issues associated with universal references. But it has a higher cost than
- 8 the alternatives. In particular, it costs an extra move for both lvalue and rvalue
- 9 arguments.
- 10 2. Consider pass by value only for *copyable parameters*. Parameters failing this
- 11 test must be of move-only types, because if they're not copyable, yet the func-
- 12 tion always makes a copy, the copy must be created via the move constructor.<sup>†</sup>
- 13 Recall that the advantage of pass by value over overloading is that with pass by
- 14 value, only one function has to be written. But for move-only types, there is no
- 15 need to provide an overload for lvalue arguments, because copying an lvalue
- 16 entails calling the copy constructor, and the copy constructor for move-only
- 17 types is disabled. That means that only rvalue arguments need to be support-
- 18 ed, and in that case, the "overloading" solution requires only one overload: the
- 19 one taking an rvalue reference.
- 20 Consider a class with a `std::unique_ptr<std::string>` data member and
- 21 a setter for it. `std::unique_ptr` (see Item 18) is a move-only type, so the
- 22 "overloading" approach to its setter consists of a single function:
- ```
23 class Widget {  
24 public:  
25     ...  
26     void setPtr(std::unique_ptr<std::string>&& ptr)  
27     { p = std::move(ptr); }
```

---

<sup>†</sup> Sentences like this are why it'd be nice to have terminology that distinguishes copies made via copy operations from copies made via move operations.

```
1   private:  
2     std::unique_ptr<std::string> p;  
3   };
```

4 A caller might use it this way:

```
5   Widget w;  
6   ...  
7   w.setPtr(std::make_unique<std::string>("C++"));
```

8 Here the rvalue `std::unique_ptr<std::string>` returned from  
9 `std::make_unique` (see Item 21) is passed by rvalue reference to `setPtr`,  
10 where it's moved into the data member `p`. The total cost is one move.

11 If `setPtr` were to take its parameter by value,

```
12  class Widget {  
13  public:  
14    ...  
15    void setPtr(std::unique_ptr<std::string> ptr)  
16    { p = std::move(ptr); }  
17    ...  
18  };
```

19 the same call would move-construct the parameter `ptr`, and `ptr` would then  
20 be move-assigned into the data member `p`. The total cost would thus be two  
21 moves—twice that of the “overloading” approach.

22 3. Pass by value is worth considering only for parameters that are *cheap to move*.  
23 When moves are cheap, the cost of an extra move is likely to be negligible. But  
24 when moves are not cheap, extra ones can incur an expense you can't afford.  
25 Item 29 explains that not all types are cheap to move—not even all types in the  
26 Standard Library. When moves are not cheap, performing an unnecessary  
27 move is analogous to performing an unnecessary copy, and the importance of  
28 avoiding unnecessary copy operations is what led to the C++98 rule about  
29 avoiding pass by value in the first place!

30 4. You should consider pass by value only for parameters that are *always copied*.  
31 For setter functions (as well as for constructors taking initialization arguments  
32 for an object's data members), this condition is typically fulfilled, but consider

1 a function that adds a value to a data structure only if the value satisfies a con-  
2 straint. Using pass by value, it could be written like this:

```
3 class Widget {  
4 public:  
5     bool insert(std::string s)  
6     {  
7         if ((s.length() >= MinLen) && (s.length() <= MaxLen)) {  
8             values.insert(std::move(s));  
9             return true;  
10        }  
11        else {  
12            return false;  
13        }  
14    }  
15    ...  
16 private:  
17     std::unordered_set<std::string> values;  
18 };
```

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

```
21 Widget w;  
22  
23 std::string famousLastWords("Tomorrow is another day");  
24 ...  
25 auto status = w.insert(famousLastWords);
```

26 we'll pay to copy `famousLastWords`, even if its length is less than `MinLen` or  
27 greater than `MaxLen`. (Note that we'll really pay for a copy, not a move, because  
28 `famousLastWords` is an lvalue.) In contrast, the overloading- and universal  
29 reference-based approaches neither copy nor move anything unless the pa-  
30 rameter's value satisfies the insertion constraint.

31 Even when you're dealing with a function performing an unconditional copy on a  
32 copyable type that's known to be cheap to move, there are times when pass by  
33 value may not be a suitable design decision. For code that has to be as fast as pos-  
34 sible, avoiding even cheap moves can be important. Besides, it's not always clear  
35 how many moves are being performed. In our `Widget::setName` example, pass

1 by value incurs only a single extra move operation, but suppose that `Widget::setName` called `Widget::validateName`, and this function also passed by  
2 value. (Presumably it has a reason for always copying its parameter, e.g., to store it  
3 in a data structure of all values it validates.) And suppose that `validateName`  
4 called a third function that also passed by value...

5  
6 You can see where this is headed. When there are chains of function calls, each of  
7 which employs pass by value because “it costs only one inexpensive move,” the  
8 cost for the entire chain of calls may not be something you can tolerate. Using by-  
9 reference parameter passing (as is the case with lvalue and rvalue overloads and  
10 the use of universal references), chains of calls don’t incur this kind of accumulated  
11 overhead.

12 If you were paying close attention during this Item, you probably noticed my  
13 comment that there’s “generally” no catch to using pass by value, provided the  
14 constraints we’ve discussed are satisfied. “Generally?,” you probably wondered.  
15 “Generally?” “What’s up with ‘generally’?”

16 What’s up with “generally” is *the slicing problem*. Pass by value is susceptible to it.  
17 Pass by reference isn’t. This is well-trod C++98 ground, so I won’t dwell on it, but if  
18 you have a function that is designed to accept a parameter of a base class type *or*  
19 *any type derived from it*, you don’t want to declare a pass-by-value parameter of  
20 that type, because you’ll “slice off” the derived-class characteristics of any derived  
21 type object that may be passed in:

```
22 class Widget { ... };                                // base class
23 class SpecialWidget: public Widget { ... };      // derived class
24 void processWidget(Widget w);    // func for any kind of Widget,
25                                         // including derived types;
26 ...                                         // suffers from slicing problem
27 SpecialWidget sw;
28 ...
29 processWidget(sw);                                // processWidget sees a
30                                         // Widget, not a SpecialWidget!
```

1 If you're not familiar with the slicing problem, search engines and the Internet are  
2 your friends; there's lots of information available. You'll find that the existence of  
3 the slicing problem is another reason (on top of the efficiency hit) why pass by  
4 value has a shady reputation in C++98. There are good reasons why one of the first  
5 things you probably learned about C++ programming was to avoid passing objects  
6 of user-defined types by value.

7 C++11 doesn't fundamentally change the C++98 wisdom regarding pass by value.  
8 In general, pass by value still entails a performance hit you'd prefer to avoid, and  
9 pass by value can still lead to the slicing problem. What's new in C++11 is the dis-  
10 tinction between lvalue and rvalue arguments. Implementing functions that take  
11 advantage of move semantics for rvalues of copyable types requires either writing  
12 multiple functions (i.e., overloading for lvalues and rvalues) or using universal ref-  
13 erences, both of which have drawbacks. For the special case of copyable, cheap-to-  
14 move types passed to functions that always copy them and where slicing is not a  
15 concern, pass by value offers an easy-to-implement alternative that's nearly as ef-  
16 ficient as its pass-by-reference competitors, but avoids their disadvantages.

17 **Things to Remember**

- 18 ♦ For copyable, cheap-to-move parameters that are intended to be copied, pass  
19 by value is nearly as efficient as pass by reference, it's easier to implement, and  
20 it can generate less object code.
- 21 ♦ Pass by value is subject to the slicing problem, so it's typically inappropriate  
22 for base class parameter types.

23 **Item 42: Consider emplacement instead of insertion.**

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

29 Logical though this may be, it's not always true. Consider this code:

```
1 std::vector<std::string> vs; // container of std::string
2 vs.push_back("xyzzy"); // add string literal
3 Here, the container holds std::strings, but what you have in hand—what you're
4 actually trying to push_back—is a string literal, i.e., a sequence of characters in-
5 side quotes. A string literal is not a std::string, and that means that the argu-
6 ment you're passing to push_back is not of the type held by the container.
```

7 push\_back for std::vector is overloaded for lvalues and rvalues as follows:

```
8 template <class T, // from the C++11
9         class Allocator = allocator<T> > // Standard
10 class vector {
11 public:
12 ...
13 void push_back(const T& x); // insert lvalue
14 void push_back(T&& x); // insert rvalue
15 ...
16 };
```

17 In the call

```
18 vs.push_back("xyzzy");
19 compilers see a mismatch between the type of the argument (const char[6]) and
20 the type of the parameter taken by push_back (a reference to a std::string).
21 They address the mismatch by generating code to create a temporary
22 std::string object from the string literal, and they pass that temporary object to
23 push_back. In other words, they treat the call as if it had been written like this:
```

```
24 vs.push_back(std::string("xyzzy")); // create temp. std::string
25 // and pass it to push_back
```

26 The code compiles and runs, and everybody goes home happy. Everybody except  
27 the performance freaks, that is, because the performance freaks recognize that this  
28 code isn't as efficient as it should be.

29 To create a new element in a container of std::strings, they understand, a  
30 std::string constructor is going to have to be called, but the code above doesn't  
31 make just one constructor call. It makes two. And it calls the std::string de-  
32 structor, too. Here's what happens at run time in the call to push\_back:

1    1. A temporary `std::string` object is created from the string literal "xyzzy".  
2    This object has no name; we'll call it *temp*. Construction of *temp* is the first  
3    `std::string` construction. Because it's a temporary object, *temp* is an rvalue.

4    2. *temp* is passed to the rvalue overload for `push_back`, where it's bound to the  
5    rvalue reference parameter *x*. A copy of *x* is then constructed in the memory  
6    for the `std::vector`. This construction—the *second* one—is what actually  
7    creates a new object inside the `std::vector`. (The constructor that's used to  
8    copy *x* into the `std::vector` is the move constructor, because *x*, being an  
9    rvalue reference, gets cast to an rvalue before it's copied. For information  
10   about the casting of rvalue reference parameters to rvalues, see Item 25.)

11   3. Immediately after `push_back` returns, *temp* is destroyed, thus calling the  
12   `std::string` destructor.

13   The performance freaks can't help but notice that if there were a way to take the  
14   string literal and pass it directly to the code in step 2 that constructs the  
15   `std::string` object inside the `std::vector`, i.e., that performs the only con-  
16   struction that's actually required, we could avoid constructing and destroying  
17   *temp*. That would be maximally efficient, and even the performance freaks could  
18   contentedly decamp.

19   Because you're a C++ programmer, there's an above-average chance you're a per-  
20   formance freak. If you're not, you're still probably sympathetic to their point of  
21   view. (If you're not at all interested in performance, shouldn't you be in the Python  
22   room down the hall?) So I'm pleased to tell you that there is a way to do exactly  
23   what is needed for maximal efficiency in the call to `push_back`. It's to not call  
24   `push_back`. `push_back` is not the droid you're looking for. The function you want  
25   is `emplace_back`.

26   `emplace_back` does exactly what we desire: it uses whatever arguments are  
27   passed to it to construct a new `std::string` directly inside the `std::vector`. No  
28   temporaries are involved:

29   `vs.emplace_back("xyzzy"); // construct std::string inside`  
30                                 `// vs directly from "xyzzy"`

1 `emplace_back` uses perfect forwarding, so, as long as you don't bump into one of  
2 perfect forwarding's limitations (see Item 30), you can pass any number of argu-  
3 ments of any combination of types through `emplace_back`. For example, if you'd  
4 like to create a `std::string` in `vs` via the `std::string` constructor taking a  
5 character and a repeat count, this would do it:

8   `emplace_back` is available for every standard container that supports  
9   `push_back`. Similarly, every standard container that supports `push_front` sup-  
10   ports `emplace_front`. And every standard container that supports `insert`  
11   (which is all but `std::forward_list` and `std::array`) supports `emplace`. The  
12   associative containers offer `emplace_hint` to complement their `insert` functions  
13   that take a “hint” iterator, and `std::forward_list` has `emplace_after` to match  
14   its `insert_after`.

What makes it possible for emplacement functions to outperform insertion functions is their more flexible interface. Insertion functions take *objects to be inserted*, while emplacement functions take *constructor arguments for objects to be inserted*. This difference permits emplacement functions to avoid the creation and destruction of temporary objects that insertion functions can necessitate. Because an argument of the type held by the container can be passed to an emplacement function (the argument thus causes the function to perform copy or move construction), emplacement can be used even when an insertion function would require no temporary. In that case, insertion and emplacement do essentially the same thing.

For example, given

```
25     std::string queenOfDisco("Donna Summer");
```

26 both of the following calls are valid, and both have the same net effect on the con-  
27 tainer:

```
28 vs.push_back(queenOfDisco);           // copy-construct queenOfDisco  
29                                         // at end of vs  
  
30 vs.emplace_back(queenOfDisco);        // ditto
```

1 Emplacement functions can thus do everything insertion functions can. They  
2 sometimes do it more efficiently, and, at least in theory, they should never do it  
3 less efficiently. So why not use them all the time?

4 Because, as the saying goes, in theory, there's no difference between theory and  
5 practice, but in practice, there is. With contemporary implementations of the  
6 Standard Library, there are situations where, as expected, emplacement outper-  
7 forms insertion, but, sadly, there are also situations where the insertion functions  
8 currently run faster. Such situations are not easy to characterize, because they de-  
9 pend on the types of arguments being passed, the containers being used, the loca-  
10 tions in the containers where insertion or emplacement is requested, the excep-  
11 tion safety of the contained types' constructors, and, for containers where dupli-  
12 cate values are prohibited (i.e., `std::set`, `std::map`, `std::unordered_set`,  
13 `std::unordered_map`), whether the value to be added is already in the container.  
14 The usual performance-tuning advice thus applies: to determine whether em-  
15 placement or insertion runs faster, benchmark them both.

16 That's not very satisfying, of course, so you'll be pleased to learn that there's a heu-  
17 ristic that can help you identify situations where emplacement functions are most  
18 likely to be worthwhile. If all the following are true, emplacement will almost cer-  
19 tainly outperform insertion:

20 • **The value being added is constructed into the container, not assigned.**  
21 The example that opened this Item (adding a `std::string` with the value  
22 "xyzzy" to a `std::vector` `vs`) showed the value being added to the end of  
23 `vs`—to a place where no object yet existed. The new value therefore had to be  
24 constructed into the `std::vector`. If we revise the example such that the new  
25 `std::string` goes into a location already occupied by an object, it's a different  
26 story. Consider:

```
27 std::vector<std::string> vs;           // as before
28 ...
29 vs.emplace(v.begin(), "xyzzy");        // add "xyzzy" to
30                                         // beginning of vs
```

1 For this code, few implementations will construct the added `std::string` in-  
2 to the memory occupied by `vs[0]`. Instead, they'll move-assign the value into  
3 place. But move assignment requires an object to move from, and that means  
4 that a temporary object will need to be created. Because the primary ad-  
5 vantage of emplacement over insertion is that temporary objects are neither  
6 created nor destroyed, when the value being added is put into the container  
7 via assignment, emplacement's edge tends to disappear.

8 Unfortunately, whether adding a value to a container is accomplished by con-  
9 struction or assignment is generally up to the implementer. But, again, heuris-  
10 tics can help. Node-based containers virtually always use construction to add  
11 new values, and most standard containers are node-based. The only ones that  
12 aren't are `std::vector`, `std::deque`, and `std::string`. (`std::array` isn't,  
13 either, but it doesn't support insertion or emplacement, so it's not relevant  
14 here.) Within the non-node-based containers, you can rely on `emplace_back`  
15 to use construction instead of assignment to get a new value into place, and for  
16 `std::deque`, the same is true of `emplace_front`.

17 • **The argument type(s) being passed differ from the type held by the con-**  
18 **tainer.** Again, emplacement's advantage over insertion generally stems from  
19 the fact that its interface doesn't require creation and destruction of a tempo-  
20 rary object when the argument(s) passed are of a type other than that held by  
21 the container. When an object of type `T` is to be added to a `container<T>`,  
22 there's no reason to expect emplacement to run faster than insertion, because  
23 no temporary needs to be created to satisfy the insertion interface.

24 • **The container is unlikely to reject the new value as a duplicate.** This  
25 means that the container either permits duplicates or that most of the values  
26 you add will be unique. The reason this matters is that in order to detect  
27 whether a value is already in the container, emplacement implementations  
28 typically create a node with the new value so that they can compare the value  
29 of this node with existing container nodes. If the value to be added isn't in the  
30 container, the node is linked in. However, if the value is already present, the  
31 emplacement is aborted and the node is destroyed, meaning that the cost of its

1 creation and destruction was wasted. Such nodes are created for emplacement  
2 functions more often than for insertion functions.

3 The following calls from earlier in this Item satisfy all the criteria above. They also  
4 run faster than the corresponding calls to `push_back`.

```
5 vs.emplace_back("xyzzy"); // construct new value at end of
6 // container; don't pass the type in
7 // container; don't use container
8 // rejecting duplicates
9 vs.emplace_back(50, 'x'); // ditto
```

10 When deciding whether to use emplacement functions, two other issues are worth  
11 keeping in mind. The first regards resource management. Suppose you have a con-  
12 tainer of `std::shared_ptr<Widget>`s,

```
13 std::list<std::shared_ptr<Widget>> ptrs;
14 and you want to add a std::shared_ptr that should be released via a custom
15 deleter (see Item 19). Item 21 explains that you should use std::make_shared to
16 create std::shared_ptrs whenever you can, but it also concedes that there are
17 situations where you can't. One such situation is when you want to specify a cus-  
18 tom deleter. In that case, you must use new directly to get the raw pointer to be
19 managed by the std::shared_ptr.
```

20 If the custom deleter is this function,

```
21 void killWidget(Widget* pWidget);
```

22 the code using an insertion function could look like this:

```
23 ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

24 It could also look like this, though the meaning would be the same:

```
25 ptrs.push_back( { new Widget, killWidget } );
```

26 Either way, a temporary `std::shared_ptr` would be constructed before calling  
27 `push_back`. `push_back`'s parameter is a reference to a `std::shared_ptr`, so  
28 there has to be a `std::shared_ptr` for this parameter to refer to.

1 The creation of the temporary `std::shared_ptr` is what `emplace_back` would  
2 avoid, but in this case, that temporary is worth far more than it costs. Consider the  
3 following potential sequence of events:

4 1. In either call above, a temporary `std::shared_ptr<Widget>` object is con-  
5 structed to hold the raw pointer resulting from “`new Widget`”. Call this object  
6 *temp*.

7 2. `push_back` takes *temp* by reference. During allocation of a list node to hold a  
8 copy of *temp*, an out-of-memory exception gets thrown.

9 3. As the exception propagates out of `push_back`, *temp* is destroyed. Being the  
10 sole `std::shared_ptr` referring to the `Widget` it’s managing, it automatically  
11 releases that `Widget`, in this case by calling `killWidget`.

12 Even though an exception occurred, nothing leaks: the `Widget` created via “`new`  
13 `Widget`” in the call to `push_back` is released in the destructor of the  
14 `std::shared_ptr` that was created to manage it (*temp*). Life is good.

15 But now consider what happens if `emplace_back` is called instead of `push_back`:

16 `ptrs.emplace_back(new Widget, killWidget);`

17 1. The raw pointer resulting from “`new Widget`” is perfect-forwarded to the point  
18 inside `emplace_back` where a list node is to be allocated. That allocation fails,  
19 and an out-of-memory exception is thrown.

20 2. As the exception propagates out of `emplace_back`, the raw pointer that was  
21 the only way to get at the `Widget` on the heap is lost. That `Widget` (and any re-  
22 sources it owns) is leaked.

23 In this scenario, life is *not* good, and the fault doesn’t lie with `std::shared_ptr`.  
24 The same kind of problem can arise through the use of `std::unique_ptr` with a  
25 custom deleter. Fundamentally, the effectiveness of resource-managing classes like  
26 `std::shared_ptr` and `std::unique_ptr` is predicated on resources (such as  
27 raw pointers from `new`) being *immediately* passed to constructors for resource-  
28 managing objects. The fact that functions like `std::make_shared` and  
29 `std::make_unique` automate this is one of the reasons they’re so important.

1 In calls to the insertion functions of containers holding resource-managing objects  
2 (e.g., `std::list<std::shared_ptr<Widget>>`), the functions' parameter types  
3 generally ensure that nothing gets between acquisition of a resource (e.g., use of  
4 `new`) and construction of the object managing the resource. In the emplacement  
5 functions, perfect-forwarding defers the creation of the resource-managing objects  
6 until they can be constructed in the container's memory, and that opens a window  
7 during which exceptions can lead to resource leaks. All standard containers are  
8 susceptible to this problem. When working with containers of resource-managing  
9 objects, you must take care to ensure that if you choose an emplacement function  
10 over its insertion counterpart, you're not paying for improved code efficiency with  
11 diminished exception safety.

12 Frankly, you shouldn't be passing expressions like "`new Widget`" to `em-`  
13 `place_back` or `push_back` or most any other function, anyway, because, as  
14 Item 21 explains, this leads to the possibility of exception safety problems of the  
15 kind we just examined. Closing the door requires taking the pointer from "`new`  
16 `Widget`" and turning it over to a resource-managing object in a standalone state-  
17 ment, then passing that object as an rvalue to the function you originally wanted to  
18 pass "`new Widget`" to. (Item 21 covers this technique in more detail.) The code us-  
19 ing `push_back` should therefore be written more like this:

```
20 std::shared_ptr<Widget> spw(new Widget); // create Widget and
21 // have spw manage it
22 ptrs.push_back(std::move(spw)); // add spw as rvalue
```

23 The `emplace_back` version is similar:

```
24 std::shared_ptr<Widget> spw(new Widget);
25 ptrs.emplace_back(std::move(spw));
```

26 Either way, the approach incurs the cost of creating and destroying `spw`. Given that  
27 the motivation for choosing emplacement over insertion is to avoid the cost of a  
28 temporary object of the type held by the container, yet that's conceptually what  
29 `spw` is, emplacement functions are unlikely to outperform insertion functions  
30 when you're adding resource-managing objects to a container and you follow the  
31 proper practice of ensuring that nothing can intervene between acquiring a re-  
32 source and turning it over to a resource-managing object.

1 A second noteworthy aspect of emplacement functions is their interaction with  
2 explicit constructors. In honor of C++11's support for regular expressions, sup-  
3 pose you create a container of regular expression objects:

4 `std::vector<std::regex> regexes;`  
5 Distracted by your colleagues' quarreling over the ideal number of times per day  
6 to check one's Facebook account, you accidentally write the following seemingly  
7 meaningless code:

8 `regexes.emplace_back(nullptr); // add nullptr to container`  
9 `// of regexes?`

10 You don't notice the error as you type it, and your compilers accept the code with-  
11 out complaint, so you end up wasting a bunch of time debugging. At some point,  
12 you discover that you appear to have inserted a null pointer into your container of  
13 regular expressions. But how is that possible? Pointers aren't regular expressions,  
14 and if you tried to do something like this,

15 `std::regex r = nullptr; // error! won't compile`  
16 compilers would reject your code. Interestingly, they would also reject it if you  
17 called `push_back` instead of `emplace_back`:

18 `regexes.push_back(nullptr); // error! won't compile`  
19 The curious behavior you're experiencing stems from the fact that `std::regex`  
20 objects can be constructed from character strings. That's what makes useful code  
21 like this legal:

22 `std::regex upperCaseWord("[A-Z]+");`  
23 Creation of a `std::regex` from a character string can exact a comparatively large  
24 runtime cost, so, to minimize the likelihood that such an expense will be incurred  
25 unintentionally, the `std::regex` constructor taking a `const char*` pointer is ex-  
26 plicit. That's why these lines don't compile:

27 `std::regex r = nullptr; // error! won't compile`  
28 `regexes.push_back(nullptr); // error! won't compile`

1 In both cases, we're requesting an implicit conversion from a pointer to a  
2 `std::regex`, and the `explicitness` of that constructor prevents such conver-  
3 sions.

4 In the call to `emplace_back`, however, we're not claiming to pass a `std::regex`  
5 object. Instead, we're passing a *constructor argument* for a `std::regex` object.  
6 That's not considered an implicit conversion request. Rather, it's viewed as if you'd  
7 written this code:

8 `std::regex r(nullptr); // compiles`

9 If the laconic comment "compiles" suggests a lack of enthusiasm, that's good, be-  
10 cause this code, though it will compile, has undefined behavior. The `std::regex`  
11 constructor taking a `const char*` pointer requires that the pointed-to string com-  
12 prise a valid regular expression, and the null pointer fails that requirement. If you  
13 write and compile such code, the best you can hope for is that it crashes at run  
14 time. If you're not so lucky, you and your debugger could be in for a special bond-  
15 ing experience.

16 Setting aside `push_back`, `emplace_back`, and bonding for a moment, notice how  
17 these very similar initialization syntaxes yield different results:

18 `std::regex r1 = nullptr; // error! won't compile`

19 `std::regex r2(nullptr); // compiles`

20 In the official terminology of the Standard, the syntax used to initialize `r1` (employ-  
21 ing the equals-sign) corresponds to what is known as *copy initialization*. In con-  
22 trast, the syntax used to initialize `r2` (with the parentheses, although braces may  
23 be used instead) yields what is called *direct initialization*. Copy initialization is not  
24 permitted to use `explicit` constructors. Direct initialization is. That's why the  
25 line initializing `r1` doesn't compile, but the line initializing `r2` does.

26 But back to `push_back` and `emplace_back` and, more generally, the insertion  
27 functions versus the emplacement functions. Emplacement functions use direct  
28 initialization, which means they may use `explicit` constructors. Insertion func-  
29 tions employ copy initialization, so they can't. Hence:

```
1 regexes.emplace_back(nullptr); // fine, direct init permits
2                                     // use of explicit std::regex
3                                     // ctor taking a pointer
4 regexes.push_back(nullptr);      // error! copy init forbids
5                                     // use of that ctor
6 The lesson to take away is that when you use an emplacement function, be espe-
7 cially careful to make sure you're passing the correct arguments, because even ex-
8 plicit constructors will be considered by compilers as they try to find a way to
9 interpret your code as valid.
```

## 10 Things to Remember

- 11     • In principle, emplacement functions should sometimes be more efficient than  
12        their insertion counterparts, and they should never be less efficient.
- 13     • In practice, they're most likely to be faster when (1) the value being added is  
14        constructed into the container, not assigned; (2) the argument type(s) passed  
15        differ from the type held by the container; and (3) the container won't reject  
16        the value being added due to it being a duplicate.
- 17     • Emplacement functions may perform type conversions that would be rejected  
18        by insertion functions.