```
  struct Impl;
  std::unique_ptr<Impl> pImpl;      // use smart pointer
};                                  // instead of raw pointer
```

and this for the implementation file:

```
#include "widget.h"                 // in "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

struct Widget::Impl {               // as before
  std::string name;
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                    // per Item 21, create
: pImpl(std::make_unique<Impl>())   // std::unique_ptr
{}                                  // via std::make_unique
```

You'll note that the `Widget` destructor is no longer present. That's because we have no code to put into it. `std::unique_ptr` automatically deletes what it points to when it (the `std::unique_ptr`) is destroyed, so we need not delete anything ourselves. That's one of the attractions of smart pointers: they eliminate the need for us to sully our hands with manual resource release.

This code compiles, but, alas, the most trivial client use doesn't:

```
#include "widget.h"

Widget w;                           // error!
```

The error message you receive depends on the compiler you're using, but the text generally mentions something about applying `sizeof` or `delete` to an incomplete type. Those operations aren't among the things you can do with such types.

This apparent failure of the Pimpl Idiom using `std::unique_ptr`s is alarming, because (1) `std::unique_ptr` is advertised as supporting incomplete types, and (2) the Pimpl Idiom is one of `std::unique_ptr`s most common use cases. Fortunately, getting the code to work is easy. All that's required is a basic understanding of the cause of the problem.

The issue arises due to the code that's generated when `w` is destroyed (e.g., goes out of scope). At that point, its destructor is called. In the class definition using `std::unique_ptr`, we didn't declare a destructor, because we didn't have any code to put into it. In accord with the usual rules for compiler-generated special member functions (see Item 17), the compiler generates a destructor for us. Within that destructor, the compiler inserts code to call the destructor for `Widget`'s data member `pImpl`. `pImpl` is a `std::unique_ptr<Widget::Impl>`, i.e., a `std::unique_ptr` using the default deleter. The default deleter is a function that uses `delete` on the raw pointer inside the `std::unique_ptr`. Prior to using `delete`, however, implementations typically have the default deleter employ C++11's `static_assert` to ensure that the raw pointer doesn't point to an incomplete type. When the compiler generates code for the destruction of the `Widget w`, then, it generally encounters a `static_assert` that fails, and that's usually what leads to the error message. This message is associated with the point where `w` is destroyed, because `Widget`'s destructor, like all compiler-generated special member functions, is implicitly `inline`. The message itself often refers to the line where `w` is created, because it's the source code explicitly creating the object that leads to its later implicit destruction.

To fix the problem, you just need to make sure that at the point where the code to destroy the `std::unique_ptr<Widget::Impl>` is generated, `Widget::Impl` is a complete type. The type becomes complete when its definition has been seen, and `Widget::Impl` is defined inside `widget.cpp`. The key to successful compilation, then, is to have the compiler see the body of `Widget`'s destructor (i.e., the place where the compiler will generate code to destroy the `std::unique_ptr` data member) only inside `widget.cpp` after `Widget::Impl` has been defined.

Arranging for that is simple. Declare `Widget`'s destructor in `widget.h`, but don't define it there:

```
class Widget {                     // as before, in "widget.h"
public:
  Widget();
  ~Widget();                       // declaration only
  …

private:                           // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

Define it in `widget.cpp` after `Widget::Impl` has been defined:

```
#include "widget.h"                // as before, in "widget.cpp"
#include "gadget.h"
#include <string>
#include <vector>

  struct Widget::Impl {            // as before, definition of
```

```
  std::string name;                // Widget::Impl
  std::vector<double> data;
  Gadget g1, g2, g3;
};

Widget::Widget()                   // as before
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget()                  // ~Widget definition
{}
```

This works well, and it requires the least typing, but if you want to emphasize that the compiler-generated destructor would do the right thing—that the only reason you declared it was to cause its definition to be generated in `Widget`'s implementation file, you can define the destructor body with "`= default`":

```
Widget::~Widget() = default;       // same effect as above
```

Classes using the Pimpl Idiom are natural candidates for move support, because compiler-generated move operations do exactly what's desired: perform a move on the underlying `std::unique_ptr`. As Item 17 explains, the declaration of a destructor in `Widget` prevents compilers from generating the move operations, so if you want move support, you must declare the functions yourself. Given that the compiler-generated versions would behave correctly, you're likely to be tempted to implement them as follows:

```
class Widget {                                  // still in
public:                                         // "widget.h"
  Widget();
  ~Widget();

  Widget(Widget&& rhs) = default;               // right idea,
  Widget& operator=(Widget&& rhs) = default;    // wrong code!

  …

private:                                        // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};
```

This approach leads to the same kind of problem as declaring the class without a destructor, and for the same fundamental reason. The compiler-generated move assignment operator needs to destroy the object pointed to by `pImpl` before reassigning it, but in the `Widget` header file, `pImpl` points to an incomplete type. The situation is different for the move constructor. The problem there is that compilers typically generate code to destroy `pImpl` in the event that an exception arises inside 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:

```
class Widget {                          // still in "widget.h"
public:
  Widget();
  ~Widget();

  Widget(Widget&& rhs);                 // declarations
  Widget& operator=(Widget&& rhs);      // only

  …

private:                                // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};

#include <string>                       // as before,
…                                       // in "widget.cpp"

struct Widget::Impl { … };              // as before

Widget::Widget()                        // as before
: pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;            // as before

Widget::Widget(Widget&& rhs) = default;             // defini-
Widget& Widget::operator=(Widget&& rhs) = default;  // tions
```

The Pimpl Idiom is a way to reduce compilation dependencies between a class's implementation and the class's clients, but, conceptually, use of the idiom doesn't change what the class represents. The original `Widget` class contained `std::string`, `std::vector`, and `Gadget` data members, and, assuming that `Gadget`s, like `std::string`s and `std::vector`s, can be copied, it would make sense for `Widget` to support the copy operations. We have to write these functions ourselves, because (1) compilers won't generate copy operations for classes with move-only types like `std::unique_ptr` and (2) even if they did, the generated functions would copy only the `std::unique_ptr` (i.e., perform a *shallow copy*), and we want to copy what the pointer points to (i.e., perform a *deep copy*).

In a ritual that is by now familiar, we declare the functions in the header file and implement them in the implementation file:

```
class Widget {                              // still in "widget.h"
public:
  …                                         // other funcs, as before

  Widget(const Widget& rhs);                   // declarations
  Widget& operator=(const Widget& rhs);        // only

private:                                       // as before
  struct Impl;
  std::unique_ptr<Impl> pImpl;
};


#include "widget.h"                       // as before,
…                                         // in "widget.cpp"

struct Widget::Impl { … };                // as before

Widget::~Widget() = default;              // other funcs, as before

Widget::Widget(const Widget& rhs)                  // copy ctor
: pImpl(std::make_unique<Impl>(*rhs.pImpl))
{}

Widget& Widget::operator=(const Widget& rhs)    // copy operator=
{
  *pImpl = *rhs.pImpl;
  return *this;
}
```

Both function implementations are conventional. In each case, we simply copy the fields of the `Impl` struct from the source object (`rhs`) to the destination object (`*this`). Rather than copy the fields one by one, we take advantage of the fact that compilers will create the copy operations for `Impl`, and these operations will copy each field automatically. We thus implement `Widget`'s copy operations by calling `Widget::Impl`'s compiler-generated copy operations. In the copy constructor, note that we still follow the advice of Item 21 to prefer use of `std::make_unique` over direct use of `new`.

For purposes of implementing the Pimpl Idiom, std::unique_ptr is the smart pointer to use, because the pImpl pointer inside an object (e.g., inside a Widget) has exclusive ownership of the corresponding implementation object (e.g., the Widget::Impl object). Still, it's interesting to note that if we were to use std::shared_ptr instead of std::unique_ptr for pImpl, we'd find that the advice of this Item no longer applied. There'd be no need to declare a destructor in Widget, and without a user-declared destructor, compilers would happily generate the move operations, which would do exactly what we'd want them to. That is, given this code in widget.h,

```
class Widget {                      // in "widget.h"
public:
  Widget();
  …                                 // no declarations for dtor
                                    // or move operations
private:
  struct Impl;
  std::shared_ptr<Impl> pImpl;      // std::shared_ptr
};                                  // instead of std::unique_ptr
```

and this client code that #includes widget.h,

```
Widget w1;

auto w2(std::move(w1));             // move-construct w2

w1 = std::move(w2);                 // move-assign w1
```

everything would compile and run as we'd hope: w1 would be default constructed, its value would be moved into w2, that value would be moved back into w1, and then both w1 and w2 would be destroyed (thus causing the pointed-to Widget::Impl object to be destroyed).

The difference in behavior between std::unique_ptr and std::shared_ptr for pImpl pointers stems from the differing ways these smart pointers support custom deleters. For std::unique_ptr, the type of the deleter is part of the type of the smart pointer, and this makes it possible for compilers to generate smaller runtime data structures and faster runtime code. A consequence of this greater efficiency is that pointed-to types must be complete when compiler-generated special functions (e.g., destructors or move operations) are used. For std::shared_ptr, the type of the deleter is not part of the type of the smart pointer. This necessitates larger runtime data structures and somewhat slower code, but pointed-to types need not be complete when compiler-generated special functions are employed.

For the Pimpl Idiom, there's not really a trade-off between the characteristics of std::unique_ptr and std::shared_ptr, because the relationship between classes like Widget and classes like Widget::Impl is exclusive ownership, and that makes std::unique_ptr the proper tool for the job. Nevertheless, it's worth knowing that in other situations—situations where shared ownership exists (and std::shared_ptr is hence a fitting design choice), there's no need to jump through the function-definition hoops that use of std::unique_ptr entails.

Things to Remember

- The Pimpl Idiom decreases build times by reducing compilation dependencies between class clients and class implementations.
- For std::unique_ptr pImpl pointers, declare special member functions in the class header, but implement them in the implementation file. Do this even if the default function implementations are acceptable.
- The above advice applies to std::unique_ptr, but not to std::shared_ptr.

CHAPTER 5

Rvalue References, Move Semantics, and Perfect Forwarding

When you first learn about them, move semantics and perfect forwarding seem pretty straightforward:

- **Move semantics** makes it possible for compilers to replace expensive copying operations with less expensive moves. In the same way that copy constructors and copy assignment operators give you control over what it means to copy objects, move constructors and move assignment operators offer control over the semantics of moving. Move semantics also enables the creation of move-only types, such as `std::unique_ptr`, `std::future`, and `std::thread`.
- **Perfect forwarding** makes it possible to write function templates that take arbitrary arguments and forward them to other functions such that the target functions receive exactly the same arguments as were passed to the forwarding functions.

Rvalue references are the glue that ties these two rather disparate features together. They're the underlying language mechanism that makes both move semantics and perfect forwarding possible.

The more experience you have with these features, the more you realize that your initial impression was based on only the metaphorical tip of the proverbial iceberg. The world of move semantics, perfect forwarding, and rvalue references is more nuanced than it appears. `std::move` doesn't move anything, for example, and perfect forwarding is imperfect. Move operations aren't always cheaper than copying; when they are, they're not always as cheap as you'd expect; and they're not always called in a context where moving is valid. The construct "*`type`*`&&`" doesn't always represent an rvalue reference.

No matter how far you dig into these features, it can seem that there's always more to uncover. Fortunately, there is a limit to their depths. This chapter will take you to the bedrock. Once you arrive, this part of C++11 will make a lot more sense. 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 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.

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

```
void f(Widget&& w);
```

the parameter `w` is an lvalue, even though its type is rvalue-reference-to-`Widget`. (If this surprises you, please review the overview of lvalues and rvalues that begins on page 2.)

Item 23: Understand `std::move` and `std::forward`.

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. At runtime, neither does anything at all. They generate no executable code. Not a single byte.

`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. That's it. The explanation leads to a new set of questions, but, fundamentally, that's the complete story.

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.

```
template<typename T>                        // in namespace std
typename remove_reference<T>::type&&
move(T&& param)
{
  using ReturnType =                        // alias declaration;
    typename remove_reference<T>::type&&;   // see Item 9

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

I've highlighted two parts of the code for you. One is the name of the function, because the return type specification is rather noisy, and I don't want you to lose your bearings in the din. The other is the cast that comprises the essence of the function. As you can see, `std::move` takes a reference to an object (a universal reference, to be precise—see Item 24) and it returns a reference to the same object.

The "`&&`" part of the function's return type implies that `std::move` returns an rvalue reference, but, as Item 28 explains, if the type `T` happens to be an lvalue reference, `T&&` would become an lvalue reference. To prevent this from happening, the type trait (see Item 9) `std::remove_reference` is applied to `T`, thus ensuring that "`&&`" is applied to a type that isn't a reference. That guarantees that `std::move` truly returns an rvalue reference, and that's important, because rvalue references returned from functions are rvalues. Thus, `std::move` casts its argument to an rvalue, and that's all it does.

As an aside, `std::move` can be implemented with less fuss in C++14. Thanks to function return type deduction (see Item 3) and to the Standard Library's alias template `std::remove_reference_t` (see Item 9), `std::move` can be written this way:

```
template<typename T>                          // C++14; still in
decltype(auto) move(T&& param)                // namespace std
{
  using ReturnType = remove_reference_t<T>&&;
  return static_cast<ReturnType>(param);
}
```

Easier on the eyes, no?

Because `std::move` does nothing but cast its argument to an rvalue, there have been suggestions that a better name for it might have been something like `rvalue_cast`. Be that as it may, the name we have is `std::move`, so it's important to remember what `std::move` does and doesn't do. It does cast. It doesn't move.

Of course, rvalues are candidates for moving, so applying `std::move` to an object tells the compiler that the object is eligible to be moved from. That's why `std::move` has the name it does: to make it easy to designate objects that may be moved from.

In truth, rvalues are only *usually* candidates for moving. Suppose you're writing a class representing annotations. The class's constructor takes a `std::string` parameter comprising the annotation, and it copies the parameter to a data member. Flush with the information in Item 41, you declare a by-value parameter:

```
class Annotation {
public:
  explicit Annotation(std::string text);  // param to be copied,
```

```
  …                                         // so per Item 41,
};                                          // pass by value
```

But `Annotation`'s constructor needs only to read `text`'s value. It doesn't need to modify it. In accord with the time-honored tradition of using `const` whenever possible, you revise your declaration such that `text` is `const`:

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  …
};
```

To avoid paying for a copy operation when copying `text` into a data member, you remain true to the advice of Item 41 and apply `std::move` to `text`, thus producing an rvalue:

```
class Annotation {
public:
  explicit Annotation(const std::string text)
  : value(std::move(text))  // "move" text into value; this code
  { … }                     // doesn't do what it seems to!

  …

private:
  std::string value;
};
```

This code compiles. This code links. This code runs. This code sets the data member `value` to the content of `text`. The only thing separating this code from a perfect realization of your vision is that `text` is not moved into `value`, it's *copied*. Sure, `text` is cast to an rvalue by `std::move`, but `text` is declared to be a `const std::string`, so before the cast, `text` is an lvalue `const std::string`, and the result of the cast is an rvalue `const std::string`, but throughout it all, the `const`ness remains.

Consider the effect that has when compilers have to determine which `std::string` constructor to call. There are two possibilities:

```
class string {                // std::string is actually a
public:                       // typedef for std::basic_string<char>
  …
  string(const string& rhs);    // copy ctor
  string(string&& rhs);         // move ctor
  …
};
```

In the `Annotation` constructor's member initialization list, the result of `std::move(text)` is an rvalue of type `const std::string`. That rvalue can't be passed to `std::string`'s move constructor, because the move constructor takes an rvalue reference to a *non-`const`* `std::string`. The rvalue can, however, be passed to the copy constructor, because an lvalue-reference-to-`const` is permitted to bind to a `const` rvalue. The member initialization therefore invokes the *copy* constructor in `std::string`, even though `text` has been cast to an rvalue! Such behavior is essential to maintaining `const`-correctness. Moving a value out of an object generally modifies the object, so the language should not permit `const` objects to be passed to functions (such as move constructors) that could modify them.

There are two lessons to be drawn from this example. First, don't declare objects `const` if you want to be able to move from them. Move requests on `const` objects are silently transformed into copy operations. Second, `std::move` not only doesn't actually move anything, it doesn't even guarantee that the object it's casting will be eligible to be moved. The only thing you know for sure about the result of applying `std::move` to an object is that it's an rvalue.

The story for `std::forward` is similar to that for `std::move`, but whereas `std::move` *unconditionally* casts its argument to an rvalue, `std::forward` does it only under certain conditions. `std::forward` is a *conditional* cast. To understand when it casts and when it doesn't, recall how `std::forward` is typically used. The most common scenario is a function template taking a universal reference parameter that is to be passed to another function:

```
void process(const Widget& lvalArg);      // process lvalues
void process(Widget&& rvalArg);           // process rvalues

template<typename T>                      // template that passes
void logAndProcess(T&& param)             // param to process
{
  auto now =                              // get current time
    std::chrono::system_clock::now();

  makeLogEntry("Calling 'process'", now);
  process(std::forward<T>(param));
}
```

Consider two calls to `logAndProcess`, one with an lvalue, the other with an rvalue:

```
Widget w;

logAndProcess(w);               // call with lvalue
logAndProcess(std::move(w));    // call with rvalue
```

Inside `logAndProcess`, the parameter `param` is passed to the function `process`. `process` is overloaded for lvalues and rvalues. When we call `logAndProcess` with an lvalue, we naturally expect that lvalue to be forwarded to `process` as an lvalue, and when we call `logAndProcess` with an rvalue, we expect the rvalue overload of `process` to be invoked.

But `param`, like all function parameters, is an lvalue. Every call to `process` inside `logAndProcess` will thus want to invoke the lvalue overload for `process`. To prevent this, we need a mechanism for `param` to be cast to an rvalue if and only if the argument with which `param` was initialized—the argument passed to `logAndProcess`—was an rvalue. 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 was initialized with an rvalue.

You may wonder how `std::forward` can know whether its argument was initialized with an rvalue. In the code above, for example, how can `std::forward` tell whether `param` was initialized with an lvalue or an rvalue? The brief answer is that that information is encoded in `logAndProcess`'s template parameter `T`. That parameter is passed to `std::forward`, which recovers the encoded information. For details on exactly how that works, consult Item 28.

Given that both `std::move` and `std::forward` boil down to casts, the only difference being that `std::move` always casts, while `std::forward` only sometimes does, you might ask whether we can dispense with `std::move` and just use `std::forward` everywhere. From a purely technical perspective, the answer is yes: `std::forward` can do it all. `std::move` isn't necessary. Of course, neither function is really *necessary*, because we could write casts everywhere, but I hope we agree that that would be, well, yucky.

`std::move`'s attractions are convenience, reduced likelihood of error, and greater clarity. Consider a class where we want to track how many times the move constructor is called. A `static` counter that's incremented during move construction is all we need. Assuming the only non-static data in the class is a `std::string`, here's the conventional way (i.e., using `std::move`) to implement the move constructor:

```
class Widget {
public:
  Widget(Widget&& rhs)
  : s(std::move(rhs.s))
  { ++moveCtorCalls; }

  …
```

```
private:
  static std::size_t moveCtorCalls;
  std::string s;
};
```

To implement the same behavior with `std::forward`, the code would look like this:

```
class Widget {
public:
  Widget(Widget&& rhs)                          // unconventional,
  : s(std::forward<std::string>(rhs.s))         // undesirable
  { ++moveCtorCalls; }                          // implementation

  …

};
```

Note first that `std::move` requires only a function argument (`rhs.s`), while `std::forward` requires both a function argument (`rhs.s`) and a template type argument (`std::string`). Then note that the type we pass to `std::forward` should be a non-reference, because that's the convention for encoding that the argument being passed is an rvalue (see Item 28). Together, this means that `std::move` requires less typing than `std::forward`, and it spares us the trouble of passing a type argument that encodes that the argument we're passing is an rvalue. It also eliminates the possibility of our passing an incorrect type (e.g., `std::string&`, which would result in the data member `s` being copy constructed instead of move constructed).

More importantly, the use of `std::move` conveys an unconditional cast to an rvalue, while the use of `std::forward` indicates a cast to an rvalue only for references to which rvalues have been bound. Those are two very different actions. The first one typically sets up a move, while the second one just passes—*forwards*—an object to another function in a way that retains its original lvalueness or rvalueness. Because these actions are so different, it's good that we have two different functions (and function names) to distinguish them.

Things to Remember

- `std::move` performs an unconditional cast to an rvalue. In and of itself, it doesn't move anything.
- `std::forward` casts its argument to an rvalue only if that argument is bound to an rvalue.
- Neither `std::move` nor `std::forward` do anything at runtime.

Item 24: Distinguish universal references from rvalue references.

It's been said that the truth shall set you free, but under the right circumstances, a well-chosen lie can be equally liberating. This Item is such a lie. Because we're dealing with software, however, let's eschew the word "lie" and instead say that this Item comprises an "abstraction."

To declare an rvalue reference to some type `T`, you write `T&&`. It thus seems reasonable to assume that if you see "`T&&`" in source code, you're looking at an rvalue reference. Alas, it's not quite that simple:

```
void f(Widget&& param);            // rvalue reference

Widget&& var1 = Widget();          // rvalue reference

auto&& var2 = var1;                // not rvalue reference

template<typename T>
void f(std::vector<T>&& param);    // rvalue reference

template<typename T>
void f(T&& param);                 // not rvalue reference
```

In fact, "`T&&`" has two different meanings. One is rvalue reference, of course. Such references behave exactly the way you expect: they bind only to rvalues, and their primary *raison d'être* is to identify objects that may be moved from.

The other meaning for "`T&&`" is *either* rvalue reference *or* lvalue reference. Such references look like rvalue references in the source code (i.e., "`T&&`"), but they can behave as if they were lvalue references (i.e., "`T&`"). Their dual nature permits them to bind to rvalues (like rvalue references) as well as lvalues (like lvalue references). Furthermore, they can bind to `const` or non-`const` objects, to `volatile` or non-`volatile` objects, even to objects that are both `const` and `volatile`. They can bind to virtually *anything*. Such unprecedentedly flexible references deserve a name of their own. I call them *universal references*.[1]

Universal references arise in two contexts. The most common is function template parameters, such as this example from the sample code above:

1 Item 25 explains that universal references should almost always have `std::forward` applied to them, and as this book goes to press, some members of the C++ community have started referring to universal references as *forwarding references*.

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

The second context is `auto` declarations, including this one from the sample code above:

```
auto&& var2 = var1;             // var2 is a universal reference
```

What these contexts have in common is the presence of *type deduction*. In the template `f`, the type of `param` is being deduced, and in the declaration for `var2`, `var2`'s type is being deduced. Compare that with the following examples (also from the sample code above), where type deduction is missing. If you see "`T&&`" without type deduction, you're looking at an rvalue reference:

```
void f(Widget&& param);         // no type deduction;
                                // param is an rvalue reference

Widget&& var1 = Widget();       // no type deduction;
                                // var1 is an rvalue reference
```

Because universal references are references, they must be initialized. The initializer for a universal reference determines whether it represents an rvalue reference or an lvalue reference. If the initializer is an rvalue, the universal reference corresponds to an rvalue reference. If the initializer is an lvalue, the universal reference corresponds to an lvalue reference. For universal references that are function parameters, the initializer is provided at the call site:

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

Widget w;
f(w);                   // lvalue passed to f; param's type is
                        // Widget& (i.e., an lvalue reference)

f(std::move(w));        // rvalue passed to f; param's type is
                        // Widget&& (i.e., an rvalue reference)
```

For a reference to be universal, type deduction is necessary, but it's not sufficient. The *form* of the reference declaration must also be correct, and that form is quite constrained. It must be precisely "`T&&`". Look again at this example from the sample code we saw earlier:

```
template<typename T>
void f(std::vector<T>&& param);  // param is an rvalue reference
```

When `f` is invoked, the type `T` will be deduced (unless the caller explicitly specifies it, an edge case we'll not concern ourselves with). But the form of `param`'s type declaration isn't "T&&", it's "std::vector<T>&&". That rules out the possibility that param is a universal reference. param is therefore an rvalue reference, something that your compilers will be happy to confirm for you if you try to pass an lvalue to f:

```
std::vector<int> v;
f(v);                              // error! can't bind lvalue to
                                   // rvalue reference
```

Even the simple presence of a const qualifier is enough to disqualify a reference from being universal:

```
template<typename T>
void f(const T&& param);          // param is an rvalue reference
```

If you're in a template and you see a function parameter of type "T&&", you might think you can assume that it's a universal reference. You can't. That's because being in a template doesn't guarantee the presence of type deduction. Consider this push_back member function in std::vector:

```
template<class T, class Allocator = allocator<T>>  // from C++
class vector {                                     // Standards
public:
  void push_back(T&& x);
  …
};
```

push_back's parameter certainly has the right form for a universal reference, but there's no type deduction in this case. That's because push_back can't exist without a particular vector instantiation for it to be part of, and the type of that instantiation fully determines the declaration for push_back. That is, saying

```
std::vector<Widget> v;
```

causes the std::vector template to be instantiated as follows:

```
class vector<Widget, allocator<Widget>> {
public:
  void push_back(Widget&& x);             // rvalue reference
  …
};
```

Now you can see clearly that push_back employs no type deduction. This push_back for vector<T> (there are two—the function is overloaded) always declares a parameter of type rvalue-reference-to-T.

In contrast, the conceptually similar emplace_back member function in std::vector *does* employ type deduction:

```
template<class T, class Allocator = allocator<T>>  // still from
class vector {                                     // C++
public:                                            // Standards
  template <class... Args>
  void emplace_back(Args&&... args);
  …
};
```

Here, the type parameter `Args` is independent of `vector`'s type parameter `T`, so `Args` must be deduced each time `emplace_back` is called. (Okay, `Args` is really a parameter pack, not a type parameter, but for purposes of this discussion, we can treat it as if it were a type parameter.)

The fact that `emplace_back`'s type parameter is named `Args`, yet it's still a universal reference, reinforces my earlier comment that it's the *form* of a universal reference that must be "`T&&`". There's no requirement that you use the name `T`. For example, the following template takes a universal reference, because the form ("*`type`*`&&`") is right, and `param`'s type will be deduced (again, excluding the corner case where the caller explicitly specifies the type):

```
template<typename MyTemplateType>          // param is a
void someFunc(MyTemplateType&& param);     // universal reference
```

I remarked earlier that `auto` variables can also be universal references. To be more precise, variables declared with the type `auto&&` are universal references, because type deduction takes place and they have the correct form ("`T&&`"). `auto` universal references are not as common as universal references used for function template parameters, but they do crop up from time to time in C++11. They crop up a lot more in C++14, because C++14 lambda expressions may declare `auto&&` parameters. For example, if you wanted to write a C++14 lambda to record the time taken in an arbitrary function invocation, you could do this:

```
auto timeFuncInvocation =
  [](auto&& func, auto&&... params)               // C++14
  {
    start timer;
    std::forward<decltype(func)>(func)(           // invoke func
      std::forward<decltype(params)>(params)...   // on params
      );
    stop timer and record elapsed time;
  };
```

If your reaction to the "`std::forward<decltype(`*`blah blah blah`*`)>`" code inside the lambda is, "What the…?!", that probably just means you haven't yet read Item 33. Don't worry about it. The important thing in this Item is the `auto&&` parameters that the lambda declares. `func` is a universal reference that can be bound to any callable object, lvalue or rvalue. `args` is zero or more universal references (i.e., a universal reference parameter pack) that can be bound to any number of objects of arbitrary types. The result, thanks to `auto` universal references, is that `timeFuncInvocation` can time pretty much any function execution. (For information on the difference between "any" and "pretty much any," turn to Item 30.)

Bear in mind that this entire Item—the foundation of universal references—is a lie… er, an "abstraction." The underlying truth is known as *reference collapsing*, a topic to which Item 28 is dedicated. But the truth doesn't make the abstraction any less useful. Distinguishing between rvalue references and universal references will help you read source code more accurately ("Does that `T&&` I'm looking at bind to rvalues only or to everything?"), and it will avoid ambiguities when you communicate with your colleagues ("I'm using a universal reference here, not an rvalue reference…"). It will also allow you to make sense of Items 25 and 26, which rely on the distinction. So embrace the abstraction. Revel in it. Just as Newton's laws of motion (which are technically incorrect) are typically just as useful as and easier to apply than Einstein's theory of general relativity ("the truth"), so is the notion of universal references normally preferable to working through the details of reference collapsing.

Things to Remember

- If a function template parameter has type `T&&` for a deduced type `T`, or if an object is declared using `auto&&`, the parameter or object is a universal reference.
- If the form of the type declaration isn't precisely *`type`*`&&`, or if type deduction does not occur, *`type`*`&&` denotes an rvalue reference.
- Universal references correspond to rvalue references if they're initialized with rvalues. They correspond to lvalue references if they're initialized with lvalues.

Item 25: Use `std::move` on rvalue references, `std::forward` on universal references.

Rvalue references bind only to objects that are candidates for moving. If you have an rvalue reference parameter, you *know* that the object it's bound to may be moved:

```
class Widget {
  Widget(Widget&& rhs);        // rhs definitely refers to an
```

```
  …                             // object eligible for moving
};
```

That being the case, you'll want to pass such objects to other functions in a way that permits those functions to take advantage of the object's rvalueness. The way to do that is to cast parameters bound to such objects to rvalues. As Item 23 explains, that's not only what `std::move` does, it's what it was created for:

```
class Widget {
public:
  Widget(Widget&& rhs)              // rhs is rvalue reference
  : name(std::move(rhs.name)),
    p(std::move(rhs.p))
    { … }
  …

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};
```

A universal reference, on the other hand (see Item 24), *might* be bound to an object that's eligible for moving. Universal references should be cast to rvalues only if they were initialized with rvalues. Item 23 explains that this is precisely what `std::forward` does:

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)             // newName is
  { name = std::forward<T>(newName); }  // universal reference

  …
};
```

In short, rvalue references should be *unconditionally cast* to rvalues (via `std::move`) when forwarding them to other functions, because they're *always* bound to rvalues, and universal references should be *conditionally cast* to rvalues (via `std::forward`) when forwarding them, because they're only *sometimes* bound to rvalues.

Item 23 explains that using `std::forward` on rvalue references can be made to exhibit the proper behavior, but the source code is wordy, error-prone, and unidiomatic, so you should avoid using `std::forward` with rvalue references. Even worse is the idea of using `std::move` with universal references, because that can have the effect of unexpectedly modifying lvalues (e.g., local variables):

```
class Widget {
public:
  template<typename T>
  void setName(T&& newName)         // universal reference
  { name = std::move(newName); }    // compiles, but is
  …                                 // bad, bad, bad!

private:
  std::string name;
  std::shared_ptr<SomeDataStructure> p;
};

std::string getWidgetName();        // factory function

Widget w;

auto n = getWidgetName();           // n is local variable

w.setName(n);                       // moves n into w!

…                                   // n's value now unknown
```

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.

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:

```
class Widget {
public:
  void setName(const std::string& newName)      // set from
  { name = newName; }                           // const lvalue

  void setName(std::string&& newName)           // set from
  { name = std::move(newName); }                // rvalue

  …
};
```

That would certainly work in this case, but there are drawbacks. First, it's more source code to write and maintain (two functions instead of a single template). Second, it can be less efficient. For example, consider this use of `setName`:

```
w.setName("Adela Novak");
```

With the version of `setName` taking a universal reference, the string literal `"Adela Novak"` would be passed to `setName`, where it would be conveyed to the assignment operator for the `std::string` inside `w`. `w`'s `name` data member would thus be assigned directly from the string literal; no temporary `std::string` objects would arise. With the overloaded versions of `setName`, however, a temporary `std::string` object would be created for `setName`'s parameter to bind to, and this temporary `std::string` would then be moved into `w`'s data member. A call to `setName` would thus entail execution of one `std::string` constructor (to create the temporary), one `std::string` move assignment operator (to move `newName` into `w.name`), and one `std::string` destructor (to destroy the temporary). That's almost certainly a more expensive execution sequence than invoking only the `std::string` assignment operator taking a `const char*` pointer. The additional cost is likely to vary from implementation to implementation, and whether that cost is worth worrying about will vary from application to application and library to library, but the fact is that replacing a template taking a universal reference with a pair of functions overloaded on lvalue references and rvalue references is likely to incur a runtime cost in some cases. If we generalize the example such that `Widget`'s data member may be of an arbitrary type (rather than knowing that it's `std::string`), the performance gap can widen considerably, because not all types are as cheap to move as `std::string` (see Item 29).

The most serious problem with overloading on lvalues and rvalues, however, isn't the volume or idiomaticity of the source code, nor is it the code's runtime performance. It's the poor scalability of the design. `Widget::setName` takes only one parameter, so only two overloads are necessary, but for functions taking more parameters, each of which could be an lvalue or an rvalue, the number of overloads grows geometrically: n parameters necessitates 2^n overloads. And that's not the worst of it. Some functions —function templates, actually—take an *unlimited* number of parameters, each of which could be an lvalue or rvalue. The poster children for such functions are `std::make_shared`, and, as of C++14, `std::make_unique` (see Item 21). Check out the declarations of their most commonly used overloads:

```
template<class T, class... Args>              // from C++11
shared_ptr<T> make_shared(Args&&... args);    // Standard

template<class T, class... Args>              // from C++14
unique_ptr<T> make_unique(Args&&... args);    // Standard
```

For functions like these, overloading on lvalues and rvalues is not an option: universal references are the only way to go. And inside such functions, I assure you, `std::forward` is applied to the universal reference parameters when they're passed to other functions. Which is exactly what you should do.

Well, usually. Eventually. But not necessarily initially. In some cases, you'll want to use the object bound to an rvalue reference or a universal reference more than once in a single function, and you'll want to make sure that it's not moved from until you're otherwise done with it. In that case, you'll want to apply `std::move` (for rvalue references) or `std::forward` (for universal references) to only the *final* use of the reference. For example:

```
template<typename T>                     // text is
void setSignText(T&& text)               // univ. reference
{
  sign.setText(text);                    // use text, but
                                         // don't modify it

  auto now =                             // get current time
    std::chrono::system_clock::now();

  signHistory.add(now,
                  std::forward<T>(text)); // conditionally cast
}                                         // text to rvalue
```

Here, we want to make sure that `text`'s value doesn't get changed by `sign.setText`, because we want to use that value when we call `signHistory.add`. Ergo the use of `std::forward` on only the final use of the universal reference.

For `std::move`, the same thinking applies (i.e., apply `std::move` to an rvalue reference the last time it's used), but it's important to note that in rare cases, you'll want to call `std::move_if_noexcept` instead of `std::move`. To learn when and why, consult Item 14.

If you're in a function that returns *by value*, and you're returning an object bound to an rvalue reference or a universal reference, you'll want to apply `std::move` or `std::forward` when you return the reference. To see why, consider an `operator+` function to add two matrices together, where the left-hand matrix is known to be an rvalue (and can hence have its storage reused to hold the sum of the matrices):

```
Matrix                                       // by-value return
operator+(Matrix&& lhs, const Matrix& rhs)
{
  lhs += rhs;
```

```
    return std::move(lhs);                       // move lhs into
  }                                              // return value
```

By casting lhs to an rvalue in the return statement (via std::move), lhs will be moved into the function's return value location. If the call to std::move were omitted,

```
  Matrix                                         // as above
  operator+(Matrix&& lhs, const Matrix& rhs)
  {
    lhs += rhs;
    return lhs;                                  // copy lhs into
  }                                              // return value
```

the fact that lhs is an lvalue would force compilers to instead *copy* it into the return value location. Assuming that the Matrix type supports move construction, which is more efficient than copy construction, using std::move in the return statement yields more efficient code.

If Matrix does not support moving, casting it to an rvalue won't hurt, because the rvalue will simply be copied by Matrix's copy constructor (see Item 23). If Matrix is later revised to support moving, operator+ will automatically benefit the next time it is compiled. That being the case, there's nothing to be lost (and possibly much to be gained) by applying std::move to rvalue references being returned from functions that return by value.

The situation is similar for universal references and std::forward. Consider a function template reduceAndCopy that takes a possibly unreduced Fraction object, reduces it, and then returns a copy of the reduced value. If the original object is an rvalue, its value should be moved into the return value (thus avoiding the expense of making a copy), but if the original is an lvalue, an actual copy must be created. Hence:

```
  template<typename T>
  Fraction                            // by-value return
  reduceAndCopy(T&& frac)             // universal reference param
  {
    frac.reduce();
    return std::forward<T>(frac);     // move rvalue into return
  }                                   // value, copy lvalue
```

If the call to std::forward were omitted, frac would be unconditionally copied into reduceAndCopy's return value.

Some programmers take the information above and try to extend it to situations where it doesn't apply. "If using std::move on an rvalue reference parameter being copied into a return value turns a copy construction into a move construction," they reason, "I can perform the same optimization on local variables that I'm returning." In other words, they figure that given a function returning a local variable by value, such as this,

```
Widget makeWidget()          // "Copying" version of makeWidget
{
  Widget w;                  // local variable

  …                          // configure w

  return w;                  // "copy" w into return value
}
```

they can "optimize" it by turning the "copy" into a move:

```
Widget makeWidget()          // Moving version of makeWidget
{
  Widget w;
  …
  return std::move(w);       // move w into return value
}                            // (don't do this!)
```

My liberal use of quotation marks should tip you off that this line of reasoning is flawed. But why is it flawed?

It's flawed, because the Standardization Committee is way ahead of such programmers when it comes to this kind of optimization. It was recognized long ago that the "copying" version of `makeWidget` can avoid the need to copy the local variable `w` by constructing it in the memory alloted for the function's return value. This is known as the *return value optimization* (RVO), and it's been expressly blessed by the C++ Standard for as long as there's been one.

Wording such a blessing is finicky business, because you want to permit such *copy elision* only in places where it won't affect the observable behavior of the software. Paraphrasing the legalistic (arguably toxic) prose of the Standard, this particular blessing says that compilers may elide the copying (or moving) of a local object[2] in a function that returns by value if (1) the type of the local object is the same as that returned by the function and (2) the local object is what's being returned. With that in mind, look again at the "copying" version of `makeWidget`:

2 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).

```
Widget makeWidget()          // "Copying" version of makeWidget
{
  Widget w;
  …
  return w;                  // "copy" w into return value
}
```

Both conditions are fulfilled here, and you can trust me when I tell you that for this code, every decent C++ compiler will employ the RVO to avoid copying `w`. That means that the "copying" version of `makeWidget` doesn't, in fact, copy anything.

The moving version of `makeWidget` does just what its name says it does (assuming `Widget` offers a move constructor): it moves the contents of `w` into `makeWidget`'s return value location. But why don't compilers use the RVO to eliminate the move, again constructing `w` in the memory alloted for the function's return value? The answer is simple: they can't. Condition (2) stipulates that the RVO may be performed only if what's being returned is a local object, but that's not what the moving version of `makeWidget` is doing. Look again at its `return` statement:

```
return std::move(w);
```

What's being returned here isn't the local object `w`, it's *a reference to* `w`—the result of `std::move(w)`. Returning a reference to a local object doesn't satisfy the conditions required for the RVO, so compilers must move `w` into the function's return value location. Developers trying to help their compilers optimize by applying `std::move` to a local variable that's being returned are actually limiting the optimization options available to their compilers!

But the RVO is an optimization. Compilers aren't *required* to elide copy and move operations, even when they're permitted to. Maybe you're paranoid, and you worry that your compilers will punish you with copy operations, just because they can. Or perhaps you're insightful enough to recognize that there are cases where the RVO is difficult for compilers to implement, e.g., when different control paths in a function return different local variables. (Compilers would have to generate code to construct the appropriate local variable in the memory allotted for the function's return value, but how could compilers determine which local variable would be appropriate?) If so, you might be willing to pay the price of a move as insurance against the cost of a copy. That is, you might still think it's reasonable to apply `std::move` to a local object you're returning, simply because you'd rest easy knowing you'd never pay for a copy.

In that case, applying `std::move` to a local object would *still* be a bad idea. The part of the Standard blessing the RVO goes on to say that if the conditions for the RVO are met, but compilers choose not to perform copy elision, the object being returned *must be treated as an rvalue*. In effect, the Standard requires that when the RVO is permitted, either copy elision takes place or `std::move` is implicitly applied to local objects being returned. So in the "copying" version of `makeWidget`,

```
Widget makeWidget()          // as before
{
  Widget w;
  …
  return w;
}
```

compilers must either elide the copying of `w` or they must treat the function as if it were written like this:

```
Widget makeWidget()
{
  Widget w;
  …
  return std::move(w);       // treat w as rvalue, because
}                            // no copy elision was performed
```

The situation is similar for by-value function parameters. They're not eligible for copy elision with respect to their function's return value, but compilers must treat them as rvalues if they're returned. As a result, if your source code looks like this,

```
Widget makeWidget(Widget w)        // by-value parameter of same
{                                  // type as function's return
  …
  return w;
}
```

compilers must treat it as if it had been written this way:

```
Widget makeWidget(Widget w)
{
  …
  return std::move(w);             // treat w as rvalue
}
```

This means that if you use `std::move` on a local object being returned from a function that's returning by value, you can't help your compilers (they have to treat the local object as an rvalue if they don't perform copy elision), but you can certainly hinder them (by precluding the RVO). There are situations where applying `std::move` to a local variable can be a reasonable thing to do (i.e., when you're passing it to a function and you know you won't be using the variable any longer), but as part of a `return` statement that would otherwise qualify for the RVO or that returns a by-value parameter isn't among them.

Things to Remember

- Apply `std::move` to rvalue references and `std::forward` to universal references the last time each is used.
- Do the same thing for rvalue references and universal references being returned from functions that return by value.
- Never apply `std::move` or `std::forward` to local objects if they would otherwise be eligible for the return value optimization.

Item 26: Avoid overloading on universal references.

Suppose you need to write a function that takes a name as a parameter, logs the current date and time, then adds the name to a global data structure. You might come up with a function that looks something like this:

```
std::multiset<std::string> names;     // global data structure

void logAndAdd(const std::string& name)
{
  auto now =                          // get current time
    std::chrono::system_clock::now();

  log(now, "logAndAdd");              // make log entry

  names.emplace(name);                // add name to global data
}                                     // structure; see Item 42
                                      // for info on emplace
```

This isn't unreasonable code, but it's not as efficient as it could be. Consider three potential calls:

```
std::string petName("Darla");

logAndAdd(petName);                   // pass lvalue std::string

logAndAdd(std::string("Persephone")); // pass rvalue std::string

logAndAdd("Patty Dog");               // pass string literal
```

In the first call, `logAndAdd`'s parameter `name` is bound to the variable `petName`. Within `logAndAdd`, `name` is ultimately passed to `names.emplace`. Because `name` is an lvalue, it is copied into `names`. There's no way to avoid that copy, because an lvalue (`petName`) was passed into `logAndAdd`.

In the second call, the parameter name is bound to an rvalue (the temporary std::string explicitly created from "Persephone"). name itself is an lvalue, so it's copied into names, but we recognize that, in principle, its value could be moved into names. In this call, we pay for a copy, but we should be able to get by with only a move.

In the third call, the parameter name is again bound to an rvalue, but this time it's to a temporary std::string that's implicitly created from "Patty Dog". As in the second call, name is copied into names, but in this case, the argument originally passed to logAndAdd was a string literal. Had that string literal been passed directly to emplace, there would have been no need to create a temporary std::string at all. Instead, emplace would have used the string literal to create the std::string object directly inside the std::multiset. In this third call, then, we're paying to copy a std::string, yet there's really no reason to pay even for a move, much less a copy.

We can eliminate the inefficiencies in the second and third calls by rewriting logAndAdd to take a universal reference (see Item 24) and, in accord with Item 25, std::forwarding this reference to emplace. The results speak for themselves:

```
template<typename T>
void logAndAdd(T&& name)
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}

std::string petName("Darla");          // as before

logAndAdd(petName);                    // as before, copy
                                       // lvalue into multiset

logAndAdd(std::string("Persephone"));  // move rvalue instead
                                       // of copying it

logAndAdd("Patty Dog");                // create std::string
                                       // in multiset instead
                                       // of copying a temporary
                                       // std::string
```

Hurray, optimal efficiency!

Were this the end of the story, we could stop here and proudly retire, but I haven't told you that clients don't always have direct access to the names that logAndAdd requires. Some clients have only an index that `logAndAdd` uses to look up the corresponding name in a table. To support such clients, `logAndAdd` is overloaded:

```
std::string nameFromIdx(int idx);      // return name
                                       // corresponding to idx

void logAndAdd(int idx)                // new overload
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(nameFromIdx(idx));
}
```

Resolution of calls to the two overloads works as expected:

```
std::string petName("Darla");          // as before

logAndAdd(petName);                    // as before, these
logAndAdd(std::string("Persephone"));  // calls all invoke
logAndAdd("Patty Dog");                // the T&& overload

logAndAdd(22);                         // calls int overload
```

Actually, resolution works as expected only if you don't expect too much. Suppose a client has a `short` holding an index and passes that to `logAndAdd`:

```
short nameIdx;
…                                      // give nameIdx a value

logAndAdd(nameIdx);                    // error!
```

The comment on the last line isn't terribly illuminating, so let me explain what happens here.

There are two `logAndAdd` overloads. The one taking a universal reference can deduce `T` to be `short`, thus yielding an exact match. The overload with an `int` parameter can match the `short` argument only with a promotion. Per the normal overload resolution rules, an exact match beats a match with a promotion, so the universal reference overload is invoked.

Within that overload, the parameter `name` is bound to the `short` that's passed in. `name` is then `std::forward`ed to the `emplace` member function on `names` (a `std::multiset<std::string>`), which, in turn, dutifully forwards it to the `std::string` constructor. There is no constructor for `std::string` that takes a `short`, so the `std::string` constructor call inside the call to `multiset::emplace` inside the call to logAndAdd fails. All because the universal reference overload was a better match for a short argument than an int.

Functions taking universal references are the greediest functions in C++. They instantiate to create exact matches for almost any type of argument. (The few kinds of arguments where this isn't the case are described in Item 30.) This is why combining overloading and universal references is almost always a bad idea: the universal reference overload vacuums up far more argument types than the developer doing the overloading generally expects.

An easy way to topple into this pit is to write a perfect forwarding constructor. A small modification to the logAndAdd example demonstrates the problem. Instead of writing a free function that can take either a std::string or an index that can be used to look up a std::string, imagine a class Person with constructors that do the same thing:

```
class Person {
public:
  template<typename T>
  explicit Person(T&& n)            // perfect forwarding ctor;
  : name(std::forward<T>(n)) {}     // initializes data member

  explicit Person(int idx)          // int ctor
  : name(nameFromIdx(idx)) {}
  …

private:
  std::string name;
};
```

As was the case with logAndAdd, passing an integral type other than int (e.g., std::size_t, short, long, etc.) will call the universal reference constructor overload instead of the int overload, and that will lead to compilation failures. The problem here is much worse, however, because there's more overloading present in Person than meets the eye. Item 17 explains that under the appropriate conditions, C++ will generate both copy and move constructors, and this is true even if the class contains a templatized constructor that could be instantiated to produce the signature of the copy or move constructor. If the copy and move constructors for Person are thus generated, Person will effectively look like this:

```
class Person {
public:
  template<typename T>               // perfect forwarding ctor
  explicit Person(T&& n)
  : name(std::forward<T>(n)) {}
```

```
  explicit Person(int idx);           // int ctor

  Person(const Person& rhs);          // copy ctor
                                      // (compiler-generated)

  Person(Person&& rhs);               // move ctor
  …                                   // (compiler-generated)

};
```

This leads to behavior that's intuitive only if you've spent so much time around compilers and compiler-writers, you've forgotten what it's like to be human:

```
Person p("Nancy");

auto cloneOfP(p);                     // create new Person from p;
                                      // this won't compile!
```

Here we're trying to create a `Person` from another `Person`, which seems like about as obvious a case for copy construction as one can get. (`p`'s an lvalue, so we can banish any thoughts we might have about the "copying" being accomplished through a move operation.) But this code won't call the copy constructor. It will call the perfect-forwarding constructor. That function will then try to initialize `Person`'s `std::string` data member with a `Person` object (`p`). `std::string` having no constructor taking a `Person`, your compilers will throw up their hands in exasperation, possibly punishing you with long and incomprehensible error messages as an expression of their displeasure.

"Why," you might wonder, "does the perfect-forwarding constructor get called instead of the copy constructor? We're initializing a `Person` with another `Person`!" Indeed we are, but compilers are sworn to uphold the rules of C++, and the rules of relevance here are the ones governing the resolution of calls to overloaded functions.

Compilers reason as follows. `cloneOfP` is being initialized with a non-`const` lvalue (`p`), and that means that the templatized constructor can be instantiated to take a non-`const` lvalue of type `Person`. After such instantiation, the `Person` class looks like this:

```
class Person {
public:
  explicit Person(Person& n)              // instantiated from
  : name(std::forward<Person&>(n)) {}     // perfect-forwarding
                                          // template

  explicit Person(int idx);               // as before
```

```
  Person(const Person& rhs);              // copy ctor
  …                                       // (compiler-generated)

};
```

In the statement,

```
auto cloneOfP(p);
```

p could be passed to either the copy constructor or the instantiated template. Calling the copy constructor would require adding const to p to match the copy constructor's parameter's type, but calling the instantiated template requires no such addition. The overload generated from the template is thus a better match, so compilers do what they're designed to do: generate a call to the better-matching function. "Copying" non-const lvalues of type Person is thus handled by the perfect-forwarding constructor, not the copy constructor.

If we change the example slightly so that the object to be copied is const, we hear an entirely different tune:

```
const Person cp("Nancy");     // object is now const

auto cloneOfP(cp);            // calls copy constructor!
```

Because the object to be copied is now const, it's an exact match for the parameter taken by the copy constructor. The templatized constructor can be instantiated to have the same signature,

```
class Person {
public:
  explicit Person(const Person& n);       // instantiated from
                                          // template

  Person(const Person& rhs);              // copy ctor
                                          // (compiler-generated)

  …
};
```

but this doesn't matter, because one of the overload-resolution rules in C++ is that in situations where a template instantiation and a non-template function (i.e., a "normal" function) are equally good matches for a function call, the normal function is preferred. The copy constructor (a normal function) thereby trumps an instantiated template with the same signature.

(If you're wondering why compilers generate a copy constructor when they could instantiate a templatized constructor to get the signature that the copy constructor would have, review Item 17.)

The interaction among perfect-forwarding constructors and compiler-generated copy and move operations develops even more wrinkles when inheritance enters the picture. In particular, the conventional implementations of derived class copy and move operations behave quite surprisingly. Here, take a look:

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs)  // copy ctor; calls
  : Person(rhs)                            // base class
  { … }                                    // forwarding ctor!

  SpecialPerson(SpecialPerson&& rhs)       // move ctor; calls
  : Person(std::move(rhs))                 // base class
  { … }                                    // forwarding ctor!
};
```

As the comments indicate, the derived class copy and move constructors don't call their base class's copy and move constructors, they call the base class's perfect-forwarding constructor! To understand why, note that the derived class functions are using arguments of type `SpecialPerson` to pass to their base class, then work through the template instantiation and overload-resolution consequences for the constructors in class `Person`. Ultimately, the code won't compile, because there's no `std::string` constructor taking a `SpecialPerson`.

I hope that by now I've convinced you that overloading on universal reference parameters is something you should avoid if at all possible. But if overloading on universal references is a bad idea, what do you do if you need a function that forwards most argument types, yet needs to treat some argument types in a special fashion? That egg can be unscrambled in a number of ways. So many, in fact, that I've devoted an entire Item to them. It's Item 27. The next Item. Keep reading, you'll bump right into it.

Things to Remember

- Overloading on universal references almost always leads to the universal reference overload being called more frequently than expected.
- Perfect-forwarding constructors are especially problematic, because they're typically better matches than copy constructors for non-`const` lvalues, and they can hijack derived class calls to base class copy and move constructors.

Item 27: Familiarize yourself with alternatives to overloading on universal references.

Item 26 explains that overloading on universal references can lead to a variety of problems, both for freestanding and for member functions (especially constructors). Yet it also gives examples where such overloading could be useful. If only it would behave the way we'd like! This Item explores ways to achieve the desired behavior, either through designs that avoid overloading on universal references or by employing them in ways that constrain the types of arguments they can match.

The discussion that follows builds on the examples introduced in Item 26. If you haven't read that Item recently, you'll want to review it before continuing.

Abandon overloading

The first example in Item 26, `logAndAdd`, is representative of the many functions that can avoid the drawbacks of overloading on universal references by simply using different names for the would-be overloads. The two `logAndAdd` overloads, for example, could be broken into `logAndAddName` and `logAndAddNameIdx`. Alas, this approach won't work for the second example we considered, the `Person` constructor, because constructor names are fixed by the language. Besides, who wants to give up overloading?

Pass by const T&

An alternative is to revert to C++98 and replace pass-by-universal-reference with pass-by-lvalue-reference-to-`const`. In fact, that's the first approach Item 26 considers (shown on page 175). The drawback is that the design isn't as efficient as we'd prefer. Knowing what we now know about the interaction of universal references and overloading, giving up some efficiency to keep things simple might be a more attractive trade-off than it initially appeared.

Pass by value

An approach that often allows you to dial up performance without any increase in complexity is to replace pass-by-reference parameters with, counterintuitively, pass by value. The design adheres to the advice in Item 41 to consider passing objects by value when you know you'll copy them, so I'll defer to that Item for a detailed discussion of how things work and how efficient they are. Here, I'll just show how the technique could be used in the `Person` example:

```
class Person {
public:
  explicit Person(std::string n) // replaces T&& ctor; see
```

```
  : name(std::move(n)) {}        // Item 41 for use of std::move

  explicit Person(int idx)       // as before
  : name(nameFromIdx(idx)) {}
  …

private:
  std::string name;
};
```

Because there's no `std::string` constructor taking only an integer, all `int` and `int`-like arguments to a `Person` constructor (e.g., `std::size_t`, `short`, `long`) get funneled to the `int` overload. Similarly, all arguments of type `std::string` (and things from which `std::string`s can be created, e.g., literals such as `"Ruth"`) get passed to the constructor taking a `std::string`. There are thus no surprises for callers. You could argue, I suppose, that some people might be surprised that using `0` or `NULL` to indicate a null pointer would invoke the `int` overload, but such people should be referred to Item 8 and required to read it repeatedly until the thought of using `0` or `NULL` as a null pointer makes them recoil.

Use Tag dispatch

Neither pass by lvalue-reference-to-`const` nor pass by value offers support for perfect forwarding. If the motivation for the use of a universal reference is perfect forwarding, we have to use a universal reference; there's no other choice. Yet we don't want to abandon overloading. So if we don't give up overloading and we don't give up universal references, how can we avoid overloading on universal references?

It's actually not that hard. Calls to overloaded functions are resolved by looking at all the parameters of all the overloads as well as all the arguments at the call site, then choosing the function with the best overall match—taking into account all parameter/argument combinations. A universal reference parameter generally provides an exact match for whatever's passed in, but if the universal reference is part of a parameter list containing other parameters that are *not* universal references, sufficiently poor matches on the non-universal reference parameters can knock an overload with a universal reference out of the running. That's the basis behind the *tag dispatch* approach, and an example will make the foregoing description easier to understand.

We'll apply tag dispatch to the `logAndAdd` example on page 177. Here's the code for that example, lest you get sidetracked looking it up:

```
std::multiset<std::string> names;     // global data structure

template<typename T>                  // make log entry and add
```

```
void logAndAdd(T&& name)              // name to data structure
{
  auto now = std::chrono::system_clock::now();
  log(now, "logAndAdd");
  names.emplace(std::forward<T>(name));
}
```

By itself, this function works fine, but were we to introduce the overload taking an `int` that's used to look up objects by index, we'd be back in the troubled land of Item 26. The goal of this Item is to avoid that. Rather than adding the overload, we'll reimplement `logAndAdd` to delegate to two other functions, one for integral values and one for everything else. `logAndAdd` itself will accept all argument types, both integral and non-integral.

The two functions doing the real work will be named `logAndAddImpl`, i.e., we'll use overloading. One of the functions will take a universal reference. So we'll have both overloading and universal references. But each function will also take a second parameter, one that indicates whether the argument being passed is integral. This second parameter is what will prevent us from tumbling into the morass described in Item 26, because we'll arrange it so that the second parameter will be the factor that determines which overload is selected.

Yes, I know, "Blah, blah, blah. Stop talking and show me the code!" No problem. Here's an almost-correct version of the updated `logAndAdd`:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(std::forward<T>(name),
                std::is_integral<T>());     // not quite correct
}
```

This function forwards its parameter to `logAndAddImpl`, but it also passes an argument indicating whether that parameter's type (`T`) is integral. At least, that's what it's supposed to do. For integral arguments that are rvalues, it's also what it does. But, as Item 28 explains, if an lvalue argument is passed to the universal reference `name`, the type deduced for `T` will be an lvalue reference. So if an lvalue of type `int` is passed to `logAndAdd`, `T` will be deduced to be `int&`. That's not an integral type, because references aren't integral types. That means that `std::is_integral<T>` will be false for any lvalue argument, even if the argument really does represent an integral value.

Recognizing the problem is tantamount to solving it, because the ever-handy Standard C++ Library has a type trait (see Item 9), `std::remove_reference`, that does both what its name suggests and what we need: remove any reference qualifiers from a type. The proper way to write `logAndAdd` is therefore:

```
template<typename T>
void logAndAdd(T&& name)
{
  logAndAddImpl(
    std::forward<T>(name),
    std::is_integral<typename std::remove_reference<T>::type>()
  );
}
```

This does the trick. (In C++14, you can save a few keystrokes by using `std::remove_reference_t<T>` in place of the highlighted text. For details, see Item 9.)

With that taken care of, we can shift our attention to the function being called, `logAndAddImpl`. There are two overloads, and the first is applicable only to non-integral types (i.e., to types where `std::is_integral<typename std::remove_reference<T>::type>` is false):

```
template<typename T>                              // non-integral
void logAndAddImpl(T&& name, std::false_type)     // argument:
{                                                 // add it to
  auto now = std::chrono::system_clock::now();    // global data
  log(now, "logAndAdd");                          // structure
  names.emplace(std::forward<T>(name));
}
```

This is straightforward code, once you understand the mechanics behind the highlighted parameter. Conceptually, `logAndAdd` passes a boolean to `logAndAddImpl` indicating whether an integral type was passed to `logAndAdd`, but `true` and `false` are *runtime* values, and we need to use overload resolution—a *compile-time* phenomenon—to choose the correct `logAndAddImpl` overload. That means we need a *type* that corresponds to `true` and a different type that corresponds to `false`. This need is common enough that the Standard Library provides what is required under the names `std::true_type` and `std::false_type`. The argument passed to `logAndAddImpl` by `logAndAdd` is an object of a type that inherits from `std::true_type` if `T` is integral and from `std::false_type` if `T` is not integral. The net result is that this `logAndAddImpl` overload is a viable candidate for the call in `logAndAdd` only if `T` is not an integral type.

The second overload covers the opposite case: when `T` is an integral type. In that event, `logAndAddImpl` simply finds the name corresponding to the passed-in index and passes that name back to `logAndAdd`:

```
std::string nameFromIdx(int idx);              // as in Item 26
```

```
void logAndAddImpl(int idx, std::true_type)   // integral
{                                             // argument: look
  logAndAdd(nameFromIdx(idx));                // up name and
}                                             // call logAndAdd
                                              // with it
```

By having `logAndAddImpl` for an index look up the corresponding name and pass it to `logAndAdd` (from where it will be `std::forward`ed to the other `logAndAddImpl` overload), we avoid the need to put the logging code in both `logAndAddImpl` overloads.

In this design, the types `std::true_type` and `std::false_type` are "tags" whose only purpose is to force overload resolution to go the way we want. Notice that we don't even name those parameters. They serve no purpose at runtime, and in fact we hope that compilers will recognize that the tag parameters are unused and will optimize them out of the program's execution image. (Some compilers do, at least some of the time.) The call to the overloaded implementation functions inside `logAndAdd` "dispatches" the work to the correct overload by causing the proper tag object to be created. Hence the name for this design: *tag dispatch*. It's a standard building block of template metaprogramming, and the more you look at code inside contemporary C++ libraries, the more often you'll encounter it.

For our purposes, what's important about tag dispatch is less how it works and more how it permits us to combine universal references and overloading without the problems described in Item 26. The dispatching function—`logAndAdd`—takes an unconstrained universal reference parameter, but this function is not overloaded. The implementation functions—`logAndAddImpl`—are overloaded, and one takes a universal reference parameter, but resolution of calls to these functions depends not just on the universal reference parameter, but also on the tag parameter, and the tag values are designed so that no more than one overload will be a viable match. As a result, it's the tag that determines which overload gets called. The fact that the universal reference parameter will always generate an exact match for its argument is immaterial.

Constraining templates that take universal references

A keystone of tag dispatch is the existence of a single (unoverloaded) function as the client API. This single function dispatches the work to be done to the implementation functions. Creating an unoverloaded dispatch function is usually easy, but the second problem case Item 26 considers, that of a perfect-forwarding constructor for the `Person` class (shown on page 178), is an exception. Compilers may generate copy and move constructors themselves, so even if you write only one constructor and use tag dispatch within it, some constructor calls may be handled by compiler-generated functions that bypass the tag dispatch system.

In truth, the real problem is not that the compiler-generated functions sometimes bypass the tag dispatch design, it's that they don't *always* pass it by. You virtually always want the copy constructor for a class to handle requests to copy lvalues of that type, but, as Item 26 demonstrates, providing a constructor taking a universal reference causes the universal reference constructor (rather than the copy constructor) to be called when copying non-`const` lvalues. That Item also explains that when a base class declares a perfect-forwarding constructor, that constructor will typically be called when derived classes implement their copy and move constructors in the conventional fashion, even though the correct behavior is for the base class's copy and move constructors to be invoked.

For situations like these, where an overloaded function taking a universal reference is greedier than you want, yet not greedy enough to act as a single dispatch function, tag dispatch is not the droid you're looking for. You need a different technique, one that lets you rachet down the conditions under which the function template that the universal reference is part of is permitted to be employed. What you need, my friend, is `std::enable_if`.

`std::enable_if` gives you a way to force compilers to behave as if a particular template didn't exist. Such templates are said to be *disabled*. By default, all templates are *enabled*, but a template using `std::enable_if` is enabled only if the condition specified by `std::enable_if` is satisfied. In our case, we'd like to enable the `Person` perfect-forwarding constructor only if the type being passed isn't `Person`. If the type being passed is `Person`, we want to disable the perfect-forwarding constructor (i.e., cause compilers to ignore it), because that will cause the class's copy or move constructor to handle the call, which is what we want when a `Person` object is initialized with another `Person`.

The way to express that idea isn't particularly difficult, but the syntax is off-putting, especially if you've never seen it before, so I'll ease you into it. There's some boilerplate that goes around the condition part of `std::enable_if`, so we'll start with that. Here's the declaration for the perfect-forwarding constructor in `Person`, showing only as much of the `std::enable_if` as is required simply to use it. I'm showing only the declaration for this constructor, because the use of `std::enable_if` has no effect on the function's implementation. The implementation remains the same as in Item 26.

```
class Person {
public:
  template<typename T,
           typename = typename std::enable_if<condition>::type>
  explicit Person(T&& n);

  …
```

```
};
```

To understand exactly what's going on in the highlighted text, I must regretfully suggest that you consult other sources, because the details take a while to explain, and there's just not enough space for it in this book. (During your research, look into "SFINAE" as well as `std::enable_if`, because SFINAE is the technology that makes `std::enable_if` work.) Here, I want to focus on expression of the condition that will control whether this constructor is enabled.

The condition we want to specify is that `T` isn't `Person`, i.e., that the templatized constructor should be enabled only if `T` is a type other than `Person`. Thanks to a type trait that determines whether two types are the same (`std::is_same`), it would seem that the condition we want is `!std::is_same<Person, T>::value`. (Notice the "`!`" at the beginning of the expression. We want for `Person` and `T` to *not* be the same.) This is close to what we need, but it's not quite correct, because, as Item 28 explains, the type deduced for a universal reference initialized with an lvalue is always an lvalue reference. That means that for code like this,

```
Person p("Nancy");

auto cloneOfP(p);          // initialize from lvalue
```

the type `T` in the universal constructor will be deduced to be `Person&`. The types `Person` and `Person&` are not the same, and the result of `std::is_same` will reflect that: `std::is_same<Person, Person&>::value` is false.

If we think more precisely about what we mean when we say that the templatized constructor in `Person` should be enabled only if `T` isn't `Person`, we'll realize that when we're looking at `T`, we want to ignore

- **Whether it's a reference.** For the purpose of determining whether the universal reference constructor should be enabled, the types `Person`, `Person&`, and `Person&&` are all the same as `Person`.
- **Whether it's `const` or `volatile`.** As far as we're concerned, a `const Person` and a `volatile Person` and a `const volatile Person` are all the same as a `Person`.

This means we need a way to strip any references, `const`s, and `volatile`s from `T` before checking to see if that type is the same as `Person`. Once again, the Standard Library gives us what we need in the form of a type trait. That trait is `std::decay`. `std::decay<T>::type` is the same as `T`, except that references and *cv-qualifiers* (i.e., `const` or `volatile` qualifiers) are removed. (I'm fudging the truth here, because `std::decay`, as its name suggests, also turns array and function types into pointers (see Item 1), but for purposes of this discussion, std::decay behaves as I've described.) The condition we want to control whether our constructor is enabled, then, is

```
!std::is_same<Person, typename std::decay<T>::type>::value
```

i.e., Person is not the same type as T, ignoring any references or cv-qualifiers. (As Item 9 explains, the "typename" in front of std::decay is required, because the type std::decay<T>::type depends on the template parameter T.)

Inserting this condition into the std::enable_if boilerplate above, plus formatting the result to make it easier to see how the pieces fit together, yields this declaration for Person's perfect-forwarding constructor:

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_same<Person,
                               typename std::decay<T>::type
                              >::value
               >::type
  >
  explicit Person(T&& n);

  …

};
```

If you've never seen anything like this before, count your blessings. There's a reason I saved this design for last. When you can use one of the other mechanisms to avoid mixing universal references and overloading (and you almost always can), you should. Still, once you get used to the functional syntax and the proliferation of angle brackets, it's not that bad. Furthermore, this gives you the behavior you've been striving for. Given the declaration above, constructing a Person from another Person—lvalue or rvalue, const or non-const, volatile or non-volatile—will never invoke the constructor taking a universal reference.

Success, right? We're done!

Um, no. Belay that celebration. There's still one loose end from Item 26 that continues to flap about. We need to tie it down.

Suppose a class derived from Person implements the copy and move operations in the conventional manner:

```
class SpecialPerson: public Person {
public:
  SpecialPerson(const SpecialPerson& rhs)  // copy ctor; calls
  : Person(rhs)                            // base class
  { … }                                    // forwarding ctor!

  SpecialPerson(SpecialPerson&& rhs)       // move ctor; calls
  : Person(std::move(rhs))                 // base class
  { … }                                    // forwarding ctor!

  …
};
```

This is the same code I showed in Item 26 (on page 206), including the comments, which, alas, remain accurate. When we copy or move a `SpecialPerson` object, we expect to copy or move its base class parts using the base class's copy and move constructors, but in these functions, we're passing `SpecialPerson` objects to the base class's constructors, and because `SpecialPerson` isn't the same as `Person` (not even after application of `std::decay`), the universal reference constructor in the base class is enabled, and it happily instantiates to perform an exact match for a `SpecialPerson` argument. This exact match is better than the derived-to-base conversions that would be necessary to bind the `SpecialPerson` objects to the `Person` parameters in `Person`'s copy and move constructors, so with the code we have now, copying and moving `SpecialPerson` objects would use the `Person` perfect-forwarding constructor to copy or move their base class parts! It's déjà Item 26 all over again.

The derived class is just following the normal rules for implementing derived class copy and move constructors, so the fix for this problem is in the base class and, in particular, in the condition that controls whether `Person`'s universal reference constructor is enabled. We now realize that we don't want to enable the templatized constructor for any argument type other than `Person`, we want to enable it for any argument type other than `Person` *or a type derived from `Person`*. Pesky inheritance!

You should not be surprised to hear that among the standard type traits is one that determines whether one type is derived from another. It's called `std::is_base_of`. `std::is_base_of<T1, T2>::value` is true if `T2` is derived from `T1`. Types are considered to be derived from themselves, so `std::is_base_of<T, T>::value` is true. This is handy, because we want to revise our condition controlling `Person`'s perfect-forwarding constructor such that the constructor is enabled only if the type `T`, after stripping it of references and cv-qualifiers, is neither `Person` nor a class derived from `Person`. Using `std::is_base_of` instead of `std::is_same` gives us what we need:

```
class Person {
public:
  template<
    typename T,
    typename = typename std::enable_if<
                 !std::is_base_of<Person,
                                  typename std::decay<T>::type
                                 >::value
               >::type
  >
  explicit Person(T&& n);

  …

};
```

Now we're finally done. Provided we're writing the code in C++11, that is. If we're using C++14, this code will still work, but we can employ alias templates for `std::enable_if` and `std::decay` to get rid of the "`typename`" and "`::type`" cruft, thus yielding this somewhat more palatable code:

```
class Person {                                    // C++14
public:
  template<
    typename T,
    typename = std::enable_if_t<               // less code here
                 !std::is_base_of<Person,
                                  std::decay_t<T>   // and here
                                 >::value
               >                                  // and here
  >
  explicit Person(T&& n);

  …

};
```

Okay, I admit it: I lied. We're still not done. But we're close. Tantalizingly close. Honest.

We've seen how to use `std::enable_if` to selectively disable `Person`'s universal reference constructor for argument types we want to have handled by the class's copy and move constructors, but we haven't yet seen how to apply it to distinguish integral and non-integral arguments. That was, after all, our original goal; the constructor ambiguity problem was just something we got dragged into along the way.

All we need to do—and I really do mean that this is everything—is (1) add a `Person` constructor overload to handle integral arguments and (2) further constrain the templatized constructor so that it's disabled for such arguments. Pour these ingredients into the pot with everything else we've discussed, simmer over a low flame, and savor the aroma of success:

```
class Person {
public:
  template<
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)        // ctor for std::strings and
  : name(std::forward<T>(n))    // args convertible to
  { … }                         // std::strings

  explicit Person(int idx)      // ctor for integral args
  : name(nameFromIdx(idx))
  { … }

  …                             // copy and move ctors, etc.

private:
  std::string name;
};
```

Voilà! A thing of beauty! Well, okay, the beauty is perhaps most pronounced for those with something of a template metaprogramming fetish, but the fact remains that this approach not only gets the job done, it does it with unique aplomb. Because it uses perfect forwarding, it offers maximal efficiency, and because it controls the combination of universal references and overloading rather than forbidding it, this technique can be applied in circumstances (such as constructors) where overloading is unavoidable.

Trade-offs

The first three techniques considered in this Item—abandoning overloading, passing by `const T&`, and passing by value—specify a type for each parameter in the function(s) to be called. The last two techniques—tag dispatch and constraining template eligibility—use perfect forwarding, hence don't specify types for the parameters. This fundamental decision—to specify a type or not—has consequences.

As a rule, perfect forwarding is more efficient, because it avoids the creation of temporary objects solely for the purpose of conforming to the type of a parameter declaration. In the case of the `Person` constructor, perfect forwarding permits a string literal such as `"Nancy"` to be forwarded to the constructor for the `std::string` inside `Person`, whereas techniques not using perfect forwarding must create a temporary `std::string` object from the string literal to satisfy the parameter specification for the `Person` constructor.

But perfect forwarding has drawbacks. One is that some kinds of arguments can't be perfect-forwarded, even though they can be passed to functions taking specific types. Item 30 explores these perfect forwarding failure cases.

A second issue is the comprehensibility of error messages when clients pass invalid arguments. Suppose, for example, a client creating a `Person` object passes a string literal made up of `char16_t`s (a type introduced in C++11 to represent 16-bit characters) instead of `char`s (which is what a `std::string` consists of):

```
Person p(u"Konrad Zuse");   // "Konrad Zuse" consists of
                            // characters of type const char16_t
```

With the first three approaches examined in this Item, compilers will see that the available constructors take either `int` or `std::string`, and they'll produce a more or less straightforward error message explaining that there's no conversion from `const char16_t[12]` to `int` or `std::string`.

With an approach based on perfect forwarding, however, the array of `const char16_t`s gets bound to the constructor's parameter without complaint. From there it's forwarded to the constructor of `Person`'s `std::string` data member, and it's only at that point that the mismatch between what the caller passed in (a `const char16_t` array) and what's required (any type acceptable to the `std::string` constructor) is discovered. The resulting error message is likely to be, er, impressive. With one of the compilers I use, it's more than 160 lines long.

In this example, the universal reference is forwarded only once (from the `Person` constructor to the `std::string` constructor), but the more complex the system, the more likely that a universal reference is forwarded through several layers of function calls before finally arriving at a site that determines whether the argument type(s) are acceptable. The more times the universal reference is forwarded, the more baffling the error message may be when something goes wrong. Many developers find that this issue alone is grounds to reserve universal reference parameters for interfaces where performance is a foremost concern.

In the case of `Person`, we know that the forwarding function's universal reference parameter is supposed to be an initializer for a `std::string`, so we can use a static_assert to verify that it can play that role. The std::is_constructible type trait performs a compile-time test to determine whether an object of one type can be constructed from an object (or set of objects) of a different type (or set of types), so the assertion is easy to write:

```
class Person {
public:
  template<                                   // as before
    typename T,
    typename = std::enable_if_t<
      !std::is_base_of<Person, std::decay_t<T>>::value
      &&
      !std::is_integral<std::remove_reference_t<T>>::value
    >
  >
  explicit Person(T&& n)
  : name(std::forward<T>(n))
  {
    // assert that a std::string can be created from a T object
    static_assert(
      std::is_constructible<std::string, T>::value,
      "Parameter n can't be used to construct a std::string"
   );

   …                    // the usual ctor work goes here

  }

  …                     // remainder of Person class (as before)

};
```

This causes the specified error message to be produced if client code tries to create a Person from a type that can't be used to construct a std::string. Unfortunately, in this example the static_assert is in the body of the constructor, but the forwarding code, being part of the member initialization list, precedes it. With the compilers I use, the result is that the nice, readable message arising from the static_assert appears only *after* the usual error messages (up to 160-plus lines of them) have been emitted.

Things to Remember

- Alternatives to the combination of universal references and overloading include the use of distinct function names, passing parameters by lvalue-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.

Item 28: Understand reference collapsing.

Item 23 remarks that when an argument is passed to a template function, the type deduced for the template parameter encodes whether the argument is an lvalue or an rvalue. The Item fails to mention that this happens only when the argument is used to initialize a parameter that's a universal reference, but there's a good reason for the omission: universal references aren't introduced until Item 24. Together, these observations about universal references and lvalue/rvalue encoding mean that for this template,

```
template<typename T>
void func(T&& param);
```

the deduced template parameter `T` will encode whether the argument passed to `param` was an lvalue or an rvalue.

The encoding mechanism is simple. When an lvalue is passed as an argument, `T` is deduced to be an lvalue reference. When an rvalue is passed, `T` is deduced to be a non-reference. (Note the asymmetry: lvalues are encoded as lvalue references, but rvalues are encoded as *non-references*.) Hence:

```
Widget widgetFactory();     // function returning rvalue

Widget w;                   // a variable (an lvalue)

func(w);                    // call func with lvalue; T deduced
                            // to be Widget&

func(widgetFactory());      // call func with rvalue; T deduced
                            // to be Widget
```

In both calls to `func`, a `Widget` is passed, yet because one `Widget` is an lvalue and one is an rvalue, different types are deduced for the template parameter `T`. This, as we shall soon see, is what determines whether universal references become rvalue references or lvalue references, and it's also the underlying mechanism through which `std::forward` does its work.

Before we can look more closely at `std::forward` and universal references, we must note that references to references are illegal in C++. Should you try to declare one, your compilers will reprimand you:

```
int x;
…
auto& & rx = x;    // error! can't declare reference to reference
```

But consider what happens when an lvalue is passed to a function template taking a universal reference:

```
template<typename T>
void func(T&& param);      // as before

func(w);                   // invoke func with lvalue;
                           // T deduced as Widget&
```

If we take the type deduced for `T` (i.e., `Widget&`) and use it to instantiate the template, we get this:

```
void func(Widget& && param);
```

A reference to a reference! And yet compilers issue no protest. We know from Item 24 that because the universal reference `param` is being initialized with an lvalue, `param`'s type is supposed to be an lvalue reference, but how does the compiler get from the result of taking the deduced type for `T` and substituting it into the template to the following, which is the ultimate function signature?

```
void func(Widget& param);
```

The answer is *reference collapsing*. Yes, *you* are forbidden from declaring references to references, but *compilers* may produce them in particular contexts, template instantiation being among them. When compilers generate references to references, reference collapsing dictates what happens next.

There are two kinds of references (lvalue and rvalue), so there are four possible reference-reference combinations (lvalue to lvalue, lvalue to rvalue, rvalue to lvalue, and rvalue to rvalue). If a reference to a reference arises in a context where this is permitted (e.g., during template instantiation), the references *collapse* to a single reference according to this rule:

If either reference is an lvalue reference, the result is an lvalue reference. Otherwise (i.e., if both are rvalue references) the result is an rvalue reference.

In our example above, substitution of the deduced type `Widget&` into the template `func` yields an rvalue reference to an lvalue reference, and the reference-collapsing rule tells us that the result is an lvalue reference.

Reference collapsing is a key part of what makes `std::forward` work. As explained in Item 25, `std::forward` is applied to universal reference parameters, so a common use case looks like this:

```
template<typename T>
void f(T&& fParam)
{
  …                                  // do some work

  someFunc(std::forward<T>(fParam));  // forward fParam to
}                                     // someFunc
```

Because `fParam` is a universal reference, we know that the type parameter `T` will encode whether the argument passed to `f` (i.e., the expression used to initialize `fParam`) was an lvalue or an rvalue. `std::forward`'s job is to cast `fParam` (an lvalue) to an rvalue if and only if `T` encodes that the argument passed to `f` was an rvalue, i.e., if `T` is a non-reference type.

Here's how `std::forward` can be implemented to do that:

```
template<typename T>                                // in
T&& forward(typename                                // namespace
              remove_reference<T>::type& param)     // std
{
  return static_cast<T&&>(param);
}
```

This isn't quite Standards-conformant (I've omitted a few interface details), but the differences are irrelevant for the purpose of understanding how `std::forward` behaves.

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 `std::forward<Widget&>`. Plugging `Widget&` into the `std::forward` implementation yields this:

```
Widget& && forward(typename
                     remove_reference<Widget&>::type& param)
{ return static_cast<Widget& &&>(param); }
```

The type trait `std::remove_reference<Widget&>::type` yields `Widget` (see Item 9), so `std::forward` becomes:

```
Widget& && forward(Widget& param)
{ return static_cast<Widget& &&>(param); }
```

Reference collapsing is also applied to the return type and the cast, and the result is the final version of `std::forward` for the call:

```
Widget& forward(Widget& param)                  // still in
{ return static_cast<Widget&>(param);  }        // namespace std
```

As you can see, when an lvalue argument is passed to the function template `f`, `std::forward` is instantiated to take and return an lvalue reference. The cast inside `std::forward` does nothing, because `param`'s type is already `Widget&`, so casting it to `Widget&` has no effect. An lvalue argument passed to `std::forward` will thus return an lvalue reference. By definition, lvalue references are lvalues, so passing an lvalue to `std::forward` causes an lvalue to be returned, just like it's supposed to.

Now suppose that the argument passed to `f` is an rvalue of type `Widget`. In this case, the deduced type for `f`'s type parameter `T` will simply be `Widget`. The call inside `f` to `std::forward` will thus be to `std::forward<Widget>`. Substituting `Widget` for `T` in the `std::forward` implementation gives this:

```
Widget&& forward(typename
                   remove_reference<Widget>::type& param)
{ return static_cast<Widget&&>(param); }
```

Applying `std::remove_reference` to the non-reference type `Widget` yields the same type it started with (`Widget`), so `std::forward` becomes this:

```
Widget&& forward(Widget& param)
{ return static_cast<Widget&&>(param); }
```

There are no references to references here, so there's no reference collapsing, and this is the final instantiated version of `std::forward` for the call.

Rvalue references returned from functions are defined to be rvalues, so in this case, `std::forward` will turn `f`'s parameter `fParam` (an lvalue) into an rvalue. The end result is that an rvalue argument passed to `f` will be forwarded to `someFunc` as an rvalue, which is precisely what is supposed to happen.

In C++14, the existence of `std::remove_reference_t` makes it possible to implement `std::forward` a bit more concisely:

```
template<typename T>                          // C++14; still in
T&& forward(remove_reference_t<T>& param)     // namespace std
{
  return static_cast<T&&>(param);
}
```

Reference collapsing occurs in four contexts. The first and most common is template instantiation. The second is type generation for `auto` variables. The details are essentially the same as for templates, because type deduction for `auto` variables is essentially the same as type deduction for templates (see Item 2). Consider again this example from earlier in the Item:

```
template<typename T>
void func(T&& param);

Widget widgetFactory();     // function returning rvalue

Widget w;                   // a variable (an lvalue)

func(w);                    // call func with lvalue; T deduced
                            // to be Widget&

func(widgetFactory());      // call func with rvalue; T deduced
                            // to be Widget
```

This can be mimicked in `auto` form. The declaration

```
auto&& w1 = w;
```

initializes `w1` with an lvalue, thus deducing the type `Widget&` for `auto`. Plugging `Widget&` in for `auto` in the declaration for `w1` yields this reference-to-reference code,

```
Widget& && w1 = w;
```

which, after reference collapsing, becomes

```
Widget& w1 = w;
```

As a result, `w1` is an lvalue reference.

On the other hand, this declaration,

```
auto&& w2 = widgetFactory();
```

initializes `w2` with an rvalue, causing the non-reference type `Widget` to be deduced for `auto`. Substituting `Widget` for `auto` gives us this:

```
Widget&& w2 = widgetFactory();
```

There are no references to references here, so we're done; `w2` is an rvalue reference.

We're now in a position to truly understand the universal references introduced in Item 24. A universal reference isn't a new kind of reference, it's actually an rvalue reference in a context where two conditions are satisfied:

- **Type deduction distinguishes lvalues from rvalues**. Lvalues of type `T` are deduced to have type `T&`, while rvalues of type `T` yield `T` as their deduced type.
- **Reference collapsing occurs**.

The concept of universal references is useful, because it frees you from having to recognize the existence of reference collapsing contexts, to mentally deduce different types for lvalues and rvalues, and to apply the reference collapsing rule after mentally substituting the deduced types into the contexts in which they occur.

I said there were four such contexts, but we've discussed only two: template instantiation and `auto` type generation. The third is the generation and use of `typedef`s and alias declarations (see Item 9). If, during creation or evaluation of a `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,

```
template<typename T>
class Widget {
public:
  typedef T&& RvalueRefToT;
  …
};
```

and suppose we instantiate `Widget` with an lvalue reference type:

```
Widget<int&> w;
```

Substituting `int&` for `T` in the `Widget` template gives us the following `typedef`:

```
typedef int& && RvalueRefToT;
```

Reference collapsing reduces it to this,

```
typedef int& RvalueRefToT;
```

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.

The final context in which reference collapsing takes place is uses of `decltype`. If, during analysis of a type involving `decltype`, a reference to a reference arises, reference collapsing will kick in to eliminate it. (For information about `decltype`, see Item 3.)

> **Things to Remember**
>
> - Reference collapsing occurs in four contexts: template instantiation, `auto` type generation, creation and use of `typedef`s and alias declarations, and `decltype`.
> - When compilers generate a reference to a reference in a reference collapsing context, the result becomes a single reference. If either of the original references is an lvalue reference, the result is an lvalue reference. Otherwise it's an rvalue reference.
> - Universal references are rvalue references in contexts where type deduction distinguishes lvalues from rvalues and where reference collapsing occurs.

Item 29: Assume that move operations are not present, not cheap, and not used.

Move semantics is arguably *the* premier feature of C++11. "Moving containers is now as cheap as copying pointers!" you're likely to hear, and "Copying temporary objects is now so efficient, coding to avoid it is tantamount to premature optimization!" Such sentiments are easy to understand. Move semantics is truly an important feature. It doesn't just allow compilers to replace expensive copy operations with comparatively cheap moves, it actually *requires* that they do so (when the proper conditions are fulfilled). Take your C++98 code base, recompile with a C++11-conformant compiler and Standard Library, and—*shazam!*—your software runs faster.

Move semantics can really pull that off, and that grants the feature an aura worthy of legend. Legends, however, are generally the result of exaggeration. The purpose of this Item is to keep your expectations grounded.

Let's begin with the observation that many types fail to support move semantics. The entire C++98 Standard Library was overhauled for C++11 to add move operations for types where moving could be implemented faster than copying, and the implementation of the library components was revised to take advantage of these operations, but chances are that you're working with a code base that has not been completely revised to take advantage of C++11. For types in your applications (or in the libraries you use) where no modifications for C++11 have been made, the existence of move support in your compilers is likely to do you little good. True, C++11 is willing to generate move operations for classes that lack them, but that happens only for classes declaring no copy operations, move operations, or destructors (see Item 17). Data members or base classes of types that have disabled moving (e.g., by deleting the move operations—see Item 11) will also suppress compiler-generated move operations. For types without explicit support for moving and that don't qualify for compiler-generated move operations, there is no reason to expect C++11 to deliver any kind of performance improvement over C++98.

Even types with explicit move support may not benefit as much as you'd hope. All containers in the standard C++11 library support moving, for example, but it would be a mistake to assume that moving all containers is cheap. For some containers, this is because there's no truly cheap way to move their contents. For others, it's because the truly cheap move operations the containers offer come with caveats the container elements can't satisfy.

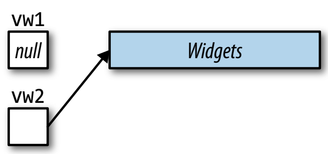
Consider `std::array`, a new container in C++11. `std::array` is essentially a built-in array with an STL interface. This is fundamentally different from the other standard containers, each of which stores its contents on the heap. Objects of such container types hold (as data members), conceptually, only a pointer to the heap memory storing the contents of the container. (The reality is more complex, but for purposes of this analysis, the differences are not important.) The existence of this pointer makes it possible to move the contents of an entire container in constant time: just copy the pointer to the container's contents from the source container to the target, and set the source's pointer to null:

```
std::vector<Widget> vw1;

// put data into vw1

…

// move vw1 into vw2. Runs in
// constant time. Only ptrs
// in vw1 and vw2 are modified
auto vw2 = std::move(vw1);
```



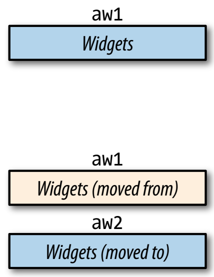
`std::array` objects lack such a pointer, because the data for a `std::array`'s contents are stored directly in the `std::array` object:

```
std::array<Widget, 10000> aw1;

// put data into aw1

…


// move aw1 into aw2. Runs in
// linear time. All elements in
// aw1 are moved into aw2
auto aw2 = std::move(aw1);
```



Note that the elements in `aw1` are *moved* into `aw2`. Assuming that `Widget` is a type where moving is faster than copying, moving a `std::array` of `Widget` will be faster than copying the same `std::array`. So `std::array` certainly offers move support. Yet both moving and copying a `std::array` have linear-time computational complexity, because each element in the container must be copied or moved. This is far from the "moving a container is now as cheap as assigning a couple of pointers" claim that one sometimes hears.

On the other hand, `std::string` offers constant-time moves and linear-time copies. That makes it sound like moving is faster than copying, but that may not be the case. Many string implementations employ the *small string optimization* (SSO). With the SSO, "small" strings (e.g., those with a capacity of no more than 15 characters) are stored in a buffer within the `std::string` object; no heap-allocated storage is used. Moving small strings using an SSO-based implementation is no faster than copying them, because the copy-only-a-pointer trick that generally underlies the performance advantage of moves over copies isn't applicable.

The motivation for the SSO is extensive evidence that short strings are the norm for many applications. Using an internal buffer to store the contents of such strings eliminates the need to dynamically allocate memory for them, and that's typically an efficiency win. An implication of the win, however, is that moves are no faster than copies, though one could just as well take a glass-half-full approach and say that for such strings, copying is no slower than moving.

Even for types supporting speedy move operations, some seemingly sure-fire move situations can end up making copies. Item 14 explains that some container operations in the Standard Library offer the strong exception safety guarantee and that to ensure that legacy C++98 code dependent on that guarantee isn't broken when upgrading to C++11, the underlying copy operations may be replaced with move operations only if the move operations are known to not throw. A consequence is that even if a type offers move operations that are more efficient than the corresponding copy operations, and even if, at a particular point in the code, a move operation would generally be appropriate (e.g., if the source object is an rvalue), compilers might still be forced to invoke a copy operation because the corresponding move operation isn't declared `noexcept`.

There are thus several scenarios in which C++11's move semantics do you no good:

- **No move operations:** The object to be moved from fails to offer move operations. The move request therefore becomes a copy request.
- **Move not faster:** The object to be moved from has move operations that are no faster than its copy operations.
- **Move not usable:** The context in which the moving would take place requires a move operation that emits no exceptions, but that operation isn't declared `noexcept`.

It's worth mentioning, too, another scenario where move semantics offers no efficiency gain:

- **Source object is lvalue:** With very few exceptions (see e.g., Item 25) only rvalues may be used as the source of a move operation.

But the title of this Item is to *assume* that move operations are not present, not cheap, and not used. This is typically the case in generic code, e.g., when writing templates, because you don't know all the types you're working with. In such circumstances, you must be as conservative about copying objects as you were in C++98—before move semantics existed. This is also the case for "unstable" code, i.e., code where the characteristics of the types being used are subject to relatively 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.

Perfect forwarding means we don't just forward objects, we also forward their salient characteristics: their types, whether they're lvalues or rvalues, and whether they're `const` or `volatile`. In conjunction with the observation that we'll be dealing with reference parameters, this implies that we'll be using universal references (see Item 24), because only universal reference parameters encode information about the lvalueness and rvalueness of the arguments that are passed to them.

Let's assume we have some function `f`, and we'd like to write a function (in truth, a function template) that forwards to it. The core of what we need looks like this:

```
template<typename T>
void fwd(T&& param)                    // accept any argument
{
  f(std::forward<T>(param));           // forward it to f
}
```

Forwarding functions are, by their nature, generic. The `fwd` template, for example, accepts any type of argument, and it forwards whatever it gets. A logical extension of this genericity is for forwarding functions to be not just templates, but *variadic* templates, thus accepting any number of arguments. The variadic form for `fwd` looks like this:

```
template<typename... Ts>
void fwd(Ts&&... params)               // accept any arguments
{
```

```
    f(std::forward<Ts>(params)...);    // forward them to f
}
```

This is the form you'll see in, among other places, the standard containers' emplacement functions (see Item 42) and the smart pointer factory functions, `std::make_shared` and `std::make_unique` (see Item 21).

Given our target function `f` and our forwarding function `fwd`, perfect forwarding *fails* if calling `f` with a particular argument does one thing, but calling `fwd` with the same argument does something different:

```
f( expression );        // if this does one thing,
fwd( expression );      // but this does something else, fwd fails
                        // to perfectly forward expression to f
```

Several kinds of arguments lead to this kind of failure. Knowing what they are and how to work around them is important, so let's tour the kinds of arguments that can't be perfect-forwarded.

Braced initializers

Suppose `f` is declared like this:

```
void f(const std::vector<int>& v);
```

In that case, calling `f` with a braced initializer compiles,

```
f({ 1, 2, 3 });         // fine, "{1, 2, 3}" implicitly
                        // converted to std::vector<int>
```

but passing the same braced initializer to `fwd` doesn't compile:

```
fwd({ 1, 2, 3 });       // error! doesn't compile
```

That's because the use of a braced initializer is a perfect forwarding failure case.

All such failure cases have the same cause. In a direct call to `f` (such as `f({ 1, 2, 3 })`), compilers see the arguments passed at the call site, and they see the types of the parameters declared by `f`. They compare the arguments at the call site to the parameter declarations to see if they're compatible, and, if necessary, they perform implicit conversions to make the call succeed. In the example above, they generate a temporary `std::vector<int>` object from `{ 1, 2, 3 }` so that `f`'s parameter `v` has a `std::vector<int>` object to bind to.

When calling `f` indirectly through the forwarding function template `fwd`, compilers no longer compare the arguments passed at `fwd`'s call site to the parameter declarations in `f`. Instead, they *deduce* the types of the arguments being passed to `fwd`, and they compare the deduced types to `f`'s parameter declarations. Perfect forwarding fails when either of the following occurs:

- **Compilers are unable to deduce a type** for one or more of `fwd`'s parameters. In this case, the code fails to compile.
- **Compilers deduce the "wrong" type** for one or more of `fwd`'s parameters. Here, "wrong" could mean that `fwd`'s instantiation won't compile with the types that were deduced, but it could also mean that the call to `f` using `fwd`'s deduced types behaves differently from a direct call to `f` with the arguments that were passed to `fwd`. One source of such divergent behavior would be if `f` were an overloaded function name, and, due to "incorrect" type deduction, the overload of `f` called inside `fwd` were different from the overload that would be invoked if `f` were called directly.

In the "`fwd({ 1, 2, 3 })`" call above, the problem is that passing a braced initializer to a function template parameter that's not declared to be a `std::initializer_list` is decreed to be, as the Standard puts it, a "non-deduced context." In plain English, that means that compilers are forbidden from deducing a type for the expression `{ 1, 2, 3 }` in the call to `fwd`, because `fwd`'s parameter isn't declared to be a `std::initializer_list`. Being prevented from deducing a type for `fwd`'s parameter, compilers must understandably reject the call.

Interestingly, Item 2 explains that type deduction succeeds for `auto` variables initialized with a braced initializer. Such variables are deemed to be `std::initializer_list` objects, and this affords a simple workaround for cases where the type the forwarding function should deduce is a `std::initializer_list`—declare a local variable using `auto`, then pass the local variable to the forwarding function:

```
auto il = { 1, 2, 3 };     // il's type deduced to be
                           // std::initializer_list<int>

fwd(il);                   // fine, perfect-forwards il to f
```

0 or NULL as null pointers

Item 8 explains that when you try to pass `0` or `NULL` as a null pointer to a template, type deduction goes awry, deducing an integral type (typically `int`) instead of a pointer type for the argument you pass. The result is that neither `0` nor `NULL` can be perfect-forwarded as a null pointer. The fix is easy, however: pass `nullptr` instead of `0` or `NULL`. For details, consult Item 8.

Declaration-only integral `static const` data members

As a general rule, there's no need to define integral `static const` data members in classes; declarations alone suffice. That's because compilers perform *const propagation* on such members' values, thus eliminating the need to set aside memory for them. For example, consider this code:

```
class Widget {
public:
  static const std::size_t MinVals = 28; // MinVals' declaration
  …
};
…                                        // no defn. for MinVals

std::vector<int> widgetData;
widgetData.reserve(Widget::MinVals);     // use of MinVals
```

Here, we're using `Widget::MinVals` (henceforth simply `MinVals`) to specify `widgetData`'s initial capacity, even though `MinVals` lacks a definition. Compilers work around the missing definition (as they are required to do) by plopping the value 28 into all places where `MinVals` is mentioned. The fact that no storage has been set aside for `MinVals`' value is unproblematic. If `MinVals`' address were to be taken (e.g., if somebody created a pointer to `MinVals`), then `MinVals` would require storage (so that the pointer had something to point to), and the code above, though it would compile, would fail at link-time until a definition for `MinVals` was provided.

With that in mind, imagine that `f` (the function `fwd` forwards its argument to) is declared like this:

```
void f(std::size_t val);
```

Calling `f` with `MinVals` is fine, because compilers will just replace `MinVals` with its value:

```
f(Widget::MinVals);          // fine, treated as "f(28)"
```

Alas, things may not go so smoothly if we try to call `f` through `fwd`:

```
fwd(Widget::MinVals);        // error! shouldn't link
```

This code will compile, but it shouldn't link. If that reminds you of what happens if we write code that takes `MinVals`' address, that's good, because the underlying problem is the same.

Although nothing in the source code takes `MinVals`' address, `fwd`'s parameter is a universal reference, and references, in the code generated by compilers, are usually treated like pointers. In the program's underlying binary code (and on the hardware), pointers and references are essentially the same thing. At this level, there's truth to the adage that references are simply pointers that are automatically dereferenced. That being the case, passing `MinVals` by reference is effectively the same as passing it by pointer, and as such, there has to be some memory for the pointer to point to. Passing integral `static const` data members by reference, then, generally requires that they be defined, and that requirement can cause code using perfect forwarding to fail where the equivalent code without perfect forwarding succeeds.

But perhaps you noticed the weasel words I sprinkled through the preceding discussion. The code "shouldn't" link. References are "usually" treated like pointers. Passing integral `static const` data members by reference "generally" requires that they be defined. It's almost like I know something I don't really want to tell you…

That's because I do. According to the Standard, passing `MinVals` by reference requires that it be defined. But not all implementations enforce this requirement. So, depending on your compilers and linkers, you may find that you can perfect-forward integral `static const` data members that haven't been defined. If you do, congratulations, but there is no reason to expect such code to port. To make it portable, simply provide a definition for the integral `static const` data member in question. For `MinVals`, that'd look like this:

```
const std::size_t Widget::MinVals;     // in Widget's .cpp file
```

Note that the definition doesn't repeat the initializer (28, in the case of `MinVals`). Don't stress over this detail, however. If you forget and provide the initializer in both places, your compilers will complain, thus reminding you to specify it only once.

Overloaded function names and template names

Suppose our function `f` (the one we keep wanting to forward arguments to via `fwd`) can have its behavior customized by passing it a function that does some of its work. Assuming this function takes and returns `int`s, `f` could be declared like this:

```
void f(int (*pf)(int));          // pf = "processing function"
```

It's worth noting that `f` could also be declared using a simpler non-pointer syntax. Such a declaration would look like this, though it'd have the same meaning as the declaration above:

```
void f(int pf(int));             // declares same f as above
```

Either way, now suppose we have an overloaded function, `processVal`:

```
int processVal(int value);
int processVal(int value, int priority);
```

We can pass `processVal` to `f`,

```
f(processVal);                    // fine
```

but it's something of a surprise that we can. f demands a pointer to a function as its argument, but processVal isn't a function pointer or even a function, it's the name of two different functions. However, compilers know which processVal they need: the one matching f's parameter type. They thus choose the processVal taking one int, and they pass that function's address to f.

What makes this work is that f's declaration lets compilers figure out which version of processVal is required. fwd, however, being a function template, doesn't have any information about what type it needs, and that makes it impossible for compilers to determine which overload should be passed:

```
fwd(processVal);              // error! which processVal?
```

processVal alone has no type. Without a type, there can be no type deduction, and without type deduction, we're left with another perfect forwarding failure case.

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 function, it represents *many* functions:

```
template<typename T>
T workOnVal(T param)          // template for processing values
{ … }

fwd(workOnVal);               // error! which workOnVal
                              // instantiation?
```

The way to get a perfect-forwarding function like fwd to accept an overloaded function name or a template name is to manually specify the overload or instantiation you want to have forwarded. For example, you can create a function pointer of the same type as f's parameter, initialize that pointer with processVal or workOnVal (thus causing the proper version of processVal to be selected or the proper instantiation of workOnVal to be generated), and pass the pointer to fwd:

```
using ProcessFuncType =                        // make typedef;
  int (*)(int);                                // see Item 9

ProcessFuncType processValPtr = processVal;    // specify needed
                                               // signature for
                                               // processVal

fwd(processValPtr);                            // fine

fwd(static_cast<ProcessFuncType>(workOnVal));  // also fine
```

Of course, this requires that you know the type of function pointer that `fwd` is forwarding to. It's not unreasonable to assume that a perfect-forwarding function will document that. After all, perfect-forwarding functions are designed to accept *anything*, so if there's no documentation telling you what to pass, how would you know?

Bitfields

The final failure case for perfect forwarding is when a bitfield is used as a function argument. To see what this means in practice, observe that an IPv4 header can be modeled as follows:[3]

```
struct IPv4Header {
  std::uint32_t version:4,
                IHL:4,
                DSCP:6,
                ECN:2,
                totalLength:16;
  …
};
```

If our long-suffering function `f` (the perennial target of our forwarding function `fwd`) is declared to take a `std::size_t` parameter, calling it with, say, the `totalLength` field of an `IPv4Header` object compiles without fuss:

```
void f(std::size_t sz);        // function to call

IPv4Header h;
…
f(h.totalLength);              // fine
```

Trying to forward `h.totalLength` to `f` via `fwd`, however, is a different story:

```
fwd(h.totalLength);            // error!
```

The problem is that `fwd`'s parameter is a reference, and `h.totalLength` is a non-`const` bitfield. That may not sound so bad, but the C++ Standard condemns the combination in unusually clear prose: "A non-`const` reference shall not be bound to a bit-field." There's an excellent reason for the prohibition. Bitfields may consist of arbitrary parts of machine words (e.g., bits 3-5 of a 32-bit `int`), but there's no way to directly address such things. I mentioned earlier that references and pointers are the same thing at the hardware level, and just as there's no way to create a pointer to

3 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.

arbitrary bits (C++ dictates that the smallest thing you can point to is a `char`), there's no way to bind a reference to arbitrary bits, either.

Working around the impossibility of perfect-forwarding a bitfield is easy, once you realize that any function that accepts a bitfield as an argument will receive a *copy* of the bitfield's value. After all, no function can bind a reference to a bitfield, nor can any function accept pointers to bitfields, because pointers to bitfields don't exist. The only kinds of parameters to which a bitfield can be passed are by-value parameters and, interestingly, references-to-`const`. In the case of by-value parameters, the called function obviously receives a copy of the value in the bitfield, and it turns out that in the case of a reference-to-`const` parameter, the Standard requires that the reference actually bind to a *copy* of the bitfield's value that's stored in an object of some standard integral type (e.g., `int`). References-to-`const` don't bind to bitfields, they bind to "normal" objects into which the values of the bitfields have been copied.

The key to passing a bitfield into a perfect-forwarding function, then, is to take advantage of the fact that the forwarded-to function will always receive a copy of the bitfield's value. You can thus make a copy yourself and call the forwarding function with the copy. In the case of our example with `IPv4Header`, this code would do the trick:

```
// copy bitfield value; see Item 6 for info on init. form
auto length = static_cast<std::uint16_t>(h.totalLength);

fwd(length);                        // forward the copy
```

Upshot

In most cases, perfect forwarding works exactly as advertised. You rarely have to think about it. But when it doesn't work—when reasonable-looking code fails to compile or, worse, compiles, but doesn't behave the way you anticipate—it's important to know about perfect forwarding's imperfections. Equally important is knowing how to work around them. In most cases, this is straightforward.

Things to Remember

- Perfect forwarding fails when template type deduction fails or when it deduces the wrong type.
- The kinds of arguments that lead to perfect forwarding failure are braced initializers, null pointers expressed as `0` or `NULL`, declaration-only integral `const static` data members, template and overloaded function names, and bitfields.

CHAPTER 6

Lambda Expressions

Lambda expressions—*lambdas*—are a game changer in C++ programming. That's somewhat surprising, because they bring no new expressive power to the language. Everything a lambda can do is something you can do by hand with a bit more typing. But lambdas are such a convenient way to create function objects, the impact on day-to-day C++ software development is enormous. Without lambdas, the STL "`_if`" algorithms (e.g., `std::find_if`, `std::remove_if`, `std::count_if`, etc.) tend to be employed with only the most trivial predicates, but when lambdas are available, use of these algorithms with nontrivial conditions blossoms. The same is true of algorithms that can be customized with comparison functions (e.g., `std::sort`, `std::nth_element`, `std::lower_bound`, etc.). Outside the STL, lambdas make it possible to quickly create custom deleters for `std::unique_ptr` and `std::shared_ptr` (see Items 18 and 19), and they make the specification of predicates for condition variables in the threading API equally straightforward (see Item 39). Beyond the Standard Library, lambdas facilitate the on-the-fly specification of callback functions, interface adaption functions, and context-specific functions for one-off calls. Lambdas really make C++ a more pleasant programming language.

The vocabulary associated with lambdas can be confusing. Here's a brief refresher:

- A *lambda expression* is just that: an expression. It's part of the source code. In

  ```
  std::find_if(container.begin(), container.end(),
               [](int val) { return 0 < val && val < 10; });
  ```

  the highlighted expression is the lambda.

- A *closure* is the runtime object created by a lambda. Depending on the capture mode, closures hold copies of or references to the captured data. In the call to `std::find_if` above, the closure is the object that's passed at runtime as the third argument to `std::find_if`.

- A *closure class* is a class from which a closure is instantiated. Each lambda causes compilers to generate a unique closure class. The statements inside a lambda become executable instructions in the member functions of its closure class.

A lambda is often used to create a closure that's used only as an argument to a function. That's the case in the call to `std::find_if` above. However, closures may generally be copied, so it's usually possible to have multiple closures of a closure type corresponding to a single lambda. For example, in the following code,

```
{

  int x;                                // x is local variable
  …

  auto c1 =                             // c1 is copy of the
    [x](int y) { return x * y > 55; };  // closure produced
                                        // by the lambda

  auto c2 = c1;                         // c2 is copy of c1

  auto c3 = c2;                         // c3 is copy of c2

  …

}
```

`c1`, `c2`, and `c3` are all copies of the closure produced by the lambda.

Informally, it's perfectly acceptable to blur the lines between lambdas, closures, and closure classes. But in the Items that follow, it's often important to distinguish what exists during compilation (lambdas and closure classes), what exists at runtime (closures), and how they relate to one another.

Item 31: Avoid default capture modes.

There are two default capture modes in C++11: by-reference and by-value. Default by-reference capture can lead to dangling references. Default by-value capture lures you into thinking you're immune to that problem (you're not), and it lulls you into thinking your closures are self-contained (they may not be).

That's the executive summary for this Item. If you're more engineer than executive, you'll want some meat on those bones, so let's start with the danger of default by-reference capture.

A by-reference capture causes a closure to contain a reference to a local variable or to a parameter that's available in the scope where the lambda is defined. If the lifetime of a closure created from that lambda exceeds the lifetime of the local variable or parameter, the reference in the closure will dangle. For example, suppose we have a container of filtering functions, each of which takes an `int` and returns a `bool` indicating whether a passed-in value satisfies the filter:

```
using FilterContainer =                     // see Item 9 for
  std::vector<std::function<bool(int)>>;    // "using", Item 2
                                            // for std::function

FilterContainer filters;                    // filtering funcs
```

We could add a filter for multiples of 5 like this:

```
filters.emplace_back(                       // see Item 42 for
  [](int value) { return value % 5 == 0; }  // info on
);                                          // emplace_back
```

However, it may be that we need to compute the divisor at runtime, i.e., we can't just hard-code 5 into the lambda. So adding the filter might look more like this:

```
void addDivisorFilter()
{
  auto calc1 = computeSomeValue1();
  auto calc2 = computeSomeValue2();

  auto divisor = computeDivisor(calc1, calc2);

  filters.emplace_back(                                // danger!
    [&](int value) { return value % divisor == 0; }  // ref to
  );                                                   // divisor
}                                                      // will
                                                       // dangle!
```

This code is a problem waiting to happen. The lambda refers to the local variable `divisor`, but that variable ceases to exist when `addDivisorFilter` returns. That's immediately after `filters.emplace_back` returns, so the function that's added to `filters` is essentially dead on arrival. Using that filter yields undefined behavior from virtually the moment it's created.

Now, the same problem would exist if `divisor`'s by-reference capture were explicit,

```
filters.emplace_back(
  [&divisor](int value)                 // danger! ref to
  { return value % divisor == 0; }      // divisor will
);                                      // still dangle!
```

but with an explicit capture, it's easier to see that the viability of the lambda is dependent on divisor's lifetime. Also, writing out the name, "divisor," reminds us to ensure that divisor lives at least as long as the lambda's closures. That's a more specific memory jog than the general "make sure nothing dangles" admonition that "[&]" conveys.

If you know that a closure will be used immediately (e.g., by being passed to an STL algorithm) and won't be copied, there is no risk that references it holds will outlive the local variables and parameters in the environment where its lambda is created. In that case, you might argue, there's no risk of dangling references, hence no reason to avoid a default by-reference capture mode. For example, our filtering lambda might be used only as an argument to C++11's std::all_of, which returns whether all elements in a range satisfy a condition:

```
template<typename C>
void workWithContainer(const C& container)
{
  auto calc1 = computeSomeValue1();            // as above
  auto calc2 = computeSomeValue2();            // as above

  auto divisor = computeDivisor(calc1, calc2);  // as above

  using ContElemT = typename C::value_type;     // type of
                                                // elements in
                                                // container

  using std::begin;                             // for
  using std::end;                               // genericity;
                                                // see Item 13

  if (std::all_of(                              // if all values
        begin(container), end(container),       // in container
        [&](const ContElemT& value)             // are multiples
        { return value % divisor == 0; })       // of divisor...
      ) {
    …                                           // they are...
  } else {
    …                                           // at least one
  }                                             // isn't...
}
```

It's true, this is safe, but its safety is somewhat precarious. If the lambda were found to be useful in other contexts (e.g., as a function to be added to the filters container) and was copy-and-pasted into a context where its closure could outlive divi sor, you'd be back in dangle-city, and there'd be nothing in the capture clause to specifically remind you to perform lifetime analysis on divisor.

Long-term, it's simply better software engineering to explicitly list the local variables and parameters that a lambda depends on.

By the way, the ability to use auto in C++14 lambda parameter specifications means that the code above can be simplified in C++14. The ContElemT typedef can be eliminated, and the if condition can be revised as follows:

```
if (std::all_of(begin(container), end(container),
                [&](const auto& value)                // C++14
                { return value % divisor == 0; }))
```

One way to solve our problem with divisor would be a default by-value capture mode. That is, we could add the lambda to filters as follows:

```
filters.emplace_back(                                  // now
  [=](int value) { return value % divisor == 0; }      // divisor
);                                                     // can't
                                                       // dangle
```

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 by value, you copy the pointer into the closures arising from the lambda, but you don't prevent code outside the lambda from deleteing the pointer and causing your copies to dangle.

"That could never happen!" you protest. "Having read Chapter 4, I worship at the house of smart pointers. Only loser C++98 programmers use raw pointers and delete." That may be true, but it's irrelevant because you do, in fact, use raw pointers, and they can, in fact, be deleted out from under you. It's just that in your modern C++ programming style, there's often little sign of it in the source code.

Suppose one of the things Widgets can do is add entries to the container of filters:

```
class Widget {
public:
  …                                  // ctors, etc.
  void addFilter() const;            // add an entry to filters

private:
  int divisor;                       // used in Widget's filter
};
```

Widget::addFilter could be defined like this:

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
  );
}
```

To the blissfully uninitiated, this looks like safe code. The lambda is dependent on `divisor`, but the default by-value capture mode ensures that `divisor` is copied into any closures arising from the lambda, right?

Wrong. Completely wrong. Horribly wrong. Fatally wrong.

Captures apply only to non-`static` local variables (including parameters) visible in the scope where the lambda is created. In the body of `Widget::addFilter`, `divisor` is not a local variable, it's a data member of the `Widget` class. It can't be captured. Yet if the default capture mode is eliminated, the code won't compile:

```
void Widget::addFilter() const
{
  filters.emplace_back(                               // error!
    [](int value) { return value % divisor == 0; }    // divisor
  );                                                  // not
}                                                     // available
```

Furthermore, if an attempt is made to explicitly capture `divisor` (either by value or by reference—it doesn't matter), the capture won't compile, because `divisor` isn't a local variable or a parameter:

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [divisor](int value)                 // error! no local
    { return value % divisor == 0; }     // divisor to capture
  );
}
```

So if the default by-value capture clause isn't capturing `divisor`, yet without the default by-value capture clause, the code won't compile, what's going on?

The explanation hinges on the implicit use of a raw pointer: `this`. Every non-`static` member function has a `this` pointer, and you use that pointer every time you mention a data member of the class. Inside any `Widget` member function, for example, compilers internally replace uses of `divisor` with `this->divisor`. In the version of `Widget::addFilter` with a default by-value capture,

```
void Widget::addFilter() const
{
  filters.emplace_back(
    [=](int value) { return value % divisor == 0; }
  );
}
```

what's being captured is the `Widget`'s `this` pointer, not `divisor`. Compilers treat the code as if it had been written as follows:

```
void Widget::addFilter() const
{
  auto currentObjectPtr = this;

  filters.emplace_back(
    [currentObjectPtr](int value)
    { return value % currentObjectPtr->divisor == 0; }
  );
}
```

Understanding this is tantamount to understanding that the viability of the closures arising from this lambda is tied to the lifetime of the `Widget` whose `this` pointer they contain a copy of. In particular, consider this code, which, in accord with Chapter 4, uses pointers of only the smart variety:

```
using FilterContainer =                     // as before
  std::vector<std::function<bool(int)>>;

FilterContainer filters;                    // as before

void doSomeWork()
{
  auto pw =                       // create Widget; see
    std::make_unique<Widget>();   // Item 21 for
                                  // std::make_unique

  pw->addFilter();                // add filter that uses
                                  // Widget::divisor

  …
}                                 // destroy Widget; filters
                                  // now holds dangling pointer!
```

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

  filters.emplace_back(
    [=](int value)                           // capture the copy
    { return value % divisorCopy == 0; }     // use the copy
  );
}
```

but why tempt fate? A default capture mode is what made it possible to accidentally capture `this` when you thought you were capturing `divisor` in the first place.

In C++14, a better way to capture a data member is to use generalized lambda capture (see Item 32):

```
void Widget::addFilter() const
{
  filters.emplace_back(                  // C++14:
    [divisor = divisor](int value)       // copy divisor to closure
    { return value % divisor == 0; }     // use the copy
  );
}
```

There's no such thing as a default capture mode for a generalized lambda capture, however, so even in C++14, the advice of this Item—to avoid default capture modes—stands.

An additional drawback to default by-value captures is that they can suggest that the corresponding closures are self-contained and insulated from changes to data outside the closures. In general, that's not true, because lambdas may be dependent not just on local variables and parameters (which may be captured), but also on objects with *static storage duration*. Such objects are defined at global or namespace scope or are declared `static` inside classes, functions, or files. These objects can be used inside lambdas, but they can't be captured. Yet specification of a default by-value capture mode can lend the impression that they are. Consider this revised version of the `addDivisorFilter` function we saw earlier:

```
void addDivisorFilter()
{
  static auto calc1 = computeSomeValue1();      // now static
  static auto calc2 = computeSomeValue2();      // now static

  static auto divisor =                         // now static
    computeDivisor(calc1, calc2);

  filters.emplace_back(
    [=](int value)                     // captures nothing!
    { return value % divisor == 0; }   // refers to above static
  );

  ++divisor;                           // modify divisor
}
```

A casual reader of this code could be forgiven for seeing "`[=]`" and thinking, "Okay, the lambda makes a copy of all the objects it uses and is therefore self-contained." But it's not self-contained. This lambda doesn't use any non-static local variables, so nothing is captured. Rather, the code for the lambda refers to the `static` variable `divisor`. When, at the end of each invocation of `addDivisorFilter`, `divisor` is incremented, any lambdas that have been added to `filters` via this function will exhibit new behavior (corresponding to the new value of `divisor`). Practically speaking, this lambda captures `divisor` by reference, a direct contradiction to what the default by-value capture clause seems to imply. If you stay away from default by-value capture clauses, you eliminate the risk of your code being misread in this way.

Things to Remember

- Default by-reference capture can lead to dangling references.
- Default by-value capture is susceptible to dangling pointers (especially `this`), and it misleadingly suggests that lambdas are self-contained.

Item 32: Use init capture to move objects into closures.

Sometimes neither by-value capture nor by-reference capture is what you want. If you have a move-only object (e.g., a `std::unique_ptr` or a `std::future`) that you want to get into a closure, C++11 offers no way to do it. If you have an object that's expensive to copy but cheap to move (e.g., most containers in the Standard Library), and you'd like to get that object into a closure, you'd much rather move it than copy it. Again, however, C++11 gives you no way to accomplish that.

But that's C++11. C++14 is a different story. It offers direct support for moving objects into closures. If your compilers are C++14-compliant, rejoice and read on. If you're still working with C++11 compilers, you should rejoice and read on, too, because there are ways to approximate move capture in C++11.

The absence of move capture was recognized as a shortcoming even as C++11 was adopted. The straightforward remedy would have been to add it in C++14, but the Standardization Committee chose a different path. They introduced a new capture mechanism that's so flexible, capture-by-move is only one of the tricks it can perform. The new capability is called *init capture*. It can do virtually everything the C++11 capture forms can do, plus more. The one thing you can't express with an init capture is a default capture mode, but Item 31 explains that you should stay away from those, anyway. (For situations covered by C++11 captures, init capture's syntax is a bit wordier, so in cases where a C++11 capture gets the job done, it's perfectly reasonable to use it.)

Using an init capture makes it possible for you to specify

1. **the name of a data member** in the closure class generated from the lambda and
2. **an expression** initializing that data member.

Here's how you can use init capture to move a `std::unique_ptr` into a closure:

```
class Widget {                          // some useful type
public:
  …

  bool isValidated() const;
  bool isProcessed() const;
  bool isArchived() const;

private:
  …
};
```

```
auto pw = std::make_unique<Widget>();    // create Widget; see
                                         // Item 21 for info on
                                         // std::make_unique

…                                        // configure *pw

auto func = [pw = std::move(pw)]                // init data mbr
            { return pw->isValidated()          // in closure w/
                     && pw->isArchived(); };    // std::move(pw)
```

The highlighted text comprises the init capture. To the left of the "=" is the name of the data member in the closure class you're specifying, and to the right is the initializing expression. Interestingly, the scope on the left of the "=" is different from the scope on the right. The scope on the left is that of the closure class. The scope on the right is the same as where the lambda is being defined. In the example above, the name `pw` on the left of the "=" refers to a data member in the closure class, while the name `pw` on the right refers to the object declared above the lambda, i.e., the variable initialized by the call to `std::make_unique`. So "`pw = std::move(pw)`" means "create a data member `pw` in the closure, and initialize that data member with the result of applying `std::move` to the local variable `pw`."

As usual, code in the body of the lambda is in the scope of the closure class, so uses of `pw` there refer to the closure class data member.

The comment "configure `*pw`" in this example indicates that after the `Widget` is created by `std::make_unique` and before the `std::unique_ptr` to that `Widget` is captured by the lambda, the `Widget` is modified in some way. If no such configuration is necessary, i.e., if the `Widget` created by `std::make_unique` is in a state suitable to be captured by the lambda, the local variable `pw` is unnecessary, because the closure class's data member can be directly initialized by `std::make_unique`:

```
auto func = [pw = std::make_unique<Widget>()]  // init data mbr
            { return pw->isValidated()          // in closure w/
                     && pw->isArchived(); };    // result of call
                                                // to make_unique
```

This should make clear that the C++14 notion of "capture" is considerably generalized from C++11, because in C++11, it's not possible to capture the result of an expression. As a result, another name for init capture is *generalized lambda capture*.

But what if one or more of the compilers you use lacks support for C++14's init capture? How can you accomplish move capture in a language lacking support for move capture?

Remember that a lambda expression is simply a way to cause a class to be generated and an object of that type to be created. There is nothing you can do with a 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:

```
class IsValAndArch {                          // "is validated
public:                                       // and archived"
  using DataType = std::unique_ptr<Widget>;

  explicit IsValAndArch(DataType&& ptr)       // Item 25 explains
  : pw(std::move(ptr)) {}                     // use of std::move

  bool operator()() const
  { return pw->isValidated() && pw->isArchived(); }

private:
  DataType pw;
};

auto func = IsValAndArch(std::make_unique<Widget>());
```

That's more work than writing the lambda, but it doesn't change the fact that if you want a class in C++11 that supports move-initialization of its data members, the only thing between you and your desire is a bit of time with your keyboard.

If you want to stick with lambdas (and given their convenience, you probably do), move capture can be emulated in C++11 by

1. **moving the object to be captured into a function object produced by `std::bind`** and
2. **giving the lambda a reference to the "captured" object**.

If you're familiar with `std::bind`, the code is pretty straightforward. If you're not familiar with `std::bind`, the code takes a little getting used to, but it's worth the trouble.

Suppose you'd like to create a local `std::vector`, put an appropriate set of values into it, then move it into a closure. In C++14, this is easy:

```
std::vector<double> data;                 // object to be moved
                                          // into closure

…                                         // populate data

auto func = [data = std::move(data)]      // C++14 init capture
            { /* uses of data */ };
```

I've highlighted key parts of this code: the type of object you want to move (`std::vector<double>`), the name of that object (`data`), and the initializing expression for the init capture (`std::move(data)`). The C++11 equivalent is as follows, where I've highlighted the same key things:

```
std::vector<double> data;                    // as above

…                                            // as above

auto func =
  std::bind(                                 // C++11 emulation
    [](const std::vector<double>& data)      // of init capture
    { /* uses of data */ },
    std::move(data)
  );
```

Like lambda expressions, `std::bind` produces function objects. I call function objects returned by `std::bind` *bind objects*. The first argument to `std::bind` is a callable object. Subsequent arguments represent values to be passed to that object.

A bind object contains copies of all the arguments passed to `std::bind`. For each lvalue argument, the corresponding object in the bind object is copy constructed. For each rvalue, it's move constructed. In this example, the second argument is an rvalue (the result of `std::move`—see Item 23), so `data` is move constructed into the bind object. This move construction is the crux of move capture emulation, because moving an rvalue into a bind object is how we work around the inability to move an rvalue into a C++11 closure.

When a bind object is "called" (i.e., its function call operator is invoked) the arguments it stores are passed to the callable object originally passed to `std::bind`. In this example, that means that when `func` (the bind object) is called, the move-constructed copy of `data` inside `func` is passed as an argument to the lambda that was passed to `std::bind`.

This lambda is the same as the lambda we'd use in C++14, except a parameter, `data`, has been added to correspond to our pseudo-move-captured object. This parameter is an lvalue reference to the copy of `data` in the bind object. (It's not an rvalue reference, because although the expression used to initialize the copy of `data` ("`std::move(data)`") is an rvalue, the copy of `data` itself is an lvalue.) Uses of `data` inside the lambda will thus operate on the move-constructed copy of `data` inside the bind object.

By default, the `operator()` member function inside the closure class generated from a lambda is `const`. That has the effect of rendering all data members in the closure const within the body of the lambda. The move-constructed copy of data inside the bind object is not const, however, so to prevent that copy of data from being modified inside the lambda, the lambda's parameter is declared reference-to-const. If the lambda were declared mutable, operator() in its closure class would not be declared const, and it would be appropriate to omit const in the lambda's parameter declaration:

```
auto func =
  std::bind(                                 // C++11 emulation
    [](std::vector<double>& data) mutable   // of init capture
    { /* uses of data */ },                  // for mutable lambda
    std::move(data)
  );
```

Because a bind object stores copies of all the arguments passed to std::bind, the bind object in our example contains a copy of the closure produced by the lambda that is its first argument. The lifetime of the closure is therefore the same as the lifetime of the bind object. That's important, because it means that as long as the closure exists, the bind object containing the pseudo-move-captured object exists, too.

If this is your first exposure to std::bind, you may need to consult your favorite C++11 reference before all the details of the foregoing discussion fall into place. Even if that's the case, these fundamental points should be clear:

- It's not possible to move-construct an object into a C++11 closure, but it is possible to move-construct an object into a C++11 bind object.
- Emulating move-capture in C++11 consists of move-constructing an object into a bind object, then passing the move-constructed object to the lambda by reference.
- Because the lifetime of the bind object is the same as that of the closure, it's possible to treat objects in the bind object as if they were in the closure.

As a second example of using std::bind to emulate move capture, here's the C++14 code we saw earlier to create a std::unique_ptr in a closure:

```
auto func = [pw = std::make_unique<Widget>()]    // as before,
            { return pw->isValidated()           // create pw
                     && pw->isArchived(); };     // in closure
```

And here's the C++11 emulation:

```
auto func = std::bind(
              [](const std::unique_ptr<Widget>& pw)
              { return pw->isValidated()
                     && pw->isArchived(); },
```

```
                    std::make_unique<Widget>()
                  );
```

It's ironic that I'm showing how to use `std::bind` to work around limitations in C++11 lambdas, because in Item 34, I advocate the use of lambdas over `std::bind`. However, that Item explains that there are some cases in C++11 where `std::bind` can be useful, and this is one of them. (In C++14, features such as init capture and `auto` parameters eliminate those cases.)

Things to Remember

- Use C++14's init capture to move objects into closures.
- In C++11, emulate init capture via hand-written classes or `std::bind`.

Item 33: Use `decltype` on `auto&&` parameters to `std::forward` them.

One of the most exciting features of C++14 is *generic lambdas*—lambdas that use `auto` in their parameter specifications. The implementation of this feature is straightforward: `operator()` in the lambda's closure class is a template. Given this lambda, for example,

```
auto f = [](auto x){ return func(normalize(x)); };
```

the closure class's function call operator looks like this:

```
class SomeCompilerGeneratedClassName {
public:
  template<typename T>                   // see Item 3 for
  auto operator()(T x) const             // auto return type
  { return func(normalize(x)); }

  …                                      // other closure class
};                                       // functionality
```

In this example, the only thing the lambda does with its parameter `x` is forward it to `normalize`. If `normalize` treats lvalues differently from rvalues, this lambda isn't written properly, because it always passes an lvalue (the parameter `x`) to `normalize`, even if the argument that was passed to the lambda was an rvalue.

The correct way to write the lambda is to have it perfect-forward `x` to `normalize`. Doing that requires two changes to the code. First, `x` has to become a universal reference (see Item 24), and second, it has to be passed to `normalize` via `std::forward` (see Item 25). In concept, these are trivial modifications:

```
auto f = [](auto&& x)
         { return func(normalize(std::forward<???>(x))); };
```

Between concept and realization, however, is the question of what type to pass to `std::forward`, i.e., to determine what should go where I've written `???` above.

Normally, when you employ perfect forwarding, you're in a template function taking a type parameter `T`, so you just write `std::forward<T>`. In the generic lambda, though, there's no type parameter `T` available to you. There is a `T` in the templatized `operator()` inside the closure class generated by the lambda, but it's not possible to refer to it from the lambda, so it does you no good.

Item 28 explains that if an lvalue argument is passed to a universal reference parameter, the type of that parameter becomes an lvalue reference. If an rvalue is passed, the parameter becomes an rvalue reference. This means that in our lambda, we can determine whether the argument passed was an lvalue or an rvalue by inspecting the type of the parameter `x`. `decltype` gives us a way to do that (see Item 3). If an lvalue was passed in, `decltype(x)` will produce a type that's an lvalue reference. If an rvalue was passed, `decltype(x)` will produce an rvalue reference type.

Item 28 also explains that when calling `std::forward`, convention dictates that the type argument be an lvalue reference to indicate an lvalue and a non-reference to indicate an rvalue. In our lambda, if `x` is bound to an lvalue, `decltype(x)` will yield an lvalue reference. That conforms to convention. However, if `x` is bound to an rvalue, `decltype(x)` will yield an rvalue reference instead of the customary non-reference.

But look at the sample C++14 implementation for `std::forward` from Item 28:

```
template<typename T>                        // in namespace
T&& forward(remove_reference_t<T>& param)   // std
{
  return static_cast<T&&>(param);
}
```

If client code wants to perfect-forward an rvalue of type `Widget`, it normally instantiates `std::forward` with the type `Widget` (i.e, a non-reference type), and the `std::forward` template yields this function:

```
Widget&& forward(Widget& param)            // instantiation of
{                                          // std::forward when
  return static_cast<Widget&&>(param);     // T is Widget
}
```

But consider what would happen if the client code wanted to perfect-forward the same rvalue of type `Widget`, but instead of following the convention of specifying `T` to be a non-reference type, it specified it to be an rvalue reference. That is, consider what would happen if `T` were specified to be `Widget&&`. After initial instantiation of `std::forward` and application of `std::remove_reference_t`, but before reference collapsing (once again, see Item 28), `std::forward` would look like this:

```
Widget&& && forward(Widget& param)          // instantiation of
{                                            // std::forward when
  return static_cast<Widget&& &&>(param);   // T is Widget&&
}                                            // (before reference-
                                             // collapsing)
```

Applying the reference-collapsing rule that an rvalue reference to an rvalue reference becomes a single rvalue reference, this instantiation emerges:

```
Widget&& forward(Widget& param)             // instantiation of
{                                            // std::forward when
  return static_cast<Widget&&>(param);      // T is Widget&&
}                                            // (after reference-
                                             // collapsing)
```

If you compare this instantiation with the one that results when `std::forward` is called with `T` set to `Widget`, you'll see that they're identical. That means that instantiating `std::forward` with an rvalue reference type yields the same result as instantiating it with a non-reference type.

That's wonderful news, because `decltype(x)` yields an rvalue reference type when an rvalue is passed as an argument to our lambda's parameter `x`. We established above that when an lvalue is passed to our lambda, `decltype(x)` yields the customary type to pass to `std::forward`, and now we realize that for rvalues, `decltype(x)` yields a type to pass to `std::forward` that's not conventional, but that nevertheless yields the same outcome as the conventional type. So for both lvalues and rvalues, passing `decltype(x)` to `std::forward` gives us the result we want. Our perfect-forwarding lambda can therefore be written like this:

```
auto f =
  [](auto&& param)
  {
    return
      func(normalize(std::forward<decltype(param)>(param)));
  };
```

From there, it's just a hop, skip, and six dots to a perfect-forwarding lambda that accepts not just a single parameter, but any number of parameters, because C++14 lambdas can also be variadic:

```
auto f =
  [](auto&&... params)
  {
    return
    func(normalize(std::forward<decltype(params)>(params)...));
  };
```

Things to Remember

- Use `decltype` on `auto&&` parameters to `std::forward` them.

Item 34: Prefer lambdas to `std::bind`.

`std::bind` is the C++11 successor to C++98's `std::bind1st` and `std::bind2nd`, but, informally, it's been part of the Standard Library since 2005. That's when the Standardization Committee adopted a document known as TR1, which included `bind`'s specification. (In TR1, `bind` was in a different namespace, so it was `std::tr1::bind`, not `std::bind`, and a few interface details were different.) This history means that some programmers have a decade or more of experience using `std::bind`. If you're one of them, you may be reluctant to abandon a tool that's served you well. That's understandable, but in this case, change is good, because in C++11, lambdas are almost always a better choice than `std::bind`. As of C++14, the case for lambdas isn't just stronger, it's downright ironclad.

This Item assumes that you're familiar with `std::bind`. If you're not, you'll want to acquire a basic understanding before continuing. Such an understanding is worthwhile in any case, because you never know when you might encounter uses of `std::bind` in a code base you have to read or maintain.

As in Item 32, I refer to the function objects returned from `std::bind` as *bind objects*.

The most important reason to prefer lambdas over `std::bind` is that lambdas are more readable. Suppose, for example, we have a function to set up an audible alarm:

```
// typedef for a point in time (see Item 9 for syntax)
using Time = std::chrono::steady_clock::time_point;

// see Item 10 for "enum class"
enum class Sound { Beep, Siren, Whistle };

// typedef for a length of time
```

```
using Duration = std::chrono::steady_clock::duration;

// at time t, make sound s for duration d
void setAlarm(Time t, Sound s, Duration d);
```

Further suppose that at some point in the program, we've determined we'll want an alarm that will go off an hour after it's set and that will stay on for 30 seconds. The alarm sound, however, remains undecided. We can write a lambda that revises `setAlarm`'s interface so that only a sound needs to be specified:

```
// setSoundL ("L" for "lambda") is a function object allowing a
// sound to be specified for a 30-sec alarm to go off an hour
// after it's set
auto setSoundL =
  [](Sound s)
  {
    // make std::chrono components available w/o qualification
    using namespace std::chrono;

    setAlarm(steady_clock::now() + hours(1),  // alarm to go off
             s,                               // in an hour for
             seconds(30));                    // 30 seconds
  };
```

I've highlighted the call to `setAlarm` inside the lambda. This is a normal-looking function call, and even a reader with little lambda experience can see that the parameter `s` passed to the lambda is passed as an argument to `setAlarm`.

We can streamline this code in C++14 by availing ourselves of the standard suffixes for seconds (`s`), milliseconds (`ms`), hours (`h`), etc., that build on C++11's support for user-defined literals. These suffixes are implemented in the `std::literals` namespace, so the above code can be rewritten as follows:

```
auto setSoundL =
  [](Sound s)
  {
    using namespace std::chrono;
    using namespace std::literals;         // for C++14 suffixes

    setAlarm(steady_clock::now() + 1h,     // C++14, but
             s,                            // same meaning
             30s);                         // as above
  };
```

Our first attempt to write the corresponding `std::bind` call is below. It has an error that we'll fix in a moment, but the correct code is more complicated, and even this simplified version brings out some important issues:

```
using namespace std::chrono;           // as above
using namespace std::literals;

using namespace std::placeholders;     // needed for use of "_1"

auto setSoundB =                       // "B" for "bind"
  std::bind(setAlarm,
            steady_clock::now() + 1h,  // incorrect! see below
            _1,
            30s);
```

I'd like to highlight the call to `setAlarm` here as I did in the lambda, but there's no call to highlight. Readers of this code simply have to know that calling `setSoundB` invokes `setAlarm` with the time and duration specified in the call to `std::bind`. To the uninitiated, the placeholder "`_1`" is essentially magic, but even readers in the know have to mentally map from the number in that placeholder to its position in the `std::bind` parameter list in order to understand that the first argument in a call to `setSoundB` is passed as the second argument to `setAlarm`. The type of this argument is not identified in the call to `std::bind`, so readers have to consult the `setAlarm` declaration to determine what kind of argument to pass to `setSoundB`.

But, as I said, the code isn't quite right. In the lambda, it's clear that the expression "`steady_clock::now() + 1h`" is an argument to `setAlarm`. It will be evaluated when `setAlarm` is called. That makes sense: we want the alarm to go off an hour after invoking `setAlarm`. In the `std::bind` call, however, "`steady_clock::now() + 1h`" is passed as an argument to `std::bind`, not to `setAlarm`. That means that the expression will be evaluated when `std::bind` is called, and the time resulting from that expression will be stored inside the resulting bind object. As a consequence, the alarm will be set to go off an hour *after the call to `std::bind`*, not an hour after the call to `setAlarm`!

Fixing the problem requires telling `std::bind` to defer evaluation of the expression until `setAlarm` is called, and the way to do that is to nest a second call to `std::bind` inside the first one:

```
auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<>(), steady_clock::now(), 1h),
            _1,
            30s);
```

If you're familiar with the `std::plus` template from C++98, you may be surprised to see that in this code, no type is specified between the angle brackets, i.e., the code contains "`std::plus<>`", not "`std::plus<`*type*`>`". In C++14, the template type argument for the standard operator templates can generally be omitted, so there's no need to provide it here. C++11 offers no such feature, so the C++11 `std::bind` equivalent to the lambda is:

```
using namespace std::chrono;                  // as above
using namespace std::placeholders;

auto setSoundB =
  std::bind(setAlarm,
            std::bind(std::plus<steady_clock::time_point>(),
                      steady_clock::now(),
                      hours(1)),
            _1,
            seconds(30));
```

If, at this point, the lambda's not looking a lot more attractive, you should probably have your eyesight checked.

When `setAlarm` is overloaded, a new issue arises. Suppose there's an overload taking a fourth parameter specifying the alarm volume:

```
enum class Volume { Normal, Loud, LoudPlusPlus };

void setAlarm(Time t, Sound s, Duration d, Volume v);
```

The lambda continues to work as before, because overload resolution chooses the three-argument version of `setAlarm`:

```
auto setSoundL =                              // same as before
  [](Sound s)
  {
    using namespace std::chrono;

    setAlarm(steady_clock::now() + 1h,        // fine, calls
             s,                               // 3-arg version
             30s);                            // of setAlarm
  };
```

The `std::bind` call, on the other hand, now fails to compile:

```
auto setSoundB =                              // error! which
  std::bind(setAlarm,                         // setAlarm?
            std::bind(std::plus<>(),
                      steady_clock::now(),
```

```
                  1h),
        _1,
        30s);
```

The problem is that compilers have no way to determine which of the two `setAlarm` functions they should pass to `std::bind`. All they have is a function name, and the name alone is ambiguous.

To get the `std::bind` call to compile, `setAlarm` must be cast to the proper function pointer type:

```
using SetAlarm3ParamType = void(*)(Time t, Sound s, Duration d);

auto setSoundB =                                             // now
  std::bind(static_cast<SetAlarm3ParamType>(setAlarm),   // okay
            std::bind(std::plus<>(),
                      steady_clock::now(),
                      1h),
            _1,
            30s);
```

But this brings up another difference between lambdas and `std::bind`. Inside the function call operator for `setSoundL` (i.e., the function call operator of the lambda's closure class), the call to `setAlarm` is a normal function invocation that can be inlined by compilers in the usual fashion:

```
setSoundL(Sound::Siren);      // body of setAlarm may
                              // well be inlined here
```

The call to `std::bind`, however, passes a function pointer to `setAlarm`, and that means that inside the function call operator for `setSoundB` (i.e., the function call operator for the bind object), the call to `setAlarm` takes place through a function pointer. Compilers are less likely to inline function calls through function pointers, and that means that calls to `setAlarm` through `setSoundB` are less likely to be fully inlined than those through `setSoundL`:

```
setSoundB(Sound::Siren);      // body of setAlarm is less
                              // likely to be inlined here
```

It's thus possible that using lambdas generates faster code than using `std::bind`.

The `setAlarm` example involves only a simple function call. If you want to do anything more complicated, the scales tip even further in favor of lambdas. For example, consider this C++14 lambda, which returns whether its argument is between a minimum value (`lowVal`) and a maximum value (`highVal`), where `lowVal` and `highVal` are local variables:

```
auto betweenL =
  [lowVal, highVal]
  (const auto& val)                          // C++14
  { return lowVal <= val && val <= highVal; };
```

std::bind can express the same thing, but the construct is an example of job security through code obscurity:

```
using namespace std::placeholders;           // as above

auto betweenB =
  std::bind(std::logical_and<>(),            // C++14
            std::bind(std::less_equal<>(), lowVal, _1),
            std::bind(std::less_equal<>(), _1, highVal));
```

In C++11, we'd have to specify the types we wanted to compare, and the std::bind call would then look like this:

```
auto betweenB =                              // C++11 version
  std::bind(std::logical_and<bool>(),
            std::bind(std::less_equal<int>(), lowVal, _1),
            std::bind(std::less_equal<int>(), _1, highVal));
```

Of course, in C++11, the lambda couldn't take an auto parameter, so it'd have to commit to a type, too:

```
auto betweenL =                              // C++11 version
  [lowVal, highVal]
  (int val)
  { return lowVal <= val && val <= highVal; };
```

Either way, I hope we can agree that the lambda version is not just shorter, but also more comprehensible and maintainable.

Earlier, I remarked that for those with little std::bind experience, its placeholders (e.g., _1, _2, etc.) are essentially magic. But it's not just the behavior of the placeholders that's opaque. Suppose we have a function to create compressed copies of Widgets,

```
enum class CompLevel { Low, Normal, High };  // compression
                                             // level

Widget compress(const Widget& w,             // make compressed
                CompLevel lev);              // copy of w
```

and we want to create a function object that allows us to specify how much a particular Widget w should be compressed. This use of std::bind will create such an object:

```
Widget w;

using namespace std::placeholders;

auto compressRateB = std::bind(compress, w, _1);
```

Now, when we pass `w` to `std::bind`, it has to be stored for the later call to `compress`. It's stored inside the object `compressRateB`, but how is it stored—by value or by reference? It makes a difference, because if `w` is modified between the call to `std::bind` and a call to `compressRateB`, storing `w` by reference will reflect the changes, while storing it by value won't.

The answer is that it's stored by value,[1] but the only way to know that is to memorize how `std::bind` works; there's no sign of it in the call to `std::bind`. Contrast that with a lambda approach, where whether `w` is captured by value or by reference is explicit:

```
auto compressRateL =                          // w is captured by
  [w](CompLevel lev)                          // value; lev is
  { return compress(w, lev); };               // passed by value
```

Equally explicit is how parameters are passed to the lambda. Here, it's clear that the parameter `lev` is passed by value. Hence:

```
compressRateL(CompLevel::High);               // arg is passed
                                              // by value
```

But in the call to the object resulting from `std::bind`, how is the argument passed?

```
compressRateB(CompLevel::High);               // how is arg
                                              // passed?
```

Again, the only way to know is to memorize how `std::bind` works. (The answer is that all arguments passed to bind objects are passed by reference, because the function call operator for such objects uses perfect forwarding.)

Compared to lambdas, then, code using `std::bind` is less readable, less expressive, and possibly less efficient. In C++14, there are no reasonable use cases for `std::bind`. In C++11, however, `std::bind` can be justified in two constrained situations:

1 `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.

- **Move capture**. C++11 lambdas don't offer move capture, but it can be emulated through a combination of a lambda and `std::bind`. For details, consult Item 32, which also explains that in C++14, lambdas' support for init capture eliminates the need for the emulation.
- **Polymorphic function objects**. Because the function call operator on a bind object uses perfect forwarding, it can accept arguments of any type (modulo the restrictions on perfect forwarding described in Item 30). This can be useful when you want to bind an object with a templatized function call operator. For example, given this class,

  ```
  class PolyWidget {
  public:
      template<typename T>
      void operator()(const T& param);
      …
  };
  ```

  `std::bind` can bind a `PolyWidget` as follows:

  ```
  PolyWidget pw;

  auto boundPW = std::bind(pw, _1);
  ```

  `boundPW` can then be called with different types of arguments:

  ```
  boundPW(1930);              // pass int to
                              // PolyWidget::operator()

  boundPW(nullptr);           // pass nullptr to
                              // PolyWidget::operator()

  boundPW("Rosebud");         // pass string literal to
                              // PolyWidget::operator()
  ```

  There is no way to do this with a C++11 lambda. In C++14, however, it's easily achieved via a lambda with an `auto` parameter:

  ```
  auto boundPW = [pw](const auto& param)    // C++14
                 { pw(param); };
  ```

These are edge cases, of course, and they're transient edge cases at that, because compilers supporting C++14 lambdas are increasingly common.

When `bind` was unofficially added to C++ in 2005, it was a big improvement over its 1998 predecessors. The addition of lambda support to C++11 rendered `std::bind` all but obsolete, however, and as of C++14, there are just no good use cases for it.

Things to Remember

- Lambdas are more readable, more expressive, and may be more efficient than using `std::bind`.
- In C++11 only, `std::bind` may be useful for implementing move capture or for binding objects with templatized function call operators.

CHAPTER 7
The Concurrency API

One of C++11's great triumphs is the incorporation of concurrency into the language and library. Programmers familiar with other threading APIs (e.g., pthreads or Windows threads) are sometimes surprised at the comparatively Spartan feature set that C++ offers, but that's because a great deal of C++'s support for concurrency is in the form of constraints on compiler-writers. The resulting language assurances mean that for the first time in C++'s history, programmers can write multithreaded programs with standard behavior across all platforms. This establishes a solid foundation on which expressive libraries can be built, and the concurrency elements of the Standard Library (tasks, futures, threads, mutexes, condition variables, atomic objects, and more) are merely the beginning of what is sure to become an increasingly rich set of tools for the development of concurrent C++ software.

In the Items that follow, bear in mind that the Standard Library has two templates for futures: `std::future` and `std::shared_future`. In many cases, the distinction is not important, so I often simply talk about *futures*, by which I mean both kinds.

Item 35: Prefer task-based programming to thread-based.

If you want to run a function `doAsyncWork` asynchronously, you have two basic choices. You can create a `std::thread` and run `doAsyncWork` on it, thus employing a *thread-based* approach:

```
int doAsyncWork();

std::thread t(doAsyncWork);
```

Or you can pass `doAsyncWork` to `std::async`, a strategy known as *task-based*:

```
auto fut = std::async(doAsyncWork);        // "fut" for "future"
```

In such calls, the function object passed to `std::async` (e.g., `doAsyncWork`) is considered a *task*.

The task-based approach is typically superior to its thread-based counterpart, and the tiny amount of code we've seen already demonstrates some reasons why. Here, `doAsyncWork` produces a return value, which we can reasonably assume the code invoking `doAsyncWork` is interested in. With the thread-based invocation, there's no straightforward way to get access to it. With the task-based approach, it's easy, because the future returned from `std::async` offers the `get` function. The `get` function is even more important if `doAsyncWork` emits an exception, because `get` provides access to that, too. With the thread-based approach, if `doAsyncWork` throws, the program dies (via a call to `std::terminate`).

A more fundamental difference between thread-based and task-based programming is the higher level of abstraction that task-based embodies. It frees you from the details of thread management, an observation that reminds me that I need to summarize the three meanings of "thread" in concurrent C++ software:

- *Hardware threads* are the threads that actually perform computation. Contemporary machine architectures offer one or more hardware threads per CPU core.
- *Software threads* (also known as *OS threads* or *system threads*) are the threads that the operating system[1] manages across all processes and schedules for execution on hardware threads. It's typically possible to create more software threads than hardware threads, because when a software thread is blocked (e.g., on I/O or waiting for a mutex or condition variable), throughput can be improved by executing other, unblocked, threads.
- *`std::thread`s* are objects in a C++ process that act as handles to underlying software threads. Some `std::thread` objects represent "null" handles, i.e., correspond to no software thread, because they're in a default-constructed state (hence have no function to execute), have been moved from (the moved-to `std::thread` then acts as the handle to the underlying software thread), have been `join`ed (the function they were to run has finished), or have been `detach`ed (the connection between them and their underlying software thread has been severed).

Software threads are a limited resource. If you try to create more than the system can provide, a `std::system_error` exception is thrown. This is true even if the function you want to run can't throw. For example, even if `doAsyncWork` is `noexcept`,

1 Assuming you have one. Some embedded systems don't.

```
int doAsyncWork() noexcept;           // see Item 14 for noexcept
```

this statement could result in an exception:

```
std::thread t(doAsyncWork);           // throws if no more
                                      // threads are available
```

Well-written software must somehow deal with this possibility, but how? One approach is to run `doAsyncWork` on the current thread, but that could lead to unbalanced loads and, if the current thread is a GUI thread, responsiveness issues. Another option is to wait for some existing software threads to complete and then try to create a new `std::thread` again, but it's possible that the existing threads are waiting for an action that `doAsyncWork` is supposed to perform (e.g., produce a result or notify a condition variable).

Even if you don't run out of threads, you can have trouble with *oversubscription*. That's when there are more ready-to-run (i.e., unblocked) software threads than hardware threads. When that happens, the thread scheduler (typically part of the OS) time-slices the software threads on the hardware. When one thread's time-slice is finished and another's begins, a context switch is performed. Such context switches increase the overall thread management overhead of the system, and they can be particularly costly when the hardware thread on which a software thread is scheduled is on a different core than was the case for the software thread during its last time-slice. In that case, (1) the CPU caches are typically cold for that software thread (i.e., they contain little data and few instructions useful to it) and (2) the running of the "new" software thread on that core "pollutes" the CPU caches for "old" threads that had been running on that core and are likely to be scheduled to run there again.

Avoiding oversubscription is difficult, because the optimal ratio of software to hardware threads depends on how often the software threads are runnable, and that can change dynamically, e.g., when a program goes from an I/O-heavy region to a computation-heavy region. The best ratio of software to hardware threads is also dependent on the cost of context switches and how effectively the software threads use the CPU caches. Furthermore, the number of hardware threads and the details of the CPU caches (e.g., how large they are and their relative speeds) depend on the machine architecture, so even if you tune your application to avoid oversubscription (while still keeping the hardware busy) on one platform, there's no guarantee that your solution will work well on other kinds of machines.

Your life will be easier if you dump these problems on somebody else, and using `std::async` does exactly that:

```
auto fut = std::async(doAsyncWork);    // onus of thread mgmt is
                                       // on implementer of
                                       // the Standard Library
```

This call shifts the thread management responsibility to the implementer of the C++ Standard Library. For example, the likelihood of receiving an out-of-threads exception is significantly reduced, because this call will probably never yield one. "How can that be?" you might wonder. "If I ask for more software threads than the system can provide, why does it matter whether I do it by creating `std::thread`s or by calling `std::async`?" It matters, because `std::async`, when called in this form (i.e., with the default launch policy—see Item 36), doesn't guarantee that it will create a new software thread. Rather, it permits the scheduler to arrange for the specified function (in this example, `doAsyncWork`) to be run on the thread requesting `doAsyncWork`'s result (i.e., on the thread calling `get` or `wait` on `fut`), and reasonable schedulers take advantage of that freedom if the system is oversubscribed or is out of threads.

If you pulled this "run it on the thread needing the result" trick yourself, I remarked that it could lead to load-balancing issues, and those issues don't go away simply because it's `std::async` and the runtime scheduler that confront them instead of you. When it comes to load balancing, however, the runtime scheduler is likely to have a more comprehensive picture of what's happening on the machine than you do, because it manages the threads from all processes, not just the one your code is running in.

With `std::async`, responsiveness on a GUI thread can still be problematic, because the scheduler has no way of knowing which of your threads has tight responsiveness requirements. In that case, you'll want to pass the `std::launch::async` launch policy to `std::async`. That will ensure that the function you want to run really executes on a different thread (see Item 36).

State-of-the-art thread schedulers employ system-wide thread pools to avoid oversubscription, and they improve load balancing across hardware cores through work-stealing algorithms. The C++ Standard does not require the use of thread pools or work-stealing, and, to be honest, there are some technical aspects of the C++11 concurrency specification that make it more difficult to employ them than we'd like. Nevertheless, some vendors take advantage of this technology in their Standard Library implementations, and it's reasonable to expect that progress will continue in this area. If you take a task-based approach to your concurrent programming, you automatically reap the benefits of such technology as it becomes more widespread. If, on the other hand, you program directly with `std::thread`s, you assume the burden of dealing with thread exhaustion, oversubscription, and load balancing yourself, not to mention how your solutions to these problems mesh with the solutions implemented in programs running in other processes on the same machine.

Compared to thread-based programming, a task-based design spares you the travails of manual thread management, and it provides a natural way to examine the results of asynchronously executed functions (i.e., return values or exceptions). Nevertheless, there are some situations where using threads directly may be appropriate. They include:

- **You need access to the API of the underlying threading implementation**. The C++ concurrency API is typically implemented using a lower-level platform-specific API, usually pthreads or Windows' Threads. Those APIs are currently richer than what C++ offers. (For example, C++ has no notion of thread priorities or affinities.) To provide access to the API of the underlying threading implementation, `std::thread` objects typically offer the `native_handle` member function. There is no counterpart to this functionality for `std::future`s (i.e., for what `std::async` returns).
- **You need to and are able to optimize thread usage for your application**. This could be the case, for example, if you're developing server software with a known execution profile that will be deployed as the only significant process on a machine with fixed hardware characteristics.
- **You need to implement threading technology beyond the C++ concurrency API**, e.g., thread pools on platforms where your C++ implementations don't offer them.

These are uncommon cases, however. Most of the time, you should choose task-based designs instead of programming with threads.

Things to Remember

- The `std::thread` API offers no direct way to get return values from asynchronously run functions, and if those functions throw, the program is terminated.
- Thread-based programming calls for manual management of thread exhaustion, oversubscription, load balancing, and adaptation to new platforms.
- Task-based programming via `std::async` with the default launch policy handles most of these issues for you.

Item 36: Specify `std::launch::async` if asynchronicity is essential.

When you call `std::async` to execute a function (or other callable object), you're generally intending to run the function asynchronously. But that's not necessarily what you're asking `std::async` to do. You're really requesting that the function be run in accord with a `std::async` *launch policy*. There are two standard policies, each represented by an enumerator in the `std::launch` scoped `enum`. (See Item 10 for information on scoped `enum`s.) Assuming a function `f` is passed to `std::async` for execution,

- **The `std::launch::async` launch policy** means that `f` must be run asynchronously, i.e., on a different thread.
- **The `std::launch::deferred` launch policy** means that `f` may run only when `get` or `wait` is called on the future returned by `std::async`.[2] That is, `f`'s execution is *deferred* until such a call is made. When `get` or `wait` is invoked, `f` will execute synchronously, i.e., the caller will block until `f` finishes running. If neither `get` nor `wait` is called, `f` will never run.

Perhaps surprisingly, `std::async`'s default launch policy—the one it uses if you don't expressly specify one—is neither of these. Rather, it's these or-ed together. The following two calls have exactly the same meaning:

```
auto fut1 = std::async(f);                     // run f using
                                               // default launch
                                               // policy

auto fut2 = std::async(std::launch::async |    // run f either
                       std::launch::deferred,  // async or
                       f);                     // deferred
```

The default policy thus permits `f` to be run either asynchronously or synchronously. As Item 35 points out, this flexibility permits `std::async` and the thread-management components of the Standard Library to assume responsibility for thread creation and destruction, avoidance of oversubscription, and load balancing. That's among the things that make concurrent programming with `std::async` so convenient.

But using `std::async` with the default launch policy has some interesting implications. Given a thread `t` executing this statement,

```
auto fut = std::async(f);   // run f using default launch policy
```

2 This is a simplification. 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`.

- **It's not possible to predict whether `f` will run concurrently with `t`**, because `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.

The default launch policy's scheduling flexibility often mixes poorly with the use of `thread_local` variables, because it means that if `f` reads or writes such *thread-local storage* (TLS), it's not possible to predict which thread's variables will be accessed:

```
auto fut = std::async(f);         // TLS for f possibly for
                                  // independent thread, but
                                  // possibly for thread
                                  // invoking get or wait on fut
```

It also affects `wait`-based loops using timeouts, because calling `wait_for` or `wait_until` on a task (see Item 35) that's deferred yields the value `std::launch::deferred`. This means that the following loop, which looks like it should eventually terminate, may, in reality, run forever:

```
using namespace std::literals;       // for C++14 duration
                                     // suffixes; see Item 34

void f()                             // f sleeps for 1 second,
{                                    // then returns
  std::this_thread::sleep_for(1s);
}

auto fut = std::async(f);            // run f asynchronously
                                     // (conceptually)

while (fut.wait_for(100ms) !=        // loop until f has
       std::future_status::ready)    // finished running...
{                                    // which may never happen!
  …
}
```

If `f` runs concurrently with the thread calling `std::async` (i.e., if the launch policy chosen for `f` is `std::launch::async`), there's no problem here (assuming `f` eventually finishes), but if `f` is deferred, `fut.wait_for` will always return `std::future_status::deferred`. That will never be equal to `std::future_status::ready`, so the loop will never terminate.

This kind of bug is easy to overlook during development and unit testing, because it may manifest itself only under heavy loads. Those are the conditions that push the machine towards oversubscription or thread exhaustion, and that's when a task may be most likely to be deferred. After all, if the hardware isn't threatened by oversubscription or thread exhaustion, there's no reason for the runtime system not to schedule the task for concurrent execution.

The fix is simple: just check the future corresponding to the `std::async` call to see whether the task is deferred, and, if so, avoid entering the timeout-based loop. Unfortunately, there's no direct way to ask a future whether its task is deferred. Instead, you have to call a timeout-based function—a function such as `wait_for`. In this case, you don't really want to wait for anything, you just want to see if the return value is `std::future_status::deferred`, so stifle your mild disbelief at the necessary circumlocution and call `wait_for` with a zero timeout:

```
auto fut = std::async(f);                  // as above

if (fut.wait_for(0s) ==                    // if task is
    std::future_status::deferred)          // deferred...
{
                        // ...use wait or get on fut
  …                     // to call f synchronously


} else {                // task isn't deferred
  while (fut.wait_for(100ms) !=            // infinite loop not
         std::future_status::ready) {      // possible (assuming
                                           // f finishes)

    …                  // task is neither deferred nor ready,
                       // so do concurrent work until it's ready
  }

  …                    // fut is ready

}
```

The upshot of these various considerations is that using `std::async` with the default launch policy for a task is fine as long as the following conditions are fulfilled:

- The task need not run concurrently with the thread calling `get` or `wait`.
- It doesn't matter which thread's `thread_local` variables are read or written.
- Either there's a guarantee that `get` or `wait` will be called on the future returned by `std::async` or it's acceptable that the task may never execute.
- Code using `wait_for` or `wait_until` takes the possibility of deferred status into account.

If any of these conditions fails to hold, you probably want to guarantee that `std::async` will schedule the task for truly asynchronous execution. The way to do that is to pass `std::launch::async` as the first argument when you make the call:

```
auto fut = std::async(std::launch::async, f);  // launch f
                                               // asynchronously
```

In fact, having a function that acts like `std::async`, but that automatically uses `std::launch::async` as the launch policy, is a convenient tool to have around, so it's nice that it's easy to write. Here's the C++11 version:

```
template<typename F, typename... Ts>
inline
std::future<typename std::result_of<F(Ts...)>::type>
reallyAsync(F&& f, Ts&&... params)       // return future
{                                        // for asynchronous
  return std::async(std::launch::async,  // call to f(params...)
                    std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

This function receives a callable object `f` and zero or more parameters `params` and perfect-forwards them (see Item 25) to `std::async`, passing `std::launch::async` as the launch policy. Like `std::async`, it returns a `std::future` for the result of invoking `f` on `params`. Determining the type of that result is easy, because the type trait `std::result_of` gives it to you. (See Item 9 for general information on type traits.)

`reallyAsync` is used just like `std::async`:

```
auto fut = reallyAsync(f);         // run f asynchronously;
                                   // throw if std::async
                                   // would throw
```

In C++14, the ability to deduce `reallyAsync`'s return type streamlines the function declaration:

```
template<typename F, typename... Ts>
inline
auto                                              // C++14
reallyAsync(F&& f, Ts&&... params)
{
  return std::async(std::launch::async,
                    std::forward<F>(f),
                    std::forward<Ts>(params)...);
}
```

This version makes it crystal clear that `reallyAsync` does nothing but invoke `std::async` with the `std::launch::async` launch policy.

Things to Remember

- The default launch policy for `std::async` permits both asynchronous and synchronous task execution.
- This flexibility leads to uncertainty when accessing `thread_local`s, implies that the task may never execute, and affects program logic for timeout-based `wait` calls.
- Specify `std::launch::async` if asynchronous task execution is essential.

Item 37: Make `std::threads` unjoinable on all paths.

Every `std::thread` object is in one of two states: *joinable* or *unjoinable*. A joinable `std::thread` corresponds to an underlying asynchronous thread of execution that is or could be running. A `std::thread` corresponding to an underlying thread that's blocked or waiting to be scheduled is joinable, for example. `std::thread` objects corresponding to underlying threads that have run to completion are also considered joinable.

An unjoinable `std::thread` is what you'd expect: a `std::thread` that's not joinable. Unjoinable `std::thread` objects include:

- **Default-constructed `std::threads`**. Such `std::threads` have no function to execute, hence don't correspond to an underlying thread of execution.

- **`std::thread` objects that have been moved from**. The result of a move is that the underlying thread of execution a `std::thread` used to correspond to (if any) now corresponds to a different `std::thread`.
- **`std::thread`s that have been `join`ed**. After a `join`, the `std::thread` object no longer corresponds to the underlying thread of execution that has finished running.
- **`std::thread`s that have been `detach`ed**. A `detach` severs the connection between a `std::thread` object and the underlying thread of execution it corresponds to.

One reason a `std::thread`'s joinability is important is that if the destructor for a joinable thread is invoked, execution of the program is terminated. For example, suppose we have a function `doWork` that takes a filtering function, `filter`, and a maximum value, `maxVal`, as parameters. `doWork` checks to make sure that all conditions necessary for its computation are satisfied, then performs the computation with all the values between 0 and `maxVal` that pass the filter. If it's time-consuming to do the filtering and it's also time-consuming to determine whether `doWork`'s conditions are satisfied, it would be reasonable to do those two things concurrently.

Our preference would be to employ a task-based design for this (see Item 35), but let's assume we'd like to set the priority of the thread doing the filtering. Item 35 explains that that requires use of the thread's native handle, and that's accessible only through the `std::thread` API; the task-based API (i.e., futures) doesn't provide it. Our approach will therefore be based on threads, not tasks.

We could come up with code like this:

```
constexpr auto tenMillion = 10000000;         // see Item 15
                                              // for constexpr

bool doWork(std::function<bool(int)> filter,  // returns whether
            int maxVal = tenMillion)          // computation was
{                                             // performed; see
                                              // Item 2 for
                                              // std::function

  std::vector<int> goodVals;                  // values that
                                              // satisfy filter

  std::thread t([&filter, maxVal, &goodVals]  // populate
                {                             // goodVals
                  for (auto i = 0; i <= maxVal; ++i)
                   { if (filter(i)) goodVals.push_back(i); }
```

```
               });

  auto nh = t.native_handle();                 // use t's native
  …                                            // handle to set
                                               // t's priority
  if (conditionsAreSatisfied()) {
    t.join();                                  // let t finish
    performComputation(goodVals);
    return true;                               // computation was
  }                                            // performed

  return false;                                // computation was
}                                              // not performed
```

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 C++14's ability to use an apostrophe as a digit separator:

```
constexpr auto tenMillion = 10'000'000;        // C++14
```

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 design would be to start `t` in a suspended state (thus making it possible to adjust its priority before it does any computation), but I don't want to distract you with that code. If you're more distracted by the code's absence, turn to Item 39, because it shows how to start threads suspended.

But back to `doWork`. If *`conditionsAreSatisfied()`* returns `true`, all is well, but if it returns `false` or throws an exception, the `std::thread` object `t` will be joinable when its destructor is called at the end of `doWork`. That would cause program execution to be terminated.

You might wonder why the `std::thread` destructor behaves this way. It's because the two other obvious options are arguably worse. They are:

- **An implicit `join`**. In this case, a `std::thread`'s destructor would wait for its underlying asynchronous thread of execution to complete. That sounds reasonable, but it could lead to performance anomalies that would be difficult to track down. For example, it would be counterintuitive that `doWork` would wait for its filter to be applied to all values if *`conditionsAreSatisfied()`* had already returned `false`.
- **An implicit `detach`**. In this case, a `std::thread`'s destructor would sever the connection between the `std::thread` object and its underlying thread of execution. The underlying thread would continue to run. This sounds no less reason-

able than the `join` approach, but the debugging problems it can lead to are worse. In `doWork`, for example, `goodVals` is a local variable that is captured by reference. It's also modified inside the lambda (via the call to `push_back`). Suppose, then, that while the lambda is running asynchronously, *`conditionsAreSatisfied()`* returns `false`. In that case, `doWork` would return, and its local variables (including `goodVals`) would be destroyed. Its stack frame would be popped, and execution of its thread would continue at `doWork`'s call site.

Statements following that call site would, at some point, make additional function calls, and at least one such call would probably end up using some or all of the memory that had once been occupied by the `doWork` stack frame. Let's call such a function `f`. While `f` was running, the lambda that `doWork` initiated would still be running asynchronously. That lambda could call `push_back` on the stack memory that used to be `goodVals` but that is now somewhere inside `f`'s stack frame. Such a call would modify the memory that used to be `goodVals`, and that means that from `f`'s perspective, the content of memory in its stack frame could spontaneously change! Imagine the fun you'd have debugging *that*.

The Standardization Committee decided that the consequences of destroying a joinable thread were sufficiently dire that they essentially banned it (by specifying that destruction of a joinable thread causes program termination).

This puts the onus on you to ensure that if you use a `std::thread` object, it's made unjoinable on every path out of the scope in which it's defined. But covering every path can be complicated. It includes flowing off the end of the scope as well as jumping out via a `return`, `continue`, `break`, `goto` or exception. That can be a lot of paths.

Any time you want to perform some action along every path out of a block, the normal approach is to put that action in the destructor of a local object. Such objects are known as *RAII objects*, and the classes they come from are known as *RAII classes*. (*RAII* itself stands for "Resource Acquisition Is Initialization," although the crux of the technique is destruction, not initialization). RAII classes are common in the Standard Library. Examples include the STL containers (each container's destructor destroys the container's contents and releases its memory), the standard smart pointers (Items 18–20 explain that `std::unique_ptr`'s destructor invokes its deleter on the object it points to, and the destructors in `std::shared_ptr` and `std::weak_ptr` decrement reference counts), `std::fstream` objects (their destructors close the files they correspond to), and many more. And yet there is no standard RAII class for `std::thread` objects, perhaps because the Standardization Committee, having rejected both `join` and `detach` as default options, simply didn't know what such a class should do.

Fortunately, it's not difficult to write one yourself. For example, the following class allows callers to specify whether `join` or `detach` should be called when a `ThreadRAII` object (an RAII object for a `std::thread`) is destroyed:

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };    // see Item 10 for
                                             // enum class info

  ThreadRAII(std::thread&& t, DtorAction a)  // in dtor, take
  : action(a), t(std::move(t)) {}            // action a on t

  ~ThreadRAII()
  {                                          // see below for
    if (t.joinable()) {                      // joinability test

      if (action == DtorAction::join) {
        t.join();
      } else {
        t.detach();
      }

    }
  }

  std::thread& get() { return t; }           // see below

private:
  DtorAction action;
  std::thread t;
};
```

I hope this code is largely self-explanatory, but the following points may be helpful:

- The constructor accepts only `std::thread` rvalues, because we want to move the passed-in `std::thread` into the `ThreadRAII` object. (Recall that `std::thread` objects aren't copyable.)
- The parameter order in the constructor is designed to be intuitive to callers (specifying the `std::thread` first and the destructor action second makes more sense than vice versa), but the member initialization list is designed to match the order of the data members' declarations. That order puts the `std::thread` object last. In this class, the order makes no difference, but in general, it's possible for the initialization of one data member to depend on another, and because `std::thread` objects may start running a function immediately after they are

initialized, it's a good habit to declare them last in a class. That guarantees that at the time they are constructed, all the data members that precede them have already been initialized and can therefore be safely accessed by the asynchronously running thread that corresponds to the `std::thread` data member.

- `ThreadRAII` offers a `get` function to provide access to the underlying `std::thread` object. This is analogous to the `get` functions offered by the standard smart pointer classes that give access to their underlying raw pointers. Providing `get` avoids the need for `ThreadRAII` to replicate the full `std::thread` interface, and it also means that `ThreadRAII` objects can be used in contexts where `std::thread` objects are required.
- Before the `ThreadRAII` destructor invokes a member function on the `std::thread` object `t`, it checks to make sure that `t` is joinable. This is necessary, because invoking `join` or `detach` on an unjoinable thread yields undefined behavior. It's possible that a client constructed a `std::thread`, created a `ThreadRAII` object from it, used `get` to acquire access to `t`, and then did a move from `t` or called `join` or `detach` on it. Each of those actions would render `t` unjoinable.

  If you're worried that in this code,

  ```
  if (t.joinable()) {

    if (action == DtorAction::join) {
      t.join();
    } else {
      t.detach();
    }
  }
  ```

  a race exists, because between execution of `t.joinable()` and invocation of `join` or `detach`, another thread could render `t` unjoinable, your intuition is commendable, but your fears are unfounded. A `std::thread` object can change state from joinable to unjoinable only through a member function call, e.g., `join`, `detach`, or a move operation. At the time a `ThreadRAII` object's destructor is invoked, no other thread should be making member function calls on that object. If there are simultaneous calls, there is certainly a race, but it isn't inside the destructor, it's in the client code that is trying to invoke two member functions (the destructor and something else) on one object at the same time. In general, simultaneous member function calls on a single object are safe only if all are to `const` member functions (see Item 16).

Employing `ThreadRAII` in our `doWork` example would look like this:

```
bool doWork(std::function<bool(int)> filter,  // as before
            int maxVal = tenMillion)
{
  std::vector<int> goodVals;                  // as before

  ThreadRAII t(                               // use RAII object
    std::thread([&filter, maxVal, &goodVals]
                {
                  for (auto i = 0; i <= maxVal; ++i)
                    { if (filter(i)) goodVals.push_back(i); }
                }),
                ThreadRAII::DtorAction::join  // RAII action
  );

  auto nh = t.get().native_handle();
  …

  if (conditionsAreSatisfied()) {
    t.get().join();
    performComputation(goodVals);
    return true;
  }

  return false;
}
```

In this case, we've chosen to do a `join` on the asynchronously running thread in the `ThreadRAII` destructor, because, as we saw earlier, doing a `detach` could lead to some truly nightmarish debugging. We also saw earlier that doing a `join` could lead to performance anomalies (that, to be frank, could also be unpleasant to debug), but given a choice between undefined behavior (which `detach` would get us), program termination (which use of a raw `std::thread` would yield), or performance anomalies, performance anomalies seems like the best of a bad lot.

Alas, Item 39 demonstrates that using `ThreadRAII` to perform a `join` on `std::thread` destruction can sometimes lead not just to a performance anomaly, but to a hung program. The "proper" solution to these kinds of problems would be to communicate to the asynchronously running lambda that we no longer need its work and that it should return early, but there's no support in C++11 for *interruptible* *threads*. They can be implemented by hand, but that's a topic beyond the scope of this book.[3]

Item 17 explains that because `ThreadRAII` declares a destructor, there will be no compiler-generated move operations, but there is no reason `ThreadRAII` objects shouldn't be movable. If compilers were to generate these functions, the functions would do the right thing, so explicitly requesting their creation isappropriate:

```
class ThreadRAII {
public:
  enum class DtorAction { join, detach };          // as before

  ThreadRAII(std::thread&& t, DtorAction a)        // as before
  : action(a), t(std::move(t)) {}

  ~ThreadRAII()
  {
    …                                               // as before
  }

  ThreadRAII(ThreadRAII&&) = default;              // support
  ThreadRAII& operator=(ThreadRAII&&) = default;   // moving

  std::thread& get() { return t; }                 // as before

private:                                           // as before
  DtorAction action;
  std::thread t;
};
```

Things to Remember

- Make `std::threads` unjoinable on all paths.
- `join`-on-destruction can lead to difficult-to-debug performance anomalies.
- `detach`-on-destruction can lead to difficult-to-debug undefined behavior.
- Declare `std::thread` objects last in lists of data members.

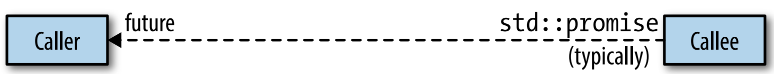
3 You'll find a nice treatment in Anthony Williams' *C++ Concurrency in Action* (Manning Publications, 2012), section 9.2.

Item 38: Be aware of varying thread handle destructor behavior.

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.

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.

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.[4] 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 result using a future. You can think of it as follows, where the dashed arrow shows the flow of information from callee to caller:



But where is the callee's result stored? The callee could finish before the caller invokes `get` on a corresponding future, so the result can't be stored in the callee's `std::promise`. That object, being local to the callee, would be destroyed when the callee finished.

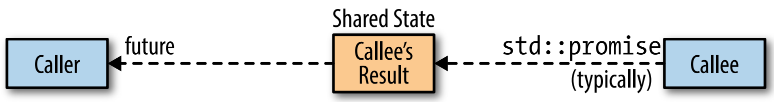
The result can't be stored in the caller's future, either, because (among other reasons) a `std::future` may be used to create a `std::shared_future` (thus transferring ownership of the callee's result from the `std::future` to the `std::shared_future`), which may then be copied many times after the original `std::future` is destroyed. Given that not all result types can be copied (i.e., move-only types) and that the result

4 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.

must live at least as long as the last future referring to it, which of the potentially many futures corresponding to the callee should be the one to contain its result?

Because neither objects associated with the callee nor objects associated with the caller are suitable places to store the callee's result, it's stored in a location outside both. This location is known as the *shared state*. The shared state is typically represented by a heap-based object, but its type, interface, and implementation are not specified by the Standard. Standard Library authors are free to implement shared states in any way they like.

We can envision the relationship among the callee, the caller, and the shared state as follows, where dashed arrows once again represent the flow of information:



The existence of the shared state is important, because the behavior of a future's destructor—the topic of this Item—is determined by the shared state associated 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 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 general 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 being destroyed, the future being destroyed follows the normal behavior (i.e., it simply destroys its data members).

Only when all of these conditions are fulfilled does a future's destructor exhibit special behavior, and that behavior is to block until the asynchronously running task completes. Practically speaking, this amounts to an implicit `join` with the thread running the `std::async`-created task.

It's common to hear this exception to normal future destructor behavior summarized as "Futures from `std::async` block in their destructors." To a first approximation, that's correct, but sometimes you need more than a first approximation. Now you know the truth in all its glory and wonder.

Your wonder may take a different form. It may be of the "I wonder why there's a special rule for shared states for non-deferred tasks that are launched by `std::async`" variety. It's a reasonable question. From what I can tell, the Standardization Committee wanted to avoid the problems associated with an implicit `detach` (see Item 37), but they didn't want to adopt as radical a policy as mandatory program termination (as they did for joinable `std::threads`—again, see Item 37), so they compromised on an implicit `join`. The decision was not without controversy, and there was serious talk about abandoning this behavior for C++14. In the end, no change was made, so the behavior of destructors for futures is consistent in C++11 and C++14.

The API for futures offers no way to determine whether a future refers to a shared state arising from a call to `std::async`, so given an arbitrary future object, it's not possible to know whether it will block in its destructor waiting for an asynchronously running task to finish. This has some interesting implications:

```
// this container might block in its dtor, because one or more
// contained futures could refer to a shared state for a non-
// deferred task launched via std::async
std::vector<std::future<void>> futs;   // see Item 39 for info
                                       // on std::future<void>

class Widget {                         // Widget objects might
public:                                // block in their dtors
```

```
  …

private:
  std::shared_future<double> fut;
};
```

Of course, if you have a way of knowing that a given future *does not* satisfy the conditions that trigger the special destructor behavior (e.g., due to program logic), you're assured that that future won't block in its destructor. For example, only shared states arising from calls to `std::async` qualify for the special behavior, but there are other ways that shared states get created. One is the use of `std::packaged_task`. A `std::packaged_task` object prepares a function (or other callable object) for asynchronous execution by wrapping it such that its result is put into a shared state. A future referring to that shared state can then be obtained via `std::packaged_task`'s `get_future` function:

```
int calcValue();                      // func to run

std::packaged_task<int()>             // wrap calcValue so it
  pt(calcValue);                      // can run asynchronously

auto fut = pt.get_future();           // get future for pt
```

At this point, we know that the future `fut` doesn't refer to a shared state created by a call to `std::async`, so its destructor will behave normally.

Once created, the `std::packaged_task` `pt` can be run on a thread. (It could be run via a call to `std::async`, too, but if you want to run a task using `std::async`, there's little reason to create a `std::packaged_task`, because `std::async` does everything `std::packaged_task` does before it schedules the task for execution.)

`std::packaged_task`s aren't copyable, so when `pt` is passed to the `std::thread` constructor, it must be cast to an rvalue (via `std::move`—see Item 23):

```
std::thread t(std::move(pt));         // run pt on t
```

This example lends some insight into the normal behavior for future destructors, but it's easier to see if the statements are put together inside a block:

```
{                                     // begin block

  std::packaged_task<int()>
    pt(calcValue);

  auto fut = pt.get_future();
```

```
    std::thread t(std::move(pt));

    …                                     // see below

  }                                       // end block
```

The most interesting code here is the "…" that follows creation of the `std::thread` object `t` and precedes the end of the block. What makes it interesting is what can happen to `t` inside the "…" region. There are three basic possibilities:

- **Nothing happens to `t`**. In this case, `t` will be joinable at the end of the scope. That will cause the program to be terminated (see Item 37).
- **A `join` is done on `t`**. In this case, there would be no need for `fut` to block in its destructor, because the `join` is already present in the calling code.
- **A `detach` is done on `t`**. In this case, there would be no need for `fut` to `detach` in its destructor, because the calling code already does that.

In other words, when you have a future corresponding to a shared state that arose due to a `std::packaged_task`, there's usually no need to adopt a special destruction policy, because the decision among termination, joining, or detaching will be made in the code that manipulates the `std::thread` on which the `std::packaged_task` is typically run.

Things to Remember

- Future destructors normally just destroy the future's data members.
- The final future referring to a shared state for a non-deferred task launched via `std::async` blocks until the task completes.

Item 39: Consider `void` futures for one-shot event communication.

Sometimes it's useful for a task to tell a second, asynchronously running task that a particular event has occurred, because the second task can't proceed until the event has taken place. Perhaps a data structure has been initialized, a stage of computation has been completed, or a significant sensor value has been detected. When that's the case, what's the best way for this kind of inter-thread communication to take place?

An obvious approach is to use a condition variable (*condvar*). If we call the task that detects the condition the *detecting task* and the task reacting to the condition the *reacting task*, the strategy is simple: the reacting task waits on a condition variable, and the detecting thread notifies that condvar when the event occurs. Given

```
std::condition_variable cv;            // condvar for event

std::mutex m;                          // mutex for use with cv
```

the code in the detecting task is as simple as simple can be:

```
…                                      // detect event

cv.notify_one();                       // tell reacting task
```

If there were multiple reacting tasks to be notified, it would be appropriate to replace `notify_one` with `notify_all`, but for now, we'll assume there's only one reacting task.

The code for the reacting task is a bit more complicated, because before calling `wait` on the condvar, it must lock a mutex through a `std::unique_lock` object. (Locking a mutex before waiting on a condition variable is typical for threading libraries. The need to lock the mutex through a `std::unique_lock` object is simply part of the C++11 API.) Here's the conceptual approach:

```
…                                     // prepare to react

{                                     // open critical section

  std::unique_lock<std::mutex> lk(m); // lock mutex

  cv.wait(lk);                        // wait for notify;
                                      // this isn't correct!

  …                                   // react to event
                                      // (m is locked)

}                                     // close crit. section;
                                      // unlock m via lk's dtor

…                                     // continue reacting
                                      // (m now unlocked)
```

The first issue with this approach is what's sometimes termed a *code smell*: even if 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 data, but it's entirely possible that the detecting and reacting tasks have no need 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 the detecting task never accesses the data structure after initializing it, and if the reacting task never accesses it before the detecting task indicates that it's ready, the two tasks will stay out of each other's way through program logic. There will be no need for a mutex. The fact that the condvar approach requires one leaves behind the unsettling aroma of suspect design.

Even if you look past that, there are two other problems you should definitely pay attention to:

- **If the detecting task notifies the condvar before the reacting task `wait`s, 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.
- **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 exceptionally easy, because it permits a lambda (or other function object) that tests for the waited-for condition to be passed to `wait`. That is, the `wait` call in the reacting task could be written like this:

  ```
  cv.wait(lk,
          []{ return whether the event has occurred; });
  ```

  Taking advantage of this capability requires that the reacting task be able to determine whether the condition it's waiting for is true. But in the scenario we've been considering, the condition it's waiting for is the occurrence of an event that the detecting thread is responsible for recognizing. The reacting thread may have no way of determining whether the event it's waiting for has taken place. That's why it's waiting on a condition variable!

There are many situations where having tasks communicate using a condvar is a good fit for the problem at hand, but this doesn't seem to be one of them.

For many developers, the next trick in their bag is a shared boolean flag. The flag is initially `false`. When the detecting thread recognizes the event it's looking for, it sets the flag:

```
std::atomic<bool> flag(false);      // shared flag; see
                                    // Item 40 for std::atomic

…                                   // detect event
```

```
flag = true;                          // tell reacting task
```

For its part, the reacting thread simply polls the flag. When it sees that the flag is set, it knows that the event it's been waiting for has occurred:

```
…                                     // prepare to react

while (!flag);                        // wait for event

…                                     // react to event
```

This approach suffers from none of the drawbacks of the condvar-based design. There's no need for a mutex, no problem if the detecting task sets the flag before the reacting task starts polling, and nothing akin to a spurious wakeup. Good, good, good.

Less good is the cost of polling in the reacting task. During the time the task is waiting for the flag to be set, the task is essentially blocked, yet it's still running. As such, it occupies a hardware thread that another task might be able to make use of, it incurs the cost of a context switch each time it starts or completes its time-slice, and it could keep a core running that might otherwise be shut down to save power. A truly blocked task would do none of these things. That's an advantage of the condvar-based approach, because a task in a `wait` call is truly blocked.

It's common to combine the condvar and flag-based designs. A flag indicates whether the event of interest has occurred, but access to the flag is synchronized by a mutex. Because the mutex prevents concurrent access to the flag, there is, as Item 40 explains, no need for the flag to be `std::atomic`; a simple `bool` will do. The detecting task would then look like this:

```
std::condition_variable cv;             // as before
std::mutex m;

bool flag(false);                       // not std::atomic

…                                       // detect event

{
  std::lock_guard<std::mutex> g(m);     // lock m via g's ctor

  flag = true;                          // tell reacting task
                                        // (part 1)

}                                       // unlock m via g's dtor
```

```
cv.notify_one();                        // tell reacting task
                                        // (part 2)
```

And here's the reacting task:

```
…                                       // prepare to react

{                                       // as before
  std::unique_lock<std::mutex> lk(m);   // as before

  cv.wait(lk, [] { return flag; });     // use lambda to avoid
                                        // spurious wakeups

  …                                     // react to event
                                        // (m is locked)
}

…                                       // continue reacting
                                        // (m now unlocked)
```

This approach avoids the problems we've discussed. It works regardless of whether the reacting task `wait`s 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.

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.

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., writes into the communications channel). Meanwhile, the reacting task `wait`s on its future. That `wait` blocks the reacting task until the `std::promise` has been set.

Now, both `std::promise` and futures (i.e., `std::future` and `std::shared_future`) are templates that require a type parameter. That parameter indicates the type of data to be transmitted through the communications channel. In our case, however, there's no data to be conveyed. The only thing of interest to the reacting task is that its future has been set. What we need for the `std::promise` and future templates is a type that indicates that no data is to be conveyed across the communications channel. That type is `void`. The detecting task will thus use a `std::promise<void>`, and the reacting task a `std::future<void>` or `std::shared_future<void>`. The detecting task will set its `std::promise<void>` when the event of interest occurs, and the reacting task will `wait` on its future. Even though the reacting task won't receive any data from the detecting task, the communications channel will permit the reacting task to know when the detecting task has "written" its `void` data by calling `set_value` on its `std::promise`.

So given

```
std::promise<void> p;                // promise for
                                     // communications channel
```

the detecting task's code is trivial,

```
…                                    // detect event

p.set_value();                       // tell reacting task
```

and the reacting task's code is equally simple:

```
…                                    // prepare to react

p.get_future().wait();               // wait on future
                                     // corresponding to p

…                                    // react to event
```

Like the approach using a flag, this design requires no mutex, works regardless of whether the detecting task sets its `std::promise` before the reacting task `wait`s, and is immune to spurious wakeups. (Only condition variables are susceptible to that problem.) Like the condvar-based approach, the reacting task is truly blocked after making the `wait` call, so it consumes no system resources while waiting. Perfect, right?

Not exactly. Sure, a future-based approach skirts those shoals, but there are other hazards to worry about. For example, Item 38 explains that between a `std::promise` and a future is a shared state, and shared states are typically dynamically allocated. You should therefore assume that this design incurs the cost of heap-based allocation and deallocation.

Perhaps more importantly, a `std::promise` may be set only once. The communications channel between a `std::promise` and a future is a *one-shot* mechanism: it can't be used repeatedly. This is a notable difference from the condvar- and flag-based designs, both of which can be used to communicate multiple times. (A condvar can be repeatedly notified, and a flag can always be cleared and set again.)

The one-shot restriction isn't as limiting as you might think. Suppose you'd like to create a system thread in a suspended state. That is, you'd like to get all the overhead associated with thread creation out of the way so that when you're ready to execute something on the thread, the normal thread-creation latency will be avoided. Or you might want to create a suspended thread so that you could configure it before letting it run. Such configuration might include things like setting its priority or core affinity. The C++ concurrency API offers no way to do those things, but `std::thread` objects offer the `native_handle` member function, the result of which is intended to give you access to the platform's underlying threading API (usually POSIX threads or Windows threads). The lower-level API often makes it possible to configure thread characteristics such as priority and affinity.

Assuming you want to suspend a thread only once (after creation, but before it's running its thread function), a design using a `void` future is a reasonable choice. Here's the essence of the technique:

```
std::promise<void> p;

void react();                        // func for reacting task

void detect()                        // func for detecting task
{
  std::thread t([]                   // create thread
                {
                  p.get_future().wait();     // suspend t until
                  react();                   // future is set
                });
  …                                  // here, t is suspended
                                     // prior to call to react

  p.set_value();                     // unsuspend t (and thus
                                     // call react)
```

```
  …                                    // do additional work

  t.join();                            // make t unjoinable
}                                      // (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:

```
void detect()
{
  ThreadRAII tr(                           // use RAII object
    std::thread([]
                {
                  p.get_future().wait();
                  react();
                }),
    ThreadRAII::DtorAction::join           // risky! (see below)
  );

  …                                        // thread inside tr
                                           // is suspended here

  p.set_value();                           // unsuspend thread
                                           // inside tr

  …

}
```

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.

There are ways to address this problem, but I'll leave them in the form of the hallowed exercise for the reader.[5] Here, I'd like to show how the original code (i.e., not using `ThreadRAII`) can be extended to suspend and then unsuspend not just one

5 A reasonable place to begin researching the matter is my 24 December 2013 blog post at *The View From Aristeia*, "ThreadRAII + Thread Suspension = Trouble?"

reacting task, but many. It's a simple generalization, because the key is to use `std::shared_future`s instead of a `std::future` in the `react` code. Once you know that the `std::future`'s `share` member function transfers ownership of its shared state to the `std::shared_future` object produced by `share`, the code nearly writes itself. The only subtlety is that each reacting thread needs its own copy of the `std::shared_future` that refers to the shared state, so the `std::shared_future` obtained from `share` is captured by value by the lambdas running on the reacting threads:

```
std::promise<void> p;                // as before

void detect()                        // now for multiple
{                                    // reacting tasks

  auto sf = p.get_future().share();  // sf's type is
                                     // std::shared_future<void>

  std::vector<std::thread> vt;              // container for
                                            // reacting threads

  for (int i = 0; i < threadsToRun; ++i) {
    vt.emplace_back([sf]{ sf.wait();        // wait on local
                          react(); });      // copy of sf; see
  }                                         // Item 42 for info
                                            // on emplace_back

  …                                  // detect hangs if
                                     // this "…" code throws!

  p.set_value();                     // unsuspend all threads

  …

  for (auto& t : vt) {               // make all threads
    t.join();                        // unjoinable; see Item 2
  }                                  // for info on "auto&"
}
```

The fact that a design using futures can achieve this effect is noteworthy, and that's why you should consider it for one-shot event communication.

Things to Remember

- For simple event communication, condvar-based designs require a superfluous mutex, impose constraints on the relative progress of detecting and reacting tasks, and require reacting tasks to verify that the event has taken place.
- Designs employing a flag avoid those problems, but are based on polling, not blocking.
- A condvar and flag can be used together, but the resulting communications mechanism is somewhat stilted.
- Using `std::promise`s and futures dodges these issues, but the approach uses heap memory for shared states, and it's limited to one-shot communication.

Item 40: Use `std::atomic` for concurrency, `volatile` for special memory.

Poor `volatile`. So misunderstood. It shouldn't even be in this chapter, because it has nothing to do with concurrent programming. But in other programming languages (e.g., Java and C#), it is useful for such programming, and even in C++, some compilers have imbued `volatile` with semantics that render it applicable to concurrent software (but only when compiled with those compilers). It's thus worthwhile to discuss `volatile` in a chapter on concurrency if for no other reason than to dispel the confusion surrounding it.

The C++ feature that programmers sometimes confuse `volatile` with—the feature that definitely does belong in this chapter—is the `std::atomic` template. Instantiations of this template (e.g., `std::atomic<int>`, `std::atomic<bool>`, `std::atomic<Widget*>`, etc.) offer operations that are guaranteed to be seen as atomic by other threads. Once a `std::atomic` object has been constructed, operations on it behave as if they were inside a mutex-protected critical section, but the operations are generally implemented using special machine instructions that are more efficient than would be the case if a mutex were employed.

Consider this code using `std::atomic`:

```
std::atomic<int> ai(0);     // initialize ai to 0

ai = 10;                    // atomically set ai to 10

std::cout << ai;            // atomically read ai's value

++ai;                       // atomically increment ai to 11
```

```
--ai;                        // atomically decrement ai to 10
```

During execution of these statements, other threads reading `ai` may see only values of 0, 10, or 11. No other values are possible (assuming, of course, that this is the only thread modifying `ai`).

Two aspects of this example are worth noting. First, in the "`std::cout << ai;`" statement, the fact that `ai` is a `std::atomic` guarantees only that the read of `ai` is atomic. There is no guarantee that the entire statement proceeds atomically. Between the time `ai`'s value is read and `operator<<` is invoked to write it to the standard output, another thread may have modified `ai`'s value. That has no effect on the behavior of the statement, because `operator<<` for `int`s uses a by-value parameter for the `int` to output (the outputted value will therefore be the one that was read from `ai`), but it's important to understand that what's atomic in that statement is nothing more than the read of `ai`.

The second noteworthy aspect of the example is the behavior of the last two statements—the increment and decrement of `ai`. These are each read-modify-write (RMW) operations, yet they execute atomically. This is one of the nicest characteristics of the `std::atomic` types: once a `std::atomic` object has been constructed, all member functions on it, including those comprising RMW operations, are guaranteed to be seen by other threads as atomic.

In contrast, the corresponding code using `volatile` guarantees virtually nothing in a multithreaded context:

```
volatile int vi(0);          // initialize vi to 0

vi = 10;                     // set vi to 10

std::cout << vi;             // read vi's value

++vi;                        // increment vi to 11

--vi;                        // decrement vi to 10
```

During execution of this code, if other threads are reading the value of `vi`, they may see anything, e.g, -12, 68, 4090727—anything! Such code would have undefined behavior, because these statements modify `vi`, so if other threads are reading `vi` at the same time, there are simultaneous readers and writers of memory that's neither `std::atomic` nor protected by a mutex, and that's the definition of a data race.

As a concrete example of how the behavior of `std::atomic`s and `volatile`s can differ in a multithreaded program, consider a simple counter of each type that's incremented by multiple threads. We'll initialize each to 0:

```
std::atomic<int> ac(0);    // "atomic counter"

volatile int vc(0);        // "volatile counter"
```

We'll then increment each counter one time in two simultaneously running threads:

```
/*-----  Thread 1  ----- */     /*-------  Thread 2  ------- */

         ++ac;                           ++ac;
         ++vc;                           ++vc;
```

When both threads have finished, `ac`'s value (i.e., the value of the `std::atomic`) must be 2, because each increment occurs as an indivisible operation. `vc`'s value, on the other hand, need not be 2, because its increments may not occur atomically. Each increment consists of reading `vc`'s value, incrementing the value that was read, and writing the result back into `vc`. But these three operations are not guaranteed to proceed atomically for `volatile` objects, so it's possible that the component parts of the two increments of `vc` are interleaved as follows:

1. Thread 1 reads `vc`'s value, which is 0.
2. Thread 2 reads `vc`'s value, which is still 0.
3. Thread 1 increments the 0 it read to 1, then writes that value into `vc`.
4. Thread 2 increments the 0 it read to 1, then writes that value into `vc`.

`vc`'s final value is therefore 1, even though it was incremented twice.

This is not the only possible outcome. `vc`'s final value is, in general, not predictable, because `vc` is involved in a data race, and the Standard's decree that data races cause undefined behavior means that compilers may generate code to do literally anything. Compilers don't use this leeway to be malicious, of course. Rather, they perform optimizations that would be valid in programs without data races, and these optimizations yield unexpected and unpredictable behavior in programs where races are present.

The use of RMW operations isn't the only situation where `std::atomic`s comprise a concurrency success story and `volatile`s suffer failure. Suppose one task computes an important value needed by a second task. When the first task has computed the value, it must communicate this to the second task. Item 39 explains that one way for the first task to communicate the availability of the desired value to the second task is by using a `std::atomic<bool>`. Code in the task computing the value would look something like this:

```
std::atomic<bool> valAvailable(false);

auto imptValue = computeImportantValue();  // compute value

valAvailable = true;                       // tell other task
                                           // it's available
```

As humans reading this code, we know it's crucial that the assignment to `imptValue` take place before the assignment to `valAvailable`, but all compilers see is a pair of assignments to independent variables. As a general rule, compilers are permitted to reorder such unrelated assignments. That is, given this sequence of assignments (where `a`, `b`, `x`, and `y` correspond to independent variables),

```
a = b;
x = y;
```

compilers may generally reorder them as follows:

```
x = y;
a = b;
```

Even if compilers don't reorder them, the underlying hardware might do it (or might make it seem to other cores as if it had), because that can sometimes make the code run faster.

However, the use of `std::atomic`s imposes restrictions on how code can be reordered, and one such restriction is that no code that, in the source code, precedes a write of a `std::atomic` variable may take place (or appear to other cores to take place) afterwards.[6] That means that in our code,

```
auto imptValue = computeImportantValue();  // compute value

valAvailable = true;                       // tell other task
                                           // it's available
```

not only must compilers retain the order of the assignments to `imptValue` and `valAvailable`, they must generate code that ensures that the underlying hardware

6 This is true only for `std::atomic`s 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, to understand, and to 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.

does, too. As a result, declaring `valAvailable` as `std::atomic` ensures that our critical ordering requirement—`imptValue` must be seen by all threads to change no later than `valAvailable` does—is maintained.

Declaring `valAvailable` as `volatile` doesn't impose the same code reordering restrictions:

```
volatile bool valAvailable(false);

auto imptValue = computeImportantValue();

valAvailable = true;  // other threads might see this assignment
                      // before the one to imptValue!
```

Here, compilers might flip the order of the assignments to `imptValue` and `valAvailable`, and even if they don't, they might fail to generate machine code that would prevent the underlying hardware from making it possible for code on other cores to see `valAvailable` change before `imptValue`.

These two issues—no guarantee of operation atomicity and insufficient restrictions on code reordering—explain why `volatile`'s not useful for concurrent programming, but it doesn't explain what it is useful for. In a nutshell, it's for telling compilers that they're dealing with memory that doesn't behave normally.

"Normal" memory has the characteristic that if you write a value to a memory location, the value remains there until something overwrites it. So if I have a normal `int`,

```
int x;
```

and a compiler sees the following sequence of operations on it,

```
auto y = x;             // read x
y = x;                  // read x again
```

the compiler can optimize the generated code by eliminating the assignment to `y`, because it's redundant with `y`'s initialization.

Normal memory also has the characteristic that if you write a value to a memory location, never read it, and then write to that memory location again, the first write can be eliminated, because it was never used. So given these two adjacent statements,

```
x = 10;                 // write x
x = 20;                 // write x again
```

compilers can eliminate the first one. That means that if we have this in the source code,

```
auto y = x;             // read x
y = x;                  // read x again
```

```
x = 10;                  // write x
x = 20;                  // write x again
```

compilers can treat it as if it had been written like this:

```
auto y = x;              // read x

x = 20;                  // write x
```

Lest you wonder who'd write code that performs these kinds of redundant reads and superfluous writes (technically known as *redundant loads* and *dead stores*), the answer is that humans don't write it directly—at least we hope they don't. However, after compilers take reasonable-looking source code and perform template instantiation, inlining, and various common kinds of reordering optimizations, it's not uncommon for the result to have redundant loads and dead stores that compilers can get rid of.

Such optimizations are valid only if memory behaves normally. "Special" memory doesn't. Probably the most common kind of special memory is memory used for *memory-mapped I/O*. Locations in such memory actually communicate with peripherals, e.g., external sensors or displays, printers, network ports, etc. rather than reading or writing normal memory (i.e., RAM). In such a context, consider again the code with seemingly redundant reads:

```
auto y = x;              // read x
y = x;                   // read x again
```

If `x` corresponds to, say, the value reported by a temperature sensor, the second read of `x` is not redundant, because the temperature may have changed between the first and second reads.

It's a similar situation for seemingly superfluous writes. In this code, for example,

```
x = 10;                  // write x
x = 20;                  // write x again
```

if `x` corresponds to the control port for a radio transmitter, it could be that the code is issuing commands to the radio, and the value 10 corresponds to a different command from the value 20. Optimizing out the first assignment would change the sequence of commands sent to the radio.

`volatile` is the way we tell compilers that we're dealing with special memory. Its meaning to compilers is "Don't perform any optimizations on operations on this memory." So if `x` corresponds to special memory, it'd be declared `volatile`:

```
volatile int x;
```

Consider the effect that has on our original code sequence:

```
auto y = x;            // read x
y = x;                 // read x again (can't be optimized away)

x = 10;                // write x (can't be optimized away)
x = 20;                // write x again
```

This is precisely what we want if `x` is memory-mapped (or has been mapped to a memory location shared across processes, etc.).

Pop quiz! In that last piece of code, what is `y`'s type: `int` or `volatile int`?[7]

The fact that seemingly redundant loads and dead stores must be preserved when dealing with special memory explains, by the way, why `std::atomic`s are unsuitable for this kind of work. Compilers are permitted to eliminate such redundant operations on `std::atomic`s. The code isn't written quite the same way it is for `volatile`s, but if we overlook that for a moment and focus on what compilers are permitted to do, we can say that, conceptually, compilers may take this,

```
std::atomic<int> x;

auto y = x;            // conceptually read x (see below)
y = x;                 // conceptually read x again (see below)

x = 10;                // write x
x = 20;                // write x again
```

and optimize it to this:

```
auto y = x;            // conceptually read x (see below)
x = 20;                // write x
```

For special memory, this is clearly unacceptable behavior.

Now, as it happens, neither of these two statements will compile when `x` is `std::atomic`:

```
auto y = x;            // error!
y = x;                 // error!
```

That's because the copy operations for `std::atomic` are deleted (see Item 11). And with good reason. Consider what would happen if the initialization of `y` with `x` com-

7 `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.

piled. Because `x` is `std::atomic`, `y`'s type would be deduced to be `std::atomic`, too (see Item 2). I remarked earlier that one of the best things about `std::atomic`s is that all their operations are atomic, but in order for the copy construction of `y` from `x` to be atomic, compilers would have to generate code to read `x` and write `y` in a single atomic operation. Hardware generally can't do that, so copy construction isn't supported for `std::atomic` types. Copy assignment is deleted for the same reason, which is why the assignment from `x` to `y` won't compile. (The move operations aren't explicitly declared in `std::atomic`, so, per the rules for compiler-generated special functions described in Item 17, `std::atomic` offers neither move construction nor move assignment.)

It's possible to get the value of `x` into `y`, but it requires use of `std::atomic`'s member functions `load` and `store`. The `load` member function reads a `std::atomic`'s value atomically, while the `store` member function writes it atomically. To initialize `y` with `x`, followed by putting `x`'s value in `y`, the code must be written like this:

```
std::atomic<int> y(x.load());      // read x

y.store(x.load());                 // read x again
```

This compiles, but the fact that reading `x` (via `x.load()`) is a separate function call from initializing or storing to `y` makes clear that there is no reason to expect either statement as a whole to execute as a single atomic operation.

Given that code, compilers could "optimize" it by storing `x`'s value in a register instead of reading it twice:

```
register = x.load();               // read x into register

std::atomic<int> y(register);      // init y with register value

y.store(register);                 // store register value into y
```

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.)

The situation should thus be clear:

- `std::atomic` is useful for concurrent programming, but not for accessing special memory.
- `volatile` is useful for accessing special memory, but not for concurrent programming.

Because std::atomic and volatile serve different purposes, they can even be used together:

```
volatile std::atomic<int> vai;    // operations on vai are
                                  // atomic and can't be
                                  // optimized away
```

This could be useful if vai corresponded to a memory-mapped I/O location that was concurrently accessed by multiple threads.

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 threads (e.g., a flag indicating the availability of data) could mean that the variable wasn't declared std::atomic when it should have been.

This is largely a style issue, however, and as such is quite different from the choice between std::atomic and volatile.

Things to Remember

- std::atomic is for data accessed from multiple threads without using mutexes. It's a tool for writing concurrent software.
- volatile is for memory where reads and writes should not be optimized away. It's a tool for working with special memory.

CHAPTER 8
Tweaks

For every general technique or feature in C++, there are circumstances where it's reasonable to use it, and there are circumstances where it's not. Describing when it makes sense to use a general technique or feature is usually fairly straightforward, but this chapter covers two exceptions. The general technique is pass by value, and the general feature is emplacement. The decision about when to employ them is affected by so many factors, the best advice I can offer is to *consider* their use. Nevertheless, both are important players in effective modern C++ programming, and the Items that follow provide the information you'll need to determine whether using them is appropriate for your software.

Item 41: Consider pass by value for copyable parameters that are cheap to move and always copied.

Some function parameters are intended to be copied.[1] For example, a member function `addName` might copy its parameter into a private container. For efficiency, such a function should copy lvalue arguments, but move rvalue arguments:

```
class Widget {
public:
  void addName(const std::string& newName)    // take lvalue;
  { names.push_back(newName); }               // copy it

  void addName(std::string&& newName)         // take rvalue;
```

1 In this Item, to "copy" a parameter generally means to use it as the source of a copy or move operation. Recall on page 2 that C++ has no terminology to distinguish a copy made by a copy operation from one made by a move operation.

```
  { names.push_back(std::move(newName)); }    // move it; see
  …                                            // Item 25 for use
                                               // of std::move
private:
  std::vector<std::string> names;
};
```

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

Furthermore, there will be two functions in the object code—something you might care about if you're concerned about your program's footprint. In this case, both functions will probably be inlined, and that's likely to eliminate any bloat issues related to the existence of two functions, but if these functions aren't inlined everywhere, you really will get two functions in your object code.

An alternative approach is to make `addName` a function template taking a universal reference (see Item 24):

```
class Widget {
public:
  template<typename T>                            // take lvalues
  void addName(T&& newName)                       // and rvalues;
  {                                               // copy lvalues,
    names.push_back(std::forward<T>(newName));    // move rvalues;
  }                                               // see Item 25
                                                  // for use of
  …                                               // std::forward

};
```

This reduces the source code you have to deal with, but the use of universal references leads to other complications. As a template, `addName`'s implementation must typically be in a header file. It may yield several functions in object code, because it not only instantiates differently for lvalues and rvalues, it also instantiates differently for `std::string` and types that are convertible to `std::string` (see Item 25). At the same time, there are argument types that can't be passed by universal reference (see Item 30), and if clients pass improper argument types, compiler error messages can be intimidating (see Item 27).

Wouldn't it be nice if there were a way to write functions like `addName` such that lvalues were copied, rvalues were moved, there was only one function to deal with (in both source and object code), and the idiosyncrasies of universal references were avoided? As it happens, there is. All you have to do is abandon one of the first rules you probably learned as a C++ programmer. That rule was to avoid passing objects of user-defined types by value. For parameters like `newName` in functions like `addName`, pass by value may be an entirely reasonable strategy.

Before we discuss why pass-by-value may be a good fit for `newName` and `addName`, let's see how it would be implemented:

```
class Widget {
public:
  void addName(std::string newName)          // take lvalue or
  { names.push_back(std::move(newName)); }    // rvalue; move it

  …

};
```

The only non-obvious part of this code is the application of `std::move` to the parameter `newName`. Typically, `std::move` is used with rvalue references, but in this case, we know that (1) `newName` is a completely independent object from whatever the caller passed in, so changing `newName` won't affect callers and (2) this is the final use of `newName`, so moving from it won't have any impact on the rest of the function.

The fact that there's only one `addName` function explains how we avoid code duplication, both in the source code and the object code. We're not using a universal reference, so this approach doesn't lead to bloated header files, odd failure cases, or confounding error messages. But what about the efficiency of this design? We're passing *by value*. Isn't that expensive?

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

```
Widget w;

…

std::string name("Bart");

w.addName(name);                  // call addName with lvalue

…

w.addName(name + "Jenne");        // call addName with rvalue
                                  // (see below)
```

In the first call to addName (when name is passed), the parameter newName is initialized with an lvalue. newName is thus copy constructed, just like it would be in C++98. In the second call, newName is initialized with the std::string object resulting from a call to operator+ for std::string (i.e., the append operation). That object is an rvalue, and newName is therefore move constructed.

Lvalues are thus copied, and rvalues are moved, just like we want. Neat, huh?

It is neat, but there are some caveats you need to keep in mind. Doing that will be easier if we recap the three versions of addName we've considered:

```
class Widget {                                  // Approach 1:
public:                                         // overload for
  void addName(const std::string& newName)      // lvalues and
  { names.push_back(newName); }                 // rvalues

  void addName(std::string&& newName)
  { names.push_back(std::move(newName)); }
  …

private:
  std::vector<std::string> names;
};

class Widget {                                  // Approach 2:
public:                                         // use universal
  template<typename T>                          // reference
  void addName(T&& newName)
  { names.push_back(std::forward<T>(newName)); }
  …
};

class Widget {                                  // Approach 3:
public:                                         // pass by value
  void addName(std::string newName)
  { names.push_back(std::move(newName)); }
  …
};
```

I refer to the first two versions as the "by-reference approaches," because they're both based on passing their parameters by reference.

Here are the two calling scenarios we've examined:

```
Widget w;
…
std::string name("Bart");

w.addName(name);                      // pass lvalue
…
w.addName(name + "Jenne");            // pass rvalue
```

Now consider the cost, in terms of copy and move operations, of adding a name to a `Widget` for the two calling scenarios and each of the three `addName` implementations we've discussed. The accounting will largely ignore the possibility of compilers optimizing copy and move operations away, because such optimizations are context- and compiler-dependent and, in practice, don't change the essence of the analysis.

- **Overloading:** Regardless of whether an lvalue or an rvalue is passed, the caller's argument is bound to a reference called `newName`. That costs nothing, in terms of copy and move operations. In the lvalue overload, `newName` is copied into `Widget::names`. In the rvalue overload, it's moved. Cost summary: one copy for lvalues, one move for rvalues.
- **Using a universal reference:** As with overloading, the caller's argument is bound to the reference `newName`. This is a no-cost operation. Due to the use of `std::forward`, lvalue `std::string` arguments are copied into `Widget::names`, while rvalue `std::string` arguments are moved. The cost summary for `std::string` arguments is the same as with overloading: one copy for lvalues, one move for rvalues.

  Item 25 explains that if a caller passes an argument of a type other than `std::string`, it will be forwarded to a `std::string` constructor, and that could cause as few as zero `std::string` copy or move operations to be performed. Functions taking universal references can thus be uniquely efficient. However, that doesn't affect the analysis in this Item, so we'll keep things simple by assuming that callers always pass `std::string` arguments.
- **Passing by value:** Regardless of whether an lvalue or an rvalue is passed, the parameter `newName` must be constructed. If an lvalue is passed, this costs a copy construction. If an rvalue is passed, it costs a move construction. In the body of the function, `newName` is unconditionally moved into `Widget::names`. The cost summary is thus one copy plus one move for lvalues, and two moves for rvalues. Compared to the by-reference approaches, that's one extra move for both lvalues and rvalues.

Look again at this Item's title:

> Consider pass by value for copyable parameters that are cheap to move and always copied.

It's worded the way it is for a reason. Four reasons, in fact:

1. You should only *consider* using pass by value. Yes, it requires writing only one function. Yes, it generates only one function in the object code. Yes, it avoids the issues associated with universal references. But it has a higher cost than the alternatives, and, as we'll see below, in some cases, there are expenses we haven't yet discussed.

2. Consider pass by value only for *copyable parameters*. Parameters failing this test must have move-only types, because if they're not copyable, yet the function always makes a copy, the copy must be created via the move constructor.[2] Recall that the advantage of pass by value over overloading is that with pass by value, only one function has to be written. But for move-only types, there is no need to provide an overload for lvalue arguments, because copying an lvalue entails calling the copy constructor, and the copy constructor for move-only types is disabled. That means that only rvalue arguments need to be supported, and in that case, the "overloading" solution requires only one overload: the one taking an rvalue reference.

   Consider a class with a `std::unique_ptr<std::string>` data member and a setter for it. `std::unique_ptr` is a move-only type, so the "overloading" approach to its setter consists of a single function:

   ```
   class Widget {
   public:
     …
     void setPtr(std::unique_ptr<std::string>&& ptr)
     { p = std::move(ptr); }

   private:
     std::unique_ptr<std::string> p;
   };
   ```

   A caller might use it this way:

2 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.

```
Widget w;
…

w.setPtr(std::make_unique<std::string>("Modern C++"));
```

Here the rvalue `std::unique_ptr<std::string>` returned from `std::make_unique` (see Item 21) is passed by rvalue reference to `setPtr`, where it's moved into the data member `p`. The total cost is one move.

If `setPtr` were to take its parameter by value,

```
class Widget {
public:
  …

  void setPtr(std::unique_ptr<std::string> ptr)
  { p = std::move(ptr); }

  …
};
```

the same call would move construct the parameter `ptr`, and `ptr` would then be move assigned into the data member `p`. The total cost would thus be two moves —twice that of the "overloading" approach.

3. Pass by value is worth considering only for parameters that are *cheap to move*. When moves are cheap, the cost of an extra one may be acceptable, but when they're not, performing an unnecessary move is analogous to performing an unnecessary copy, and the importance of avoiding unnecessary copy operations is what led to the C++98 rule about avoiding pass by value in the first place!

4. You should consider pass by value only for parameters that are *always copied*. To see why this is important, suppose that before copying its parameter into the `names` container, `addName` checks to see if the new name is too short or too long. If it is, the request to add the name is ignored. A pass-by-value implementation could be written like this:

```
class Widget {
public:
  void addName(std::string newName)
  {
    if ((newName.length() >= minLen) &&
        (newName.length() <= maxLen))
      {
        names.push_back(std::move(newName));
      }
  }
```

```
    …

  private:
    std::vector<std::string> names;
  };
```

> This function incurs the cost of constructing and destroying `newName`, even if nothing is added to `names`. That's a price the by-reference approaches wouldn't be asked to pay.

Even when you're dealing with a function performing an unconditional copy on a copyable type that's cheap to move, there are times when pass by value may not be appropriate. That's because a function can copy a parameter in two ways: via *construction* (i.e., copy construction or move construction) and via *assignment* (i.e., copy assignment or move assignment). `addName` uses construction: its parameter `newName` is passed to `vector::push_back`, and inside that function, `newName` is copy constructed into a new element created at the end of the `std::vector`. For functions that use construction to copy their parameter, the analysis we saw earlier is complete: using pass by value incurs the cost of an extra move for both lvalue and rvalue arguments.

When a parameter is copied using assignment, the situation is more complicated. Suppose, for example, we have a class representing passwords. Because passwords can be changed, we provide a setter function, `changeTo`. Using a pass-by-value strategy, we could implement `Password` like this:

```
class Password {
public:
  explicit Password(std::string pwd)      // pass by value
  : text(std::move(pwd)) {}               // construct text

  void changeTo(std::string newPwd)       // pass by value
  { text = std::move(newPwd); }           // assign text

  …

private:
  std::string text;                       // text of password
};
```

Storing the password as plain text will whip your software security SWAT team into a frenzy, but ignore that and consider this code:

```
std::string initPwd("Supercalifragilisticexpialidocious");

Password p(initPwd);
```

There are no suprises here: `p.text` is constructed with the given password, and using pass by value in the constructor incurs the cost of a `std::string` move construction that would not be necessary if overloading or perfect forwarding were employed. All is well.

A user of this program may not be as sanguine about the password, however, because "Supercalifragilisticexpialidocious" is found in many dictionaries. He or she may therefore take actions that lead to code equivalent to the following being executed:

```
std::string newPassword = "Beware the Jabberwock";

p.changeTo(newPassword);
```

Whether the new password is better than the old one is debatable, but that's the user's problem. Ours is that `changeTo`'s use of assignment to copy the parameter `newPwd` probably causes that function's pass-by-value strategy to explode in cost.

The argument passed to `changeTo` is an lvalue (`newPassword`), so when the parameter `newPwd` is constructed, it's the `std::string` copy constructor that's called. That constructor allocates memory to hold the new password. `newPwd` is then move-assigned to `text`, which causes the memory already held by `text` to be deallocated. There are thus two dynamic memory management actions within `changeTo`: one to allocate memory for the new password, and one to deallocate the memory for the old password.

But in this case, the old password ("Supercalifragilisticexpialidocious") is longer than the new one ("Beware the Jabberwock"), so there's no need to allocate or deallocate anything. If the overloading approach were used, it's likely that none would take place:

```
class Password {
public:
  …

  void changeTo(const std::string& newPwd)      // the overload
  {                                             // for lvalues

    text = newPwd;           // can reuse text's memory if
                             // text.capacity() >= newPwd.size()
  }

  …
```

```
private:
  std::string text;                          // as above
};
```

In this scenario, the cost of pass by value includes an extra memory allocation and deallocation—costs that are likely to exceed that of a `std::string` move operation by orders of magnitude.

Interestingly, if the old password were shorter than the new one, it would typically be impossible to avoid an allocation-deallocation pair during the assignment, and in that case, pass by value would run at about the same speed as pass by reference. The cost of assignment-based parameter copying can thus depend on the values of the objects participating in the assignment! This kind of analysis applies to any parameter type that holds values in dynamically allocated memory. Not all types qualify, but many—including `std::string` and `std::vector`—do.

This potential cost increase generally applies only when lvalue arguments are passed, because the need to perform memory allocation and deallocation typically occurs only when true copy operations (i.e., not moves) are performed. For rvalue arguments, moves almost always suffice.

The upshot is that the extra cost of pass by value for functions that copy a parameter using assignment depends on the type being passed, the ratio of lvalue to rvalue arguments, whether the type uses dynamically allocated memory, and, if so, the implementation of that type's assignment operators and the likelihood that the memory associated with the assignment target is at least as large as the memory associated with the assignment source. For `std::string`, it also depends on whether the implementation uses the small string optimization (SSO—see Item 29) and, if so, whether the values being assigned fit in the SSO buffer.

So, as I said, when parameters are copied via assignment, analyzing the cost of pass by value is complicated. Usually, the most practical approach is to adopt a "guilty until proven innocent" policy, whereby you use overloading or universal references instead of pass by value unless it's been demonstrated that pass by value yields acceptably efficient code for the parameter type you need.

Now, for software that must be as fast as possible, pass by value may not be a viable strategy, because avoiding even cheap moves can be important. Moreover, it's not always clear how many moves will take place. In the `Widget::addName` example, pass by value incurs only a single extra move operation, but suppose that `Widget::addName` called `Widget::validateName`, and this function also passed by value. (Presumably it has a reason for always copying its parameter, e.g., to store it in a data structure of all values it validates.) And suppose that `validateName` called a third function that also passed by value…

You can see where this is headed. When there are chains of function calls, each of which employs pass by value because "it costs only one inexpensive move," the cost for the entire chain of calls may not be something you can tolerate. Using by-reference parameter passing, chains of calls don't incur this kind of accumulated overhead.

An issue unrelated to performance, but still worth keeping in mind, is that pass by value, unlike pass by reference, is susceptible to *the slicing problem*. This is well-trod C++98 ground, so I won't dwell on it, but if you have a function that is designed to accept a parameter of a base class type *or any type derived from it*, you don't want to declare a pass-by-value parameter of that type, because you'll "slice off" the derived-class characteristics of any derived type object that may be passed in:

```
class Widget { … };                          // base class

class SpecialWidget: public Widget { … };    // derived class

void processWidget(Widget w);   // func for any kind of Widget,
                                // including derived types;
…                               // suffers from slicing problem

SpecialWidget sw;

…

processWidget(sw);              // processWidget sees a
                                // Widget, not a SpecialWidget!
```

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

C++11 doesn't fundamentally change the C++98 wisdom regarding pass by value. In general, pass by value still entails a performance hit you'd prefer to avoid, and pass by value can still lead to the slicing problem. What's new in C++11 is the distinction between lvalue and rvalue arguments. Implementing functions that take advantage of move semantics for rvalues of copyable types requires either overloading or using universal references, both of which have drawbacks. For the special case of copyable, cheap-to-move types passed to functions that always copy them and where slicing is not a concern, pass by value can offer an easy-to-implement alternative that's nearly as efficient as its pass-by-reference competitors, but avoids their disadvantages.

Things to Remember

- For copyable, cheap-to-move parameters that are always copied, pass by value may be nearly as efficient as pass by reference, it's easier to implement, and it can generate less object code.
- Copying parameters via construction may be significantly more expensive than copying them via assignment.
- Pass by value is subject to the slicing problem, so it's typically inappropriate for base class parameter types.

Item 42: Consider emplacement instead of insertion.

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

Logical though this may be, it's not always true. Consider this code:

```
std::vector<std::string> vs;         // container of std::string

vs.push_back("xyzzy");               // add string literal
```

Here, the container holds `std::string`s, but what you have in hand—what you're actually trying to `push_back`—is a string literal, i.e., a sequence of characters inside quotes. A string literal is not a `std::string`, and that means that the argument you're passing to `push_back` is not of the type held by the container.

`push_back` for `std::vector` is overloaded for lvalues and rvalues as follows:

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

In the call

```
vs.push_back("xyzzy");
```

compilers see a mismatch between the type of the argument (`const char[6]`) and the type of the parameter taken by `push_back` (a reference to a `std::string`). They address the mismatch by generating code to create a temporary `std::string` object from the string literal, and they pass that temporary object to `push_back`. In other words, they treat the call as if it had been written like this:

```
vs.push_back(std::string("xyzzy"));  // create temp. std::string
                                     // and pass it to push_back
```

The code compiles and runs, and everybody goes home happy. Everybody except the performance freaks, that is, because the performance freaks recognize that this code isn't as efficient as it should be.

To create a new element in a container of `std::string`s, they understand, a `std::string` constructor is going to have to be called, but the code above doesn't make just one constructor call. It makes two. And it calls the `std::string` destructor, too. Here's what happens at runtime in the call to `push_back`:

1. A temporary `std::string` object is created from the string literal `"xyzzy"`. This object has no name; we'll call it *`temp`*. Construction of *`temp`* is the first `std::string` construction. Because it's a temporary object, *`temp`* is an rvalue.
2. *`temp`* is passed to the rvalue overload for `push_back`, where it's bound to the rvalue reference parameter `x`. A copy of `x` is then constructed in the memory for the `std::vector`. This construction—the *second* one—is what actually creates a new object inside the `std::vector`. (The constructor that's used to copy `x` into the `std::vector` is the move constructor, because `x`, being an rvalue reference, gets cast to an rvalue before it's copied. For information about the casting of rvalue reference parameters to rvalues, see Item 25.)
3. Immediately after `push_back` returns, *`temp`* is destroyed, thus calling the `std::string` destructor.

The performance freaks can't help but notice that if there were a way to take the string literal and pass it directly to the code in step 2 that constructs the `std::string` object inside the `std::vector`, we could avoid constructing and destroying *`temp`*. That would be maximally efficient, and even the performance freaks could contentedly decamp.

Because you're a C++ programmer, there's an above-average chance you're a performance freak. If you're not, you're still probably sympathetic to their point of view. (If you're not at all interested in performance, shouldn't you be in the Python room down the hall?) So I'm pleased to tell you that there is a way to do exactly what is needed for maximal efficiency in the call to push_back. It's to not call push_back. push_back is the wrong function. The function you want is emplace_back.

emplace_back does exactly what we desire: it uses whatever arguments are passed to it to construct a std::string directly inside the std::vector. No temporaries are involved:

```
vs.emplace_back("xyzzy");   // construct std::string inside
                            // vs directly from "xyzzy"
```

emplace_back uses perfect forwarding, so, as long as you don't bump into one of perfect forwarding's limitations (see Item 30), you can pass any number of arguments of any combination of types through emplace_back. For example, if you'd like to create a std::string in vs via the std::string constructor taking a character and a repeat count, this would do it:

```
vs.emplace_back(50, 'x');   // insert std::string consisting
                            // of 50 'x' characters
```

emplace_back is available for every standard container that supports push_back. Similarly, every standard container that supports push_front supports emplace_front. And every standard container that supports insert (which is all but std::forward_list and std::array) supports emplace. The associative containers offer emplace_hint to complement their insert functions that take a "hint" iterator, and std::forward_list has emplace_after to match 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

```
std::string queenOfDisco("Donna Summer");
```

both of the following calls are valid, and both have the same net effect on the container:

```
vs.push_back(queenOfDisco);       // copy-construct queenOfDisco
                                  // at end of vs

vs.emplace_back(queenOfDisco);    // ditto
```

Emplacement functions can thus do everything insertion functions can. They sometimes do it more efficiently, and, at least in theory, they should never do it less efficiently. So why not use them all the time?

Because, as the saying goes, in theory, there's no difference between theory and practice, but in practice, there is. With current implementations of the Standard Library, there are situations where, as expected, emplacement outperforms insertion, but, sadly, there are also situations where the insertion functions run faster. Such situations are not easy to characterize, because they depend on the types of arguments being passed, the containers being used, the locations in the containers where insertion or emplacement is requested, the exception safety of the contained types' constructors, and, for containers where duplicate values are prohibited (i.e., `std::set`, `std::map`, `std::unordered_set`, `std::unordered_map`), whether the value to be added is already in the container. The usual performance-tuning advice thus applies: to determine whether emplacement or insertion runs faster, benchmark them both.

That's not very satisfying, of course, so you'll be pleased to learn that there's a heuristic that can help you identify situations where emplacement functions are most likely to be worthwhile. If all the following are true, emplacement will almost certainly outperform insertion:

- **The value being added is constructed into the container, not assigned.** The example that opened this Item (adding a `std::string` with the value `"xyzzy"` to a `std::vector vs`) showed the value being added to the end of `vs`—to a place where no object yet existed. The new value therefore had to be constructed into the `std::vector`. If we revise the example such that the new `std::string` goes into a location already occupied by an object, it's a different story. Consider:

  ```
  std::vector<std::string> vs;         // as before

  …                                    // add elements to vs

  vs.emplace(vs.begin(), "xyzzy");     // add "xyzzy" to
                                       // beginning of vs
  ```

  For this code, few implementations will construct the added `std::string` into the memory occupied by `vs[0]`. Instead, they'll move-assign the value into place. But move assignment requires an object to move from, and that means that a temporary object will need to be created to be the source of the move. Because the primary advantage of emplacement over insertion is that temporary objects are neither created nor destroyed, when the value being added is put into the container via assignment, emplacement's edge tends to disappear.

  Alas, whether adding a value to a container is accomplished by construction or assignment is generally up to the implementer. But, again, heuristics can help.

Node-based containers virtually always use construction to add new values, and most standard containers are node-based. The only ones that aren't are `std::vector`, `std::deque`, and `std::string`. (`std::array` isn't, either, but it doesn't support insertion or emplacement, so it's not relevant here.) Within the non-node-based containers, you can rely on `emplace_back` to use construction instead of assignment to get a new value into place, and for `std::deque`, the same is true of `emplace_front`.

- **The argument type(s) being passed differ from the type held by the container.** Again, emplacement's advantage over insertion generally stems from the fact that its interface doesn't require creation and destruction of a temporary object when the argument(s) passed are of a type other than that held by the container. When an object of type `T` is to be added to a *`container`*`<T>`, there's no reason to expect emplacement to run faster than insertion, because no temporary needs to be created to satisfy the insertion interface.
- **The container is unlikely to reject the new value as a duplicate.** This means that the container either permits duplicates or that most of the values you add will be unique. The reason this matters is that in order to detect whether a value is already in the container, emplacement implementations typically create a node with the new value so that they can compare the value of this node with existing container nodes. If the value to be added isn't in the container, the node is linked in. However, if the value is already present, the emplacement is aborted and the node is destroyed, meaning that the cost of its construction and destruction was wasted. Such nodes are created for emplacement functions more often than for insertion functions.

The following calls from earlier in this Item satisfy all the criteria above. They also run faster than the corresponding calls to `push_back`.

```
vs.emplace_back("xyzzy");   // construct new value at end of
                            // container; don't pass the type in
                            // container; don't use container
                            // rejecting duplicates

vs.emplace_back(50, 'x');   // ditto
```

When deciding whether to use emplacement functions, two other issues are worth keeping in mind. The first regards resource management. Suppose you have a container of `std::shared_ptr<Widget>`s,

```
std::list<std::shared_ptr<Widget>> ptrs;
```

and you want to add a `std::shared_ptr` that should be released via a custom deleter (see Item 19). Item 21 explains that you should use `std::make_shared` to create `std::shared_ptr`s whenever you can, but it also concedes that there are situations where you can't. One such situation is when you want to specify a custom deleter. In that case, you must use `new` directly to get the raw pointer to be managed by the `std::shared_ptr`.

If the custom deleter is this function,

```
void killWidget(Widget* pWidget);
```

the code using an insertion function could look like this:

```
ptrs.push_back(std::shared_ptr<Widget>(new Widget, killWidget));
```

It could also look like this, though the meaning would be the same:

```
ptrs.push_back({ new Widget, killWidget });
```

Either way, a temporary `std::shared_ptr` would be constructed before calling `push_back`. `push_back`'s parameter is a reference to a `std::shared_ptr`, so there has to be a `std::shared_ptr` for this parameter to refer to.

The creation of the temporary `std::shared_ptr` is what `emplace_back` would avoid, but in this case, that temporary is worth far more than it costs. Consider the following potential sequence of events:

1. In either call above, a temporary `std::shared_ptr<Widget>` object is constructed to hold the raw pointer resulting from "`new Widget`". Call this object *`temp`*.
2. `push_back` takes *`temp`* by reference. During allocation of a list node to hold a copy of *`temp`*, an out-of-memory exception gets thrown.
3. As the exception propagates out of `push_back`, *`temp`* is destroyed. Being the sole `std::shared_ptr` referring to the `Widget` it's managing, it automatically releases that `Widget`, in this case by calling `killWidget`.

Even though an exception occurred, nothing leaks: the `Widget` created via "`new Widget`" in the call to `push_back` is released in the destructor of the `std::shared_ptr` that was created to manage it (*`temp`*). Life is good.

Now consider what happens if `emplace_back` is called instead of `push_back`:

```
ptrs.emplace_back(new Widget, killWidget);
```

1. The raw pointer resulting from "`new Widget`" is perfect-forwarded to the point inside `emplace_back` where a list node is to be allocated. That allocation fails, and an out-of-memory exception is thrown.

2. As the exception propagates out of `emplace_back`, the raw pointer that was the only way to get at the `Widget` on the heap is lost. That `Widget` (and any resources it owns) is leaked.

In this scenario, life is *not* good, and the fault doesn't lie with `std::shared_ptr`. The same kind of problem can arise through the use of `std::unique_ptr` with a custom deleter. Fundamentally, the effectiveness of resource-managing classes like `std::shared_ptr` and `std::unique_ptr` is predicated on resources (such as raw pointers from `new`) being *immediately* passed to constructors for resource-managing objects. The fact that functions like `std::make_shared` and `std::make_unique` automate this is one of the reasons they're so important.

In calls to the insertion functions of containers holding resource-managing objects (e.g., `std::list<std::shared_ptr<Widget>>`), the functions' parameter types generally ensure that nothing gets between acquisition of a resource (e.g., use of `new`) and construction of the object managing the resource. In the emplacement functions, perfect-forwarding defers the creation of the resource-managing objects until they can be constructed in the container's memory, and that opens a window during which exceptions can lead to resource leaks. All standard containers are susceptible to this problem. When working with containers of resource-managing objects, you must take care to ensure that if you choose an emplacement function over its insertion counterpart, you're not paying for improved code efficiency with diminished exception safety.

Frankly, you shouldn't be passing expressions like "`new Widget`" to `emplace_back` or `push_back` or most any other function, anyway, because, as Item 21 explains, this leads to the possibility of exception safety problems of the kind we just examined. Closing the door requires taking the pointer from "`new Widget`" and turning it over to a resource-managing object in a standalone statement, then passing that object as an rvalue to the function you originally wanted to pass "`new Widget`" to. (Item 21 covers this technique in more detail.) The code using `push_back` should therefore be written more like this:

```
std::shared_ptr<Widget> spw(new Widget,    // create Widget and
                            killWidget);   // have spw manage it

ptrs.push_back(std::move(spw));            // add spw as rvalue
```

The `emplace_back` version is similar:

```
std::shared_ptr<Widget> spw(new Widget, killWidget);
ptrs.emplace_back(std::move(spw));
```

Either way, the approach incurs the cost of creating and destroying `spw`. Given that the motivation for choosing emplacement over insertion is to avoid the cost of a temporary object of the type held by the container, yet that's conceptually what spw is, emplacement functions are unlikely to outperform insertion functions when you're adding resource-managing objects to a container and you follow the proper practice of ensuring that nothing can intervene between acquiring a resource and turning it over to a resource-managing object.

A second noteworthy aspect of emplacement functions is their interaction with explicit constructors. In honor of C++11's support for regular expressions, suppose you create a container of regular expression objects:

```
std::vector<std::regex> regexes;
```

Distracted by your colleagues' quarreling over the ideal number of times per day to check one's Facebook account, you accidentally write the following seemingly meaningless code:

```
regexes.emplace_back(nullptr);     // add nullptr to container
                                   // of regexes?
```

You don't notice the error as you type it, and your compilers accept the code without complaint, so you end up wasting a bunch of time debugging. At some point, you discover that you have inserted a null pointer into your container of regular expressions. But how is that possible? Pointers aren't regular expressions, and if you tried to do something like this,

```
std::regex r = nullptr;            // error! won't compile
```

compilers would reject your code. Interestingly, they would also reject it if you called push_back instead of emplace_back:

```
regexes.push_back(nullptr);        // error! won't compile
```

The curious behavior you're experiencing stems from the fact that std::regex objects can be constructed from character strings. That's what makes useful code like this legal:

```
std::regex upperCaseWord("[A-Z]+");
```

Creation of a std::regex from a character string can exact a comparatively large runtime cost, so, to minimize the likelihood that such an expense will be incurred unintentionally, the std::regex constructor taking a const char* pointer is explicit. That's why these lines don't compile:

```
std::regex r = nullptr;            // error! won't compile

regexes.push_back(nullptr);        // error! won't compile
```

In both cases, we're requesting an implicit conversion from a pointer to a std::regex, and the explicitness of that constructor prevents such conversions.

In the call to `emplace_back`, however, we're not claiming to pass a `std::regex` object. Instead, we're passing a *constructor argument* for a `std::regex` object. That's not considered an implicit conversion request. Rather, it's viewed as if you'd written this code:

```
std::regex r(nullptr);            // compiles
```

If the laconic comment "compiles" suggests a lack of enthusiasm, that's good, because this code, though it will compile, has undefined behavior. The `std::regex` constructor taking a `const char*` pointer requires that the pointed-to string comprise a valid regular expression, and the null pointer fails that requirement. If you write and compile such code, the best you can hope for is that it crashes at runtime. If you're not so lucky, you and your debugger could be in for a special bonding experience.

Setting aside `push_back`, `emplace_back`, and bonding for a moment, notice how these very similar initialization syntaxes yield different results:

```
std::regex r1 = nullptr;          // error! won't compile

std::regex r2(nullptr);           // compiles
```

In the official terminology of the Standard, the syntax used to initialize `r1` (employing the equals sign) corresponds to what is known as *copy initialization*. In contrast, the syntax used to initialize `r2` (with the parentheses, although braces may be used instead) yields what is called *direct initialization*. Copy initialization is not permitted to use `explicit` constructors. Direct initialization is. That's why the line initializing `r1` doesn't compile, but the line initializing `r2` does.

But back to `push_back` and `emplace_back` and, more generally, the insertion functions versus the emplacement functions. Emplacement functions use direct initialization, which means they may use `explicit` constructors. Insertion functions employ copy initialization, so they can't. Hence:

```
regexes.emplace_back(nullptr);  // compiles. Direct init permits
                                // use of explicit std::regex
                                // ctor taking a pointer

regexes.push_back(nullptr);     // error! copy init forbids
                                // use of that ctor
```

The lesson to take away is that when you use an emplacement function, be especially careful to make sure you're passing the correct arguments, because even `explicit` constructors will be considered by compilers as they try to find a way to interpret your code as valid.

Things to Remember

- In principle, emplacement functions should sometimes be more efficient than their insertion counterparts, and they should never be less efficient.
- In practice, they're most likely to be faster when (1) the value being added is constructed into the container, not assigned; (2) the argument type(s) passed differ from the type held by the container; and (3) the container won't reject the value being added due to it being a duplicate.
- Emplacement functions may perform type conversions that would be rejected by insertion functions.

Index

Symbols

A

B

C

E

F

G

H

I

S

T

About the Author

Scott Meyers is one of the world's foremost experts on C++. A sought-after trainer, consultant, and conference presenter, his *Effective C++* books (*Effective C++*, *More Effective C++*, and *Effective STL*) have set the bar for C++ programming guidance for more than 20 years. He has a Ph.D. in computer science from Brown University. His website is *aristeia.com*.

Colophon

The animal on the cover of *Effective Modern C++* is a *Rose-crowned fruit dove* (*Ptilinopus regina*). This species of dove also goes by the names pink-capped fruit dove or Swainson's fruit dove. It is distinguished by its striking plumage: grey head and breast, orange belly, whitish throat, yellow-orange iris, and grey green bill and feet.

Distributed in lowland rainforests in eastern Australia, monsoon forests in northern Australia, and the Lesser Sunda Islands and Maluku Islands of Indonesia, the Rose-crowned fruit dove's diet consists of various fruits like figs (which it swallows whole), palms, and vines. Camphor Laurel, a large evergreen tree, is another food source for the fruit dove. They feed—in pairs, small parties, or singly—in rainforest canopies, usually in the morning or late afternoon. To hydrate, they get water from leaves or dew, not from the ground.

The fruit dove is considered vulnerable in New South Wales due to rainforest clearing and fragmentation, logging, weeds, fire regime–altered habitats, and the removal of Laurel Camphor without adequate alternatives.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from Wood's *Illustrated Natural History*, bird volume. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.