```
};

class B {
public:
   explicit B(int x = 0, bool b = true);      // default constructor; see below
};                                            // for info on "explicit"
```



```
class C {
public:
   explicit C(int x);                         // not a default constructor
};
```

The constructors for classes B and C are declared explicit here. That prevents them from being used to perform implicit type conversions, though they may still be used for explicit type conversions:

```
void doSomething(B bObject);          // a function taking an object of
                                      // type B

B bObj1;                              // an object of type B
doSomething(bObj1);                   // fine, passes a B to doSomething
B bObj2(28);                          // fine, creates a B from the int 28
                                      // (the bool defaults to true)

doSomething(28);                      // error! doSomething takes a B,
                                      // not an int, and there is no
                                      // implicit conversion from int to B

doSomething(B(28));                   // fine, uses the B constructor to
                                      // explicitly convert (i.e., cast) the
                                      // int to a B for this call. (See
                                      // Item 27 for info on casting.)
```

Constructors declared explicit are usually preferable to non-explicit ones, because they prevent compilers from performing unexpected (often unintended) type conversions. Unless I have a good reason for allowing a constructor to be used for implicit type conversions, I declare it explicit. I encourage you to follow the same policy.

Read carefully when you see what appears to be an assignment, because the "=" syntax can also be used to call the copy constructor:

```
Widget w3 = w2;                    // invoke copy constructor!
```

Fortunately, copy construction is easy to distinguish from copy assignment. If a new object is being defined (such as w3 in the statement above), a constructor has to be called; it can't be an assignment. If no new object is being defined (such as in the "w1 = w2" statement above), no constructor can be involved, so it's an assignment.

The copy constructor is a particularly important function, because it defines how an object is passed by value. For example, consider this:

```
bool hasAcceptableQuality(Widget w);
...
Widget aWidget;
```

after all, bitwise const, and that's all compilers check for. But look what it allows to happen:

```
const CTextBlock cctb("Hello");          // declare constant object
char *pc = &cctb[0];                     // call the const operator[] to get a
                                         // pointer to cctb's data
*pc = 'J';                               // cctb now has the value "Jello"
```

Surely there is something wrong when you create a constant object with a particular value and you invoke only const member functions on it, yet you still change its value!

This leads to the notion of logical constness. Adherents to this philosophy — and you should be among them — argue that a const member function might modify some of the bits in the object on which it's invoked, but only in ways that clients cannot detect. For example, your CTextBlock class might want to cache the length of the textblock whenever it's requested:

```
class CTextBlock {
public:
  ...
  std::size_t length() const;
```

```
class CTextBlock {
public:
  ...
  std::size_t length() const;

private:
  char *pText;
  mutable std::size_t textLength;          // these data members may
  mutable bool lengthIsValid;              // always be modified, even in
};                                         // const member functions

std::size_t CTextBlock::length() const
{
  if (!lengthIsValid) {
    textLength = std::strlen(pText);       // now fine
    lengthIsValid = true;                  // also fine
  }

  return textLength;
}
```

Avoiding Duplication in const and Non-const Member Functions

Even more worth knowing is that trying to do things the other way around — avoiding duplication by having the const version call the non-const version — is *not* something you want to do. Remember, a const member function promises never to change the logical state of its object, but a non-const member function makes no such promise. If you were to call a non-const function from a const one, you'd run the risk that the object you'd promised not to modify would be changed. That's why having a const member function call a non-const one is wrong: the object could be changed. In fact, to get the code to compile, you'd have to use a const_cast to get rid of the const on *this, a clear sign of trouble. The reverse calling sequence — the one we used above — is safe: the non-const member function can do whatever it wants with an object, so calling a const member function imposes no risk. That's why a static_cast works on *this in that case: there's no const-related danger.

As I noted at the beginning of this Item, const is a wonderful thing. On pointers and iterators; on the objects referred to by pointers, iterators,

```
class ABEntry {                                  // ABEntry = "Address Book Entry"
public:
  ABEntry(const std::string& name, const std::string& address,
          const std::list<PhoneNumber>& phones);
private:
  std::string theName;
  std::string theAddress;
  std::list<PhoneNumber> thePhones;
  int numTimesConsulted;
};

ABEntry::ABEntry(const std::string& name, const std::string& address,
                 const std::list<PhoneNumber>& phones)
{
  theName = name;                                // these are all assignments,
  theAddress = address;                          // not initializations
  thePhones = phones;
  numTimesConsulted = 0;
}
```

non-local static objects defined in different translation units. How can you be sure that tfs will be initialized before tempDir?

You can't. Again, *the relative order of initialization of non-local static objects defined in different translation units is undefined*. There is a reason for this. Determining the "proper" order in which to initialize non-local static objects is hard. Very hard. Unsolvably hard. In its most general form — with multiple translation units and non-local static objects generated through implicit template instantiations (which may themselves arise via implicit template instantiations) — it's not only impossible to determine the right order of initialization, it's typically not even worth looking for special cases where it *is* possible to determine the right order.

Fortunately, a small design change eliminates the problem entirely. All that has to be done is to move each non-local static object into its own function, where it's declared static. These functions return references to the objects they contain. Clients then call the functions instead of referring to the objects. In other words, non-local static objects are replaced with *local* static objects. (Aficionados of design patterns will recognize this as a common implementation of the Singleton pattern.†)

Fortunately, a small design change eliminates the problem entirely. All that has to be done is to move each non-local static object into its own function, where it's declared static. These functions return references to the objects they contain. Clients then call the functions instead of referring to the objects. In other words, non-local static objects are replaced with *local* static objects. (Aficionados of design patterns will recognize this as a common implementation of the Singleton pattern.†)

This approach is founded on C++'s guarantee that local static objects are initialized when the object's definition is first encountered during a call to that function. So if you replace direct accesses to non-local

† Actually, it's only *part* of a Singleton implementation. An essential part of Singleton I ignore in this Item is preventing the creation of multiple objects of a particular type.

static objects with calls to functions that return references to local static objects, you're guaranteed that the references you get back will refer to initialized objects. As a bonus, if you never call a function emulating a non-local static object, you never incur the cost of constructing and destructing the object, something that can't be said for true non-local static objects.

Here's the technique applied to both tfs and tempDir:

```
class FileSystem { ... };              // as before

FileSystem& tfs()                      // this replaces the tfs object; it could be
{                                      // static in the FileSystem class
  static FileSystem fs;                // define and initialize a local static object
  return fs;                           // return a reference to it
}

class Directory { ... };               // as before

Directory::Directory( params )         // as before, except references to tfs are
{                                      // now to tfs()
  ...
  std::size_t disks = tfs().numDisks();
```

Here's the technique applied to both tfs and tempDir:

```
class FileSystem { ... };              // as before

FileSystem& tfs()                      // this replaces the tfs object; it could be
{                                      // static in the FileSystem class
  static FileSystem fs;                // define and initialize a local static object
  return fs;                           // return a reference to it
}

class Directory { ... };               // as before

Directory::Directory( params )         // as before, except references to tfs are
{                                      // now to tfs()
  ...
  std::size_t disks = tfs().numDisks();
  ...
}

Directory& tempDir()                   // this replaces the tempDir object; it
{                                      // could be static in the Directory class
  static Directory td( params );       // define/initialize local static object
  return td;                           // return reference to it
}
```

compiler-generated copy assignment operator should do?

Faced with this conundrum, C++ refuses to compile the code. If you want to support copy assignment in a class containing a reference member, you must define the copy assignment operator yourself. Compilers behave similarly for classes containing const members (such as objectValue in the modified class above). It's not legal to modify const members, so compilers are unsure how to treat them during an implicitly generated assignment function. Finally, compilers reject implicit copy assignment operators in derived classes that inherit from base classes declaring the copy assignment operator private. After all, compiler-generated copy assignment operators for derived classes are supposed to handle base class parts, too (see Item 12), but in doing so, they certainly can't invoke member functions the derived class has no right to call.

Things to Remember

- Compilers may implicitly generate a class's default constructor, copy constructor, copy assignment operator, and destructor.

Item 6: Explicitly disallow the use of compiler-

member or a friend function, the linker will complain.

It's possible to move the link-time error up to compile time (always a good thing — earlier error detection is better than later) by declaring the copy constructor and copy assignment operator private not in HomeForSale itself, but in a base class specifically designed to prevent copying. The base class is simplicity itself:

```
class Uncopyable {
protected:                                      // allow construction
  Uncopyable() {}                               // and destruction of
  ~Uncopyable() {}                              // derived objects...

private:
  Uncopyable(const Uncopyable&);                // ...but prevent copying
  Uncopyable& operator=(const Uncopyable&);
};
```

To keep HomeForSale objects from being copied, all we have to do now is inherit from Uncopyable:

```
class HomeForSale: private Uncopyable {         // class no longer
  ...                                           // declares copy ctor or
};                                              // copy assign. operator
```

This works, because compilers will try to generate a copy constructor

```
class TimeKeeper {
public:
   TimeKeeper();
   ~TimeKeeper();
   ...
};

class AtomicClock: public TimeKeeper { ... };

class WaterClock: public TimeKeeper { ... };

class WristWatch: public TimeKeeper { ... };
```

Many clients will want access to the time without worrying about the details of how it's calculated, so a *factory function* — a function that returns a base class pointer to a newly-created derived class object — can be used to return a pointer to a timekeeping object:

```
TimeKeeper* getTimeKeeper();          // returns a pointer to a dynamic-
                                      // ally allocated object of a class
                                      // derived from TimeKeeper
```

In keeping with the conventions of factory functions, the objects returned by getTimeKeeper are on the heap, so to avoid leaking memory and other resources, it's important that each returned object be properly deleted:

```
TimeKeeper *ptk = getTimeKeeper();    // get dynamically allocated object
                                      // from TimeKeeper hierarchy
...                                   // use it
delete ptk;                           // release it to avoid resource leak
```

The problem is that getTimeKeeper returns a pointer to a derived class object (e.g., AtomicClock), that object is being deleted via a base class pointer (i.e., a TimeKeeper* pointer), and the base class (TimeKeeper) has a *non-virtual destructor*. This is a recipe for disaster, because C++

specifies that when a derived class object is deleted through a pointer to a base class with a non-virtual destructor, results are undefined. What typically happens at runtime is that the derived part of the object is never destroyed. If a call to getTimeKeeper happened to return a pointer to an AtomicClock object, the AtomicClock part of the object

a pointer to an AtomicClock object, the AtomicClock part of the object (i.e., the data members declared in the AtomicClock class) would probably not be destroyed, nor would the AtomicClock destructor run. However, the base class part (i.e., the TimeKeeper part) typically would be destroyed, thus leading to a curious "partially destroyed" object. This is an excellent way to leak resources, corrupt data structures, and spend a lot of time with a debugger.

Eliminating the problem is simple: give the base class a virtual destructor. Then deleting a derived class object will do exactly what you want. It will destroy the entire object, including all its derived class parts:

```
class TimeKeeper {
public:
  TimeKeeper();
  virtual ~TimeKeeper();
  ...
};

TimeKeeper *ptk = getTimeKeeper();

...

delete ptk;                                    // now behaves correctly
```

from functions written in other languages unless you explicitly compensate for the vptr, which is itself an implementation detail and hence unportable.

The bottom line is that gratuitously declaring all destructors virtual is just as wrong as never declaring them virtual. In fact, many people summarize the situation this way: declare a virtual destructor in a class if and only if that class contains at least one virtual function.

It is possible to get bitten by the non-virtual destructor problem even in the complete absence of virtual functions. For example, the standard string type contains no virtual functions, but misguided programmers sometimes use it as a base class anyway:

```
class SpecialString: public std::string {      // bad idea! std::string has a
    ...                                        // non-virtual destructor
};
```

At first glance, this may look innocuous, but if anywhere in an application you somehow convert a pointer-to-SpecialString into a pointer-to-

classes.)

Occasionally it can be convenient to give a class a pure virtual destructor. Recall that pure virtual functions result in *abstract* classes — classes that can't be instantiated (i.e., you can't create objects of that type). Sometimes, however, you have a class that you'd like to be abstract, but you don't have any pure virtual functions. What to do? Well, because an abstract class is intended to be used as a base class, and because a base class should have a virtual destructor, and because a pure virtual function yields an abstract class, the solution is simple: declare a pure virtual destructor in the class you want to be abstract. Here's an example:

```
class AWOV {                              // AWOV = "Abstract w/o Virtuals"
public:
  virtual ~AWOV() = 0;                    // declare pure virtual destructor
};
```

This class has a pure virtual function, so it's abstract, and it has a virtual destructor, so you won't have to worry about the destructor problem. There is one twist, however: you must provide a *definition* for the pure virtual destructor:

```
AWOV::~AWOV() {}                          // definition of pure virtual dtor
```

The way destructors work is that the most derived class's destructor is called first, then the destructor of each base class is called. Compil-

ers will generate a call to ~AWOV from its derived classes' destructors, so you have to be sure to provide a body for the function. If you don't, the linker will complain.

The rule for giving base classes virtual destructors applies only to *polymorphic* base classes — to base classes designed to allow the manipulation of derived class types through base class interfaces. TimeKeeper is a polymorphic base class, because we expect to be able

ers will generate a call to ~AWOV from its derived classes' destructors, so you have to be sure to provide a body for the function. If you don't, the linker will complain.

The rule for giving base classes virtual destructors applies only to *polymorphic* base classes — to base classes designed to allow the manipulation of derived class types through base class interfaces. TimeKeeper is a polymorphic base class, because we expect to be able to manipulate AtomicClock and WaterClock objects, even if we have only TimeKeeper pointers to them.

Not all base classes are designed to be used polymorphically. Neither the standard string type, for example, nor the STL container types are designed to be base classes at all, much less polymorphic ones. Some classes are designed to be used as base classes, yet are not designed to be used polymorphically. Such classes — examples include Uncopyable from Item 6 and input_iterator_tag from the standard library (see Item 47) — are not designed to allow the manipulation of derived class objects via base class interfaces. As a result, they don't need virtual destructors.

Item 8: Prevent exceptions from leaving destructors.

C++ doesn't prohibit destructors from emitting exceptions, but it certainly discourages the practice. With good reason. Consider:

```
class Widget {
public:
  ...
  ~Widget() { ... }                    // assume this might emit an exception
};

void doSomething()
{
  std::vector<Widget> v;
  ...
}                                      // v is automatically destroyed here
```

When the vector v is destroyed, it is responsible for destroying all the Widgets it contains. Suppose v has ten Widgets in it, and during destruction of the first one, an exception is thrown. The other nine

Widgets still have to be destroyed (otherwise any resources they hold would be leaked), so v should invoke their destructors. But suppose that during those calls, a second Widget destructor throws an exception. Now there are two simultaneously active exceptions, and that's one too many for C++. Depending on the precise conditions under which such pairs of simultaneously active exceptions arise, program execution either terminates or yields undefined behavior. In this example, it yields undefined behavior. It would yield equally undefined behavior using any other standard library container (e.g., list, set), any container in TR1 (see Item 54), or even an array. Not that containers or arrays are required to get into trouble. Premature program termination or undefined behavior can result from destructors emitting exceptions even without using containers and arrays. C++ does *not* like destructors that emit exceptions!

That's easy enough to understand, but what should you do if your

This is fine as long as the call to close succeeds, but if the call yields an exception, DBConn's destructor will propagate that exception, i.e., allow it to leave the destructor. That's a problem, because destructors that throw mean trouble.

There are two primary ways to avoid the trouble. DBConn's destructor could:

- **Terminate the program** if close throws, typically by calling abort:

```
DBConn::~DBConn()
{
    try { db.close(); }
    catch (...) {
        make log entry that the call to close failed;
        std::abort();
    }
}
```

This is a reasonable option if the program cannot continue to run after an error is encountered during destruction. It has the advantage that if allowing the exception to propagate from the destructor would lead to undefined behavior, this prevents that from happening. That is, calling abort may forestall undefined behavior.

- **Swallow the exception** arising from the call to close:

```
DBConn::~DBConn()
{
   try { db.close(); }
   catch (...) {
      make log entry that the call to close failed;
   }
}
```

In general, swallowing exceptions is a bad idea, because it suppresses important information — *something failed!* Sometimes, however, swallowing exceptions is preferable to running the risk of

```
class DBConn {
public:
   ...
   void close()                                    // new function for
   {                                               // client use
      db.close();
      closed = true;
   }

   ~DBConn()
   {
      if (!closed) {
         try {                                     // close the connection
            db.close();                            // if the client didn't
         }
         catch (...) {                             // if closing fails,
            make log entry that call to close failed;   // note that and
            ...                                    // terminate or swallow
         }
      }
   }

private:
   DBConnection db;
   bool closed;
```

Moving the responsibility for calling close from DBConn's destructor to DBConn's client (with DBConn's destructor containing a "backup" call) may strike you as an unscrupulous shift of burden. You might even view it as a violation of Item 18's advice to make interfaces easy to use correctly. In fact, it's neither. If an operation may fail by throwing an exception and there may be a need to handle that exception, the exception *has to come from some non-destructor function*. That's

because destructors that emit exceptions are dangerous, always running the risk of premature program termination or undefined behavior. In this example, telling clients to call close themselves doesn't impose a burden on them; it gives them an opportunity to deal with

find that opportunity useful (perhaps because they believe that no error will really occur), they can ignore it, relying on DBConn's destructor to call close for them. If an error occurs at that point — if close *does* throw — they're in no position to complain if DBConn swallows the exception or terminates the program. After all, they had first crack at dealing with the problem, and they chose not to use it.

Things to Remember

- Destructors should never emit exceptions. If functions called in a destructor may throw, the destructor should catch any exceptions, then swallow them or terminate the program.
- If class clients need to be able to react to exceptions thrown during an operation, the class should provide a regular (i.e., non-destructor) function that performs the operation.

Item 9: Never call virtual functions during construction or destruction.

I'll begin with the recap: you shouldn't call virtual functions during construction or destruction, because the calls won't do what you

It's not always so easy to detect calls to virtual functions during construction or destruction. If Transaction had multiple constructors, each of which had to perform some of the same work, it would be good software engineering to avoid code replication by putting the common initialization code, including the call to logTransaction, into a private non-virtual initialization function, say, init:

```
class Transaction {
public:
   Transaction()
   { init(); }                                   // call to non-virtual...

   virtual void logTransaction() const = 0;
   ...

private:
   void init()
   {
      ...
      logTransaction();                          // ...that calls a virtual!
   }
};
```

In other words, since you can't use virtual functions to call down from base classes during construction, you can compensate by having derived classes pass necessary construction information up to base class constructors instead.

In this example, note the use of the (private) static function createLogString in BuyTransaction. Using a helper function to create a value to pass to a base class constructor is often more convenient (and more readable) than going through contortions in the member initialization list to give the base class what it needs. By making the function static, there's no danger of accidentally referring to the nascent BuyTransaction object's as-yet-uninitialized data members. That's important, because the fact that those data members will be in an undefined state is why calling virtual functions during base class construction and destruction doesn't go down into derived classes in the first place.

Things to Remember

- Don't call virtual functions during construction or destruction, because such calls will never go to a more derived class than that of the currently executing constructor or destructor.

Item 10: Have assignment operators return a reference to *this.

One of the interesting things about assignments is that you can chain them together:

```
int x, y, z;
x = y = z = 15;                    // chain of assignments
```

Also interesting is that assignment is right-associative, so the above assignment chain is parsed like this:

```
x = (y = (z = 15));
```

Here, 15 is assigned to z, then the result of that assignment (the updated z) is assigned to y, then the result of that assignment (the updated y) is assigned to x.

The way this is implemented is that assignment returns a reference to its left-hand argument, and that's the convention you should follow when you implement assignment operators for your classes:

```
class Widget {
public:
```

Personally, I worry that this approach sacrifices clarity at the altar of cleverness, but by moving the copying operation from the body of the function to construction of the parameter, it's a fact that compilers can sometimes generate more efficient code.

Things to Remember

- Make sure operator= is well-behaved when an object is assigned to itself. Techniques include comparing addresses of source and target objects, careful statement ordering, and copy-and-swap.
- Make sure that any function operating on more than one object behaves correctly if two or more of the objects are the same.

Item 12: Copy all parts of an object.

In well-designed object-oriented systems that encapsulate the internal parts of objects, only two functions copy objects: the aptly named copy constructor and copy assignment operator. We'll call these the *copying functions*. Item 5 observes that compilers will generate the

```
Customer::Customer(const Customer& rhs)
: name(rhs.name)                                    // copy rhs's data
{
  logCall("Customer copy constructor");
}

Customer& Customer::operator=(const Customer& rhs)
{
  logCall("Customer copy assignment operator");
  name = rhs.name;                                  // copy rhs's data
  return *this;                                     // see Item 10
}
```

Everything here looks fine, and in fact everything is fine — until another data member is added to Customer:

```
public:
  ...                                                    // as before

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

At this point, the existing copying functions are performing a *partial copy*: they're copying the customer's name, but not its lastTransaction. Yet most compilers say nothing about this, not even at maximal warning level (see also Item 53). That's their revenge for your writing the copying functions yourself. You reject the copying functions they'd write, so they don't tell you if your code is incomplete. The conclusion is obvious: if you add a data member to a class, you need to make sure that you update the copying functions, too. (You'll also need to update all the constructors (see Items 4 and 45) as well as any nonstandard forms of operator= in the class (Item 10 gives an example). If you forget, compilers are unlikely to remind you.)

One of the most insidious ways this issue can arise is through inheritance. Consider:

```
class PriorityCustomer: public Customer {              // a derived class
public:
```

to do.

Trying things the other way around — having the copy constructor call the copy assignment operator — is equally nonsensical. A constructor initializes new objects, but an assignment operator applies only to objects that have already been initialized. Performing an assignment on an object under construction would mean doing something to a not-yet-initialized object that makes sense only for an initialized object. Nonsense! Don't try it.

Instead, if you find that your copy constructor and copy assignment operator have similar code bodies, eliminate the duplication by creating a third member function that both call. Such a function is typically private and is often named init. This strategy is a safe, proven way to eliminate code duplication in copy constructors and copy assignment operators.

Things to Remember

- Copying functions should be sure to copy all of an object's data members and all of its base class parts.
- Don't try to implement one of the copying functions in terms of the other. Instead, put common functionality in a third function that

Many resources are dynamically allocated on the heap, are used only within a single block or function, and should be released when control leaves that block or function. The standard library's auto_ptr is tailor-made for this kind of situation. auto_ptr is a pointer-like object (a *smart pointer*) whose destructor automatically calls delete on what it points to. Here's how to use auto_ptr to prevent f's potential resource leak:

```
void f()
{
  std::auto_ptr<Investment> pInv(createInvestment());   // call factory
                                                        // function
  ...                                                   // use pInv as
                                                        // before
}                                                       // automatically
                                                        // delete pInv via
                                                        // auto_ptr's dtor
```

This simple example demonstrates the two critical aspects of using objects to manage resources:

- **Resources are acquired and immediately turned over to resource-managing objects.** Above, the resource returned by createInvestment is used to initialize the auto_ptr that will manage it. In fact, the idea of using objects to manage resources is often called *Resource Acquisition Is Initialization* (RAII), because it's so common

Because an auto_ptr automatically deletes what it points to when the auto_ptr is destroyed, it's important that there never be more than one auto_ptr pointing to an object. If there were, the object would be deleted more than once, and that would put your program on the fast track to undefined behavior. To prevent such problems, auto_ptrs have an unusual characteristic: copying them (via copy constructor or copy

assignment operator) sets them to null, and the copying pointer assumes sole ownership of the resource!

```
std::auto_ptr<Investment>
  pInv1(createInvestment());                  // pInv1 points to the
                                              // object returned from
                                              // createInvestment

std::auto_ptr<Investment> pInv2(pInv1);       // pInv2 now points to the
```

An alternative to auto_ptr is a *reference-counting smart pointer* (RCSP). An RCSP is a smart pointer that keeps track of how many objects point to a particular resource and automatically deletes the resource when nobody is pointing to it any longer. As such, RCSPs offer behavior that is similar to that of garbage collection. Unlike garbage collection, however, RCSPs can't break cycles of references (e.g., two otherwise unused objects that point to one another).

TR1's tr1::shared_ptr (see Item 54) is an RCSP, so you could write f this way:

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>
    plnv(createInvestment());                    // call factory function
  ...                                            // use plnv as before
}                                                // automatically delete
                                                 // plnv via shared_ptr's dtor
```

This code looks almost the same as that employing auto_ptr, but copying shared_ptrs behaves much more naturally:

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>               // plnv1 points to the
```

This code looks almost the same as that employing auto_ptr, but copying shared_ptrs behaves much more naturally:

```
void f()
{
  ...
  std::tr1::shared_ptr<Investment>
    pInv1(createInvestment());          // pInv1 points to the
                                        // object returned from
                                        // createInvestment
```



```
  std::tr1::shared_ptr<Investment>
    pInv2(pInv1);                       // both pInv1 and pInv2 now
                                        // point to the object
  pInv1 = pInv2;                        // ditto — nothing has
                                        // changed
  ...
```

```
    std::tr1::shared_ptr<Investment>          // both pInv1 and pInv2 now
      pInv2(pInv1);                           // point to the object

    pInv1 = pInv2;                            // ditto — nothing has
                                              // changed

    ...
}                                             // pInv1 and pInv2 are
                                              // destroyed, and the
                                              // object they point to is
                                              // automatically deleted
```

Because copying tr1::shared_ptrs works "as expected," they can be used in STL containers and other contexts where auto_ptr's unorthodox copying behavior is inappropriate.

Don't be misled, though. This Item isn't about auto_ptr, tr1::shared_ptr, or any other kind of smart pointer. It's about the importance of using objects to manage resources. auto_ptr and tr1::shared_ptr are just examples of objects that do that. (For more information on tr1:shared_ptr, consult Items 14, 18, and 54.)

Both auto_ptr and tr1::shared_ptr use delete in their destructors, not delete []. (Item 16 describes the difference.) That means that using auto_ptr or tr1::shared_ptr with dynamically allocated arrays is a bad idea, though, regrettably, one that will compile:

Both auto_ptr and tr1::shared_ptr use delete in their destructors, not delete []. (Item 16 describes the difference.) That means that using auto_ptr or tr1::shared_ptr with dynamically allocated arrays is a bad idea, though, regrettably, one that will compile:

```
std::auto_ptr<std::string>
  aps(new std::string[10]);                  // bad idea! the wrong
                                             // delete form will be used

std::tr1::shared_ptr<int> spi(new int[1024]); // same problem
```

You may be surprised to discover that there is nothing like auto_ptr or tr1::shared_ptr for dynamically allocated arrays in C++, not even in TR1. That's because vector and string can almost always replace dynamically allocated arrays. If you still think it would be nice to have auto_ptr- and tr1::shared_ptr-like classes for arrays, look to Boost (see Item 55). There you'll be pleased to find the boost::scoped_array and boost::shared_array classes that offer the behavior you're looking for.

This Item's guidance to use objects to manage resources suggests that if you're releasing resources manually (e.g., using delete other than in a resource-managing class), you're doing something wrong. Pre-canned resource-managing classes like auto_ptr and tr1::shared_ptr often make following this Item's advice easy, but sometimes you're using a resource where these pre-fab classes don't do what you need. When that's the case, you'll need to craft your own resource-managing classes. That's not terribly difficult to do, but it does involve some subtleties you'll need to consider. Those considerations are the topic of Items 14 and 15.

As a final comment, I have to point out that createInvestment's raw pointer return type is an invitation to a resource leak, because it's so easy for callers to forget to call delete on the pointer they get back. (Even if they use an auto_ptr or tr1::shared_ptr to perform the delete, they still have to remember to store createInvestment's return value in a smart pointer object.) Combatting that problem calls for an interface modification to createInvestment, a topic I address in Item 18.

Things to Remember

- To prevent resource leaks, use RAII objects that acquire resources in their constructors and release them in their destructors.
- Two commonly useful RAII classes are tr1::shared_ptr and auto_ptr. tr1::shared_ptr is usually the better choice, because its behavior when copied is intuitive. Copying an auto_ptr sets it to null.

Item 14: Think carefully about copying behavior in resource-managing classes.

Item 13 introduces the idea of *Resource Acquisition Is Initialization* (RAII) as the backbone of resource-managing classes, and it describes

Fortunately, tr1::shared_ptr allows specification of a "deleter" — a function or function object to be called when the reference count goes to zero. (This functionality does not exist for auto_ptr, which *always* deletes its pointer.) The deleter is an optional second parameter to the tr1::shared_ptr constructor, so the code would look like this:

```
class Lock {
public:
   explicit Lock(Mutex *pm)          // init shared_ptr with the Mutex
   : mutexPtr(pm, unlock)            // to point to and the unlock func
   {                                 // as the deleter†

      lock(mutexPtr.get());          // see Item 15 for info on "get"
   }

private:
   std::tr1::shared_ptr<Mutex> mutexPtr;      // use shared_ptr
};                                            // instead of raw pointer
```

In this example, notice how the Lock class no longer declares a destructor. That's because there's no need to. Item 5 explains that a class's destructor (regardless of whether it is compiler-generated or user-defined) automatically invokes the destructors of the class's non-static data members. In this example, that's mutexPtr. But mutexPtr's destructor will automatically call the tr1::shared_ptr's deleter — unlock, in this case — when the mutex's reference count

This rule is also noteworthy for the typedef-inclined, because it means that a typedef's author must document which form of delete should be employed when new is used to conjure up objects of the typedef type. For example, consider this typedef:

```
typedef std::string AddressLines[4];      // a person's address has 4 lines,
                                          // each of which is a string
```

Because AddressLines is an array, this use of new,

```
std::string *pal = new AddressLines;      // note that "new AddressLines"
                                          // returns a string*, just like
                                          // "new string[4]" would
```

must be matched with the *array* form of delete:

```
delete pal;                               // undefined!
delete [] pal;                            // fine
```

To avoid such confusion, abstain from typedefs for array types. That's easy, because the standard C++ library (see Item 54) includes string and vector, and those templates reduce the need for dynamically allocated arrays to nearly zero. Here, for example, AddressLines could be defined to be a vector of strings, i.e., the type vector<string>.

```
processWidget(std::tr1::shared_ptr<Widget>(new Widget), priority());
```

Surprisingly, although we're using object-managing resources everywhere here, this call may leak resources. It's illuminating to see how.

Before compilers can generate a call to processWidget, they have to evaluate the arguments being passed as its parameters. The second argument is just a call to the function priority, but the first argument, ("std::tr1::shared_ptr<Widget>(new Widget)") consists of two parts:

- Execution of the expression "new Widget".
- A call to the tr1::shared_ptr constructor.

Before processWidget can be called, then, compilers must generate code to do these three things:

- Call priority.
- Execute "new Widget".
- Call the tr1::shared_ptr constructor.

C++ compilers are granted considerable latitude in determining the order in which these things are to be done. (This is different from the way languages like Java and C# work, where function parameters are always evaluated in a particular order.) The "new Widget" expression must be executed before the tr1::shared_ptr constructor can be called, because the result of the expression is passed as an argument to the tr1::shared_ptr constructor, but the call to priority can be performed

way languages like Java and C# work, where function parameters are always evaluated in a particular order.) The "new Widget" expression must be executed before the tr1::shared_ptr constructor can be called, because the result of the expression is passed as an argument to the tr1::shared_ptr constructor, but the call to priority can be performed first, second, or third. If compilers choose to perform it second (something that may allow them to generate more efficient code), we end up with this sequence of operations:

1. Execute "new Widget".
2. Call priority.
3. Call the tr1::shared_ptr constructor.

But consider what will happen if the call to priority yields an exception. In that case, the pointer returned from "new Widget" will be lost, because it won't have been stored in the tr1::shared_ptr we were expecting would guard against resource leaks. A leak in the call to processWidget can arise because an exception can intervene between the time

in that case, the pointer returned from "new Widget" will be lost, because it won't have been stored in the tr1::shared_ptr we were expecting would guard against resource leaks. A leak in the call to processWidget can arise because an exception can intervene between the time

a resource is created (via "new Widget") and the time that resource is turned over to a resource-managing object.

The way to avoid problems like this is simple: use a separate statement to create the Widget and store it in a smart pointer, then pass the smart pointer to processWidget:

```
std::tr1::shared_ptr<Widget> pw(new Widget);     // store newed object
                                                 // in a smart pointer in a
                                                 // standalone statement

processWidget(pw, priority());                   // this call won't leak
```

Things to Remember

- Store newed objects in smart pointers in standalone statements. Failure to do this can lead to subtle resource leaks when exceptions are thrown.

of a computation, as in averageSoFar, above).

Suppose we have a public data member, and we eliminate it. How much code might be broken? All the client code that uses it, which is generally an *unknowably large* amount. Public data members are thus completely unencapsulated. But suppose we have a protected data member, and we eliminate it. How much code might be broken now? All the derived classes that use it, which is, again, typically an *unknowably large* amount of code. Protected data members are thus as unencapsulated as public ones, because in both cases, if the data members are changed, an unknowably large amount of client code is broken. This is unintuitive, but as experienced library implementers will tell you, it's still true. Once you've declared a data member public or protected and clients have started using it, it's very hard to change anything about that data member. Too much code has to be rewritten,

```
class Rational {
   ...                                                  // contains no operator*
};
const Rational operator*(const Rational& lhs,           // now a non-member
                         const Rational& rhs)           // function
{
   return Rational(lhs.numerator() * rhs.numerator(),
                   lhs.denominator() * rhs.denominator());
}
Rational oneFourth(1, 4);
Rational result;
result = oneFourth * 2;                                 // fine
result = 2 * oneFourth;                                 // hooray, it works!
```

This is certainly a happy ending to the tale, but there is a nagging worry. Should operator* be made a friend of the Rational class?

In this case, the answer is no, because operator* can be implemented entirely in terms of Rational's public interface. The code above shows one way to do it. That leads to an important observation: the opposite of a member function is a *non-member* function, not a friend function. Too many C++ programmers assume that if a function is related to a class and should not be a member (due, for example, to a need for type conversions on all arguments), it should be a friend. This example demonstrates that such reasoning is flawed. Whenever you can

be a friend.

This Item contains the truth and nothing but the truth, but it's not the whole truth. When you cross the line from Object-Oriented C++ into Template C++ (see Item 1) and make Rational a class *template* instead of a class, there are new issues to consider, new ways to resolve them, and some surprising design implications. Such issues, resolutions, and implications are the topic of Item 46.

Things to Remember

- If you need type conversions on all parameters to a function (including the one that would otherwise be pointed to by the this pointer), the function must be a non-member.

However, the default swap implementation may not thrill you. It involves copying three objects: a to temp, b to a, and temp to b. For some types, none of these copies are really necessary. For such types, the default swap puts you on the fast track to the slow lane.

Foremost among such types are those consisting primarily of a pointer to another type that contains the real data. A common manifestation of this design approach is the "pimpl idiom" ("pointer to implementation" — see Item 31). A Widget class employing such a design might look like this:

```
class WidgetImpl {                          // class for Widget data;
public:                                     // details are unimportant
  ...

private:
  int a, b, c;                              // possibly lots of data —
  std::vector<double> v;                    // expensive to copy!
  ...
};
```

```
class Widget {                                  // class using the pimpl idiom
public:
  Widget(const Widget& rhs);
  Widget& operator=(const Widget& rhs)          // to copy a Widget, copy its
  {                                             // WidgetImpl object. For
    ...                                         // details on implementing
    *pImpl = *(rhs.pImpl);                      // operator= in general,
    ...                                         // see Items 10, 11, and 12.
  }
  ...

private:
  WidgetImpl *pImpl;                            // ptr to object with this
};                                              // Widget's data
```

To swap the value of two Widget objects, all we really need to do is swap their pImpl pointers, but the default swap algorithm has no way to know that. Instead, it would copy not only three Widgets, but also three WidgetImpl objects. Very inefficient. Not a thrill.

What we'd like to do is tell std::swap that when Widgets are being swapped, the way to perform the swap is to swap their internal plmpl pointers. There is a way to say exactly that: specialize std::swap for Widget. Here's the basic idea, though it won't compile in this form:

```
namespace std {

  template<>                              // this is a specialized version
  void swap<Widget>(Widget& a,            // of std::swap for when T is
                    Widget& b)            // Widget
  {
    swap(a.plmpl, b.plmpl);               // to swap Widgets, swap their
  }                                       // plmpl pointers; this won't
compile

}
```

The "template<>" at the beginning of this function says that this is a *total template specialization* for std::swap, and the "<Widget>" after the name of the function says that the specialization is for when T is Widget. In other words, when the general swap template is applied to Widgets, this is the implementation that should be used. In general, we're not permitted to alter the contents of the std namespace, but we are allowed to totally specialize standard templates (like swap) for types of our own creation (such as Widget). That's what we're doing here.

our own creation (such as Widget). That's what we're doing here.

As I said, though, this function won't compile. That's because it's trying to access the pImpl pointers inside a and b, and they're private. We could declare our specialization a friend, but the convention is different: it's to have Widget declare a public member function called swap

that does the actual swapping, then specialize std::swap to call the member function:

```
class Widget {                                  // same as above, except for the
```

that does the actual swapping, then specialize std::swap to call the member function:

```
class Widget {                          // same as above, except for the
public:                                 // addition of the swap mem func
  ...
  void swap(Widget& other)
  {
    using std::swap;                    // the need for this declaration
                                        // is explained later in this Item

    swap(pImpl, other.pImpl);           // to swap Widgets, swap their
  }                                     // pImpl pointers
  ...
};

namespace std {

  template<>                            // revised specialization of
  void swap<Widget>(Widget& a,          // std::swap
                    Widget& b)
  {
    a.swap(b);                          // to swap Widgets, call their
  }                                     // swap member function

}
```

rule, these two swap characteristics go hand in hand, because highly efficient swaps are almost always based on operations on built-in types (such as the pointers underlying the pimpl idiom), and operations on built-in types never throw exceptions.

Things to Remember

- Provide a swap member function when std::swap would be inefficient for your type. Make sure your swap doesn't throw exceptions.
- If you offer a member swap, also offer a non-member swap that calls the member. For classes (not templates), specialize std::swap, too.
- When calling swap, employ a using declaration for std::swap, then call swap without namespace qualification.
- It's fine to totally specialize std templates for user-defined types, but never try to add something completely new to std.

```
// this function defines the variable "encrypted" too soon
std::string encryptPassword(const std::string& password)
{
   using namespace std;
   string encrypted;
   if (password.length() < MinimumPasswordLength) {
      throw logic_error("Password is too short");
   }
   ...                                  // do whatever is necessary to place an
                                        // encrypted version of password in encrypted
   return encrypted;
}
```

The object encrypted isn't *completely* unused in this function, but it's unused if an exception is thrown. That is, you'll pay for the construction and destruction of encrypted even if encryptPassword throws an exception. As a result, you're better off postponing encrypted's definition until you *know* you'll need it:

```
// this function postpones encrypted's definition until it's truly necessary
std::string encryptPassword(const std::string& password)
{
    using namespace std;

    if (password.length() < MinimumPasswordLength) {
        throw logic_error("Password is too short");
    }

    string encrypted;

    ...                                 // do whatever is necessary to place an
                                        // encrypted version of password in encrypted

    return encrypted;
}
```

This code still isn't as tight as it might be, because encrypted is defined without any initialization arguments. That means its default constructor will be used. In many cases, the first thing you'll do to an object is give it some value, often via an assignment. Item 4 explains why default-constructing an object and then assigning to it is less efficient than initializing it with the value you really want it to have. That analysis applies here, too. For example, suppose the hard part of encryptPassword is performed in this function:

"But what about loops?" you may wonder. If a variable is used only inside a loop, is it better to define it outside the loop and make an assignment to it on each loop iteration, or is it be better to define the variable inside the loop? That is, which of these general structures is better?

```
// Approach A: define outside loop
Widget w;
for (int i = 0; i < n; ++i) {
   w = some value dependent on i;
   ...
}
```

```
// Approach B: define inside loop

for (int i = 0; i < n; ++i) {
   Widget w(some value dependent on i);
   ...
}
```

Here I've switched from an object of type string to an object of type Widget to avoid any preconceptions about the cost of performing a construction, destruction, or assignment for the object.

In terms of Widget operations, the costs of these two approaches are as follows:

- Approach A: 1 constructor + 1 destructor + n assignments.
- Approach B: n constructors + n destructors.

For classes where an assignment costs less than a constructor-destructor pair, Approach A is generally more efficient. This is especially the case as n gets large. Otherwise, Approach B is probably better. Furthermore, Approach A makes the name w visible in a larger scope (the one containing the loop) than Approach B, something that's contrary to program comprehensibility and maintainability. As a result, unless you know that (1) assignment is less expensive than a constructor-destructor pair and (2) you're dealing with a performance-sensitive part of your code, you should default to using Approach B.

- const_cast is typically used to cast away the constness of objects. It is the only C++-style cast that can do this.
- dynamic_cast is primarily used to perform "safe downcasting," i.e., to determine whether an object is of a particular type in an inheritance hierarchy. It is the only cast that cannot be performed using the old-style syntax. It is also the only cast that may have a significant runtime cost. (I'll provide details on this a bit later.)
- reinterpret_cast is intended for low-level casts that yield implementation-dependent (i.e., unportable) results, e.g., casting a pointer to an int. Such casts should be rare outside low-level code. I use it only once in this book, and that's only when discussing how you might write a debugging allocator for raw memory (see Item 50).
- static_cast can be used to force implicit conversions (e.g., non-const object to const object (as in Item 3), int to double, etc.). It can also be used to perform the reverse of many such conversions (e.g., void* pointers to typed pointers, pointer-to-base to pointer-to-derived), though it cannot cast from const to non-const objects. (Only const_cast can do that.)

The old-style casts continue to be legal, but the new forms are prefer

```
class Base { ... };
class Derived: public Base { ... };
Derived d;
Base *pb = &d;                    // implicitly convert Derived* ⇒ Base*
```

Here we're just creating a base class pointer to a derived class object, but sometimes, the two pointer values will not be the same. When that's the case, an offset is applied *at runtime* to the Derived* pointer to get the correct Base* pointer value.

This last example demonstrates that a single object (e.g., an object of type Derived) might have more than one address (e.g., its address when pointed to by a Base* pointer and its address when pointed to by a Derived* pointer). That can't happen in C. It can't happen in Java. It can't happen in C#. It *does* happen in C++. In fact, when multiple

Good C++ uses very few casts, but it's generally not practical to get rid of all of them. The cast from int to double on page 118, for example, is a reasonable use of a cast, though it's not strictly necessary. (The code could be rewritten to declare a new variable of type double that's initialized with x's value.) Like most suspicious constructs, casts should be isolated as much as possible, typically hidden inside functions whose interfaces shield callers from the grubby work being done inside.

Things to Remember

- Avoid casts whenever practical, especially dynamic_casts in performance-sensitive code. If a design requires casting, try to develop a cast-free alternative.
- When casting is necessary, try to hide it inside a function. Clients can then call the function instead of putting casts in their own code.
- Prefer C++-style casts to old-style casts. They are easier to see, and they are more specific about what they do.

Item 28: Avoid returning "handles" to object internals

ber. But rec is supposed to be const!

This immediately leads to two lessons. First, a data member is only as encapsulated as the most accessible function returning a reference to it. In this case, though ulhc and lrhc are supposed to be private to their Rectangle, they're effectively public, because the public functions

upperLeft and lowerRight return references to them. Second, if a const member function returns a reference to data associated with an object that is stored outside the object itself, the caller of the function can modify that data. (This is just a fallout of the limitations of bitwise constness — see Item 3.)

Everything we've done has involved member functions returning references, but if they returned pointers or iterators, the same problems would exist for the same reasons. References, pointers, and iterators

We generally think of an object's "internals" as its data members, but member functions not accessible to the general public (i.e., that are protected or private) are part of an object's internals, too. As such, it's important not to return handles to them. This means you should never have a member function return a pointer to a less accessible member function. If you do, the effective access level will be that of the more accessible function, because clients will be able to get a pointer to the less accessible function, then call that function through the pointer.

Functions that return pointers to member functions are uncommon, however, so let's turn our attention back to the Rectangle class and its upperLeft and lowerRight member functions. Both of the problems we've identified for those functions can be eliminated by simply applying const to their return types:

```
class Rectangle {
public:
  ...
  const Point& upperLeft() const { return pData->ulhc; }
  const Point& lowerRight() const { return pData->lrhc; }
  ...
};
```

allows you to pluck individual elements out of strings and vectors, and these operator[]s work by returning references to the data in the containers (see Item 3) — data that is destroyed when the containers themselves are. Still, such functions are the exception, not the rule.

Things to Remember

- Avoid returning handles (references, pointers, or iterators) to object internals. Not returning handles increases encapsulation, helps const member functions act const, and minimizes the creation of dangling handles.

Item 29: Strive for exception-safe code.

Exception safety is sort of like pregnancy...but hold that thought for a

On the other hand, if an inline function body is *very* short, the code generated for the function body may be smaller than the code generated for a function call. If that is the case, inlining the function may actually lead to *smaller* object code and a higher instruction cache hit rate!

Bear in mind that inline is a *request* to compilers, not a command. The request can be given implicitly or explicitly. The implicit way is to define a function inside a class definition:

```
class Person {
public:
  ...
  int age() const { return theAge; }      // an implicit inline request: age is
  ...                                     // defined in a class definition

private:
  int theAge;
};
```

Such functions are usually member functions, but Item 46 explains that friend functions can also be defined inside classes. When they are, they're also implicitly declared inline.

The explicit way to declare an inline function is to precede its defini-

```
private:
   int theAge;
};
```

Such functions are usually member functions, but Item 46 explains that friend functions can also be defined inside classes. When they are, they're also implicitly declared inline.

The explicit way to declare an inline function is to precede its definition with the inline keyword. For example, this is how the standard max template (from <algorithm>) is often implemented:

```
template<typename T>                          // an explicit inline
inline const T& std::max(const T& a, const T& b)   // request: std::max is
{ return a < b ? b : a; }                     // preceded by "inline"
```

The fact that max is a template brings up the observation that both inline functions and templates are typically defined in header files. This leads some programmers to conclude that function templates must be inline. This conclusion is both invalid and potentially harmful, so it's worth looking into it a bit.

Inline functions must typically be in header files, because most build environments do inlining during compilation. In order to replace a

Templates are typically in header files, because compilers need to know what a template looks like in order to instantiate it when it's used. (Again, this is not universal. Some build environments perform template instantiation during linking. However, compile-time instantiation is more common.)

Template instantiation is independent of inlining. If you're writing a template and you believe that all the functions instantiated from the template should be inlined, declare the template inline; that's what's done with the std::max implementation above. But if you're writing a template for functions that you have no reason to want inlined, avoid declaring the template inline (either explicitly or implicitly). Inlining has costs, and you don't want to incur them without forethought. We've already mentioned how inlining can cause code bloat (a particularly important consideration for template authors — see Item 44), but there are other costs, too, which we'll discuss in a moment.

Before we do that, let's finish the observation that inline is a request that compilers may ignore. Most compilers refuse to inline functions they deem too complicated (e.g., those that contain loops or are recursive), and all but the most trivial calls to virtual functions defy inlining. This latter observation shouldn't be a surprise. virtual means "wait

```
  Derived() {}                          // Derived's ctor is empty — OK is it?
  ...

private:
  std::string dm1, dm2, dm3;            // derived members 1–3
};
```

This constructor looks like an excellent candidate for inlining, since it contains no code. But looks can be deceiving.

C++ makes various guarantees about things that happen when objects are created and destroyed. When you use new, for example, your dynamically created objects are automatically initialized by their constructors, and when you use delete, the corresponding destructors are invoked. When you create an object, each base class of and each data member in that object is automatically constructed, and the reverse process regarding destruction automatically occurs when an object is destroyed. If an exception is thrown during construction of an object, any parts of the object that have already been fully constructed are automatically destroyed. In all these scenarios, C++ says *what* must happen, but it doesn't say *how*. That's up to compiler implementers, but it should be clear that those things don't happen by themselves. There has to be some code in your program to make those things happen, and that code — the code written by compilers

```
Derived::Derived()                          // conceptual implementation of
{                                           // "empty" Derived ctor

   Base::Base();                            // initialize Base part

   try { dm1.std::string::string(); }       // try to construct dm1
   catch (...) {                            // if it throws,
      Base::~Base();                        // destroy base class part and
      throw;                                // propagate the exception
   }

   try { dm2.std::string::string(); }       // try to construct dm2
   catch(...) {                             // if it throws,
      dm1.std::string::~string();           // destroy dm1,
      Base::~Base();                        // destroy base class part, and
      throw;                                // propagate the exception
   }

   try { dm3.std::string::string(); }       // construct dm3
   catch(...) {                             // if it throws,
      dm2.std::string::~string();           // destroy dm2,
      dm1.std::string::~string();           // destroy dm1,
      Base::~Base();                        // destroy base class part, and
      throw;                                // propagate the exception
   }
}
```

```
        throw;                                  // propagate the exception
      }
    }
```

This code is unrepresentative of what real compilers emit, because real compilers deal with exceptions in more sophisticated ways. Still, this accurately reflects the behavior that Derived's "empty" constructor must offer. No matter how sophisticated a compiler's exception implementation, Derived's constructor must at least call constructors for its data members and base class, and those calls (which might themselves be inlined) could affect its attractiveness for inlining.

The same reasoning applies to the Base constructor, so if it's inlined, all the code inserted into it is also inserted into the Derived constructor (via the Derived constructor's call to the Base constructor). And if the string constructor also happens to be inlined, the Derived constructor will gain *five copies* of that function's code, one for each of the five strings in a Derived object (the two it inherits plus the three it declares itself). Perhaps now it's clear why it's not a no-brain decision whether to inline Derived's constructor. Similar considerations apply to Derived's destructor, which, one way or another, must see to it that all the objects initialized by Derived's constructor are properly destroyed.

compiler knows how big an int is. When compilers see the definition for p, they know they have to allocate enough space for a Person, but how are they supposed to know how big a Person object is? The only way they can get that information is to consult the class definition, but if it were legal for a class definition to omit the implementation details, how would compilers know how much space to allocate?

This question fails to arise in languages like Smalltalk and Java, because, when an object is defined in such languages, compilers allocate only enough space for a *pointer* to an object. That is, they handle the code above as if it had been written like this:

```
int main()
{
  int x;                    // define an int
  Person *p;                // define a pointer to a Person
  ...
}
```

This, of course, is legal C++, so you can play the "hide the object implementation behind a pointer" game yourself. One way to do that

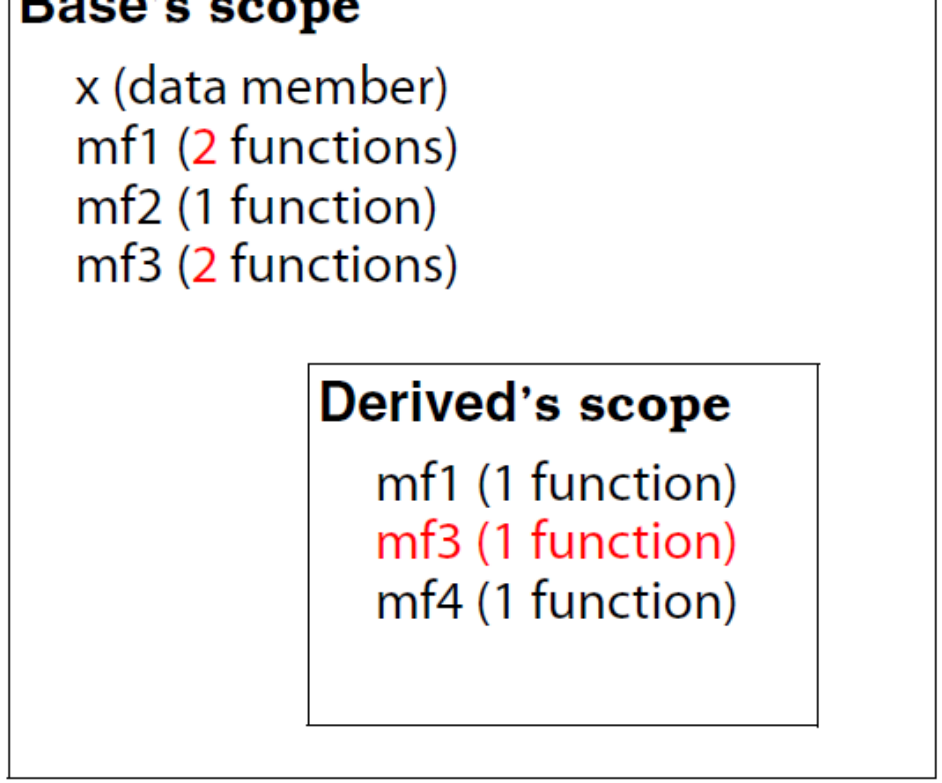
tions that specify the interface.

Interface classes are akin to Java's and .NET's Interfaces, but C++ doesn't impose the restrictions on Interface classes that Java and .NET impose on Interfaces. Neither Java nor .NET allow data members or function implementations in Interfaces, for example, but C++ forbids neither of these things. C++'s greater flexibility can be useful. As Item 36 explains, the implementation of non-virtual functions should be the same for all classes in a hierarchy, so it makes sense to implement such functions as part of the Interface class that declares them.

An Interface class for Person could look like this:

```
class Person {
public:
   virtual ~Person();

   virtual std::string name() const = 0;
   virtual std::string birthDate() const = 0;
   virtual std::string address() const = 0;
   ...
};
```

```
    int x;

public:
    virtual void mf1() = 0;
    virtual void mf1(int);

    virtual void mf2();

    void mf3();
    void mf3(double);
    ...
};

class Derived: public Base {
public:
    virtual void mf1();
    void mf3();
    void mf4();
    ...
};
```



This code leads to behavior that surprises every C++ programmer the first time they encounter it. The scope-based name hiding rule hasn't changed, so *all* functions named mf1 and mf3 in the base class are hidden by the functions named mf1 and mf3 in the derived class. From the perspective of name lookup, Base::mf1 and Base::mf3 are no longer inherited by Derived!

```
Derived d;
int x;

...

d.mf1();          // fine, calls Derived::mf1
d.mf1(x);         // error! Derived::mf1 hides Base::mf1
```



```
d.mf2();          // fine, calls Base::mf2

d.mf3();          // fine, calls Derived::mf3
d.mf3(x);         // error! Derived::mf3 hides Base::mf3
```

As you can see, this applies even though the functions in the base and derived classes take different parameter types, and it also applies regardless of whether the functions are virtual or non-virtual. In the same way that, at the beginning of this Item, the double x in the function someFunc hides the int x at global scope, here the function mf3 in Derived hides a Base function named mf3 that has a different type.

The rationale behind this behavior is that it prevents you from accidentally inheriting overloads from distant base classes when you create a new derived class in a library or application framework. Unfortunately, you typically *want* to inherit the overloads. In fact, if you're using public inheritance and you don't inherit the overloads, you're violating the is-a relationship between base and derived classes that Item 32 explains is fundamental to public inheritance. That being the case, you'll almost always want to override C++'s default hiding of inherited names.

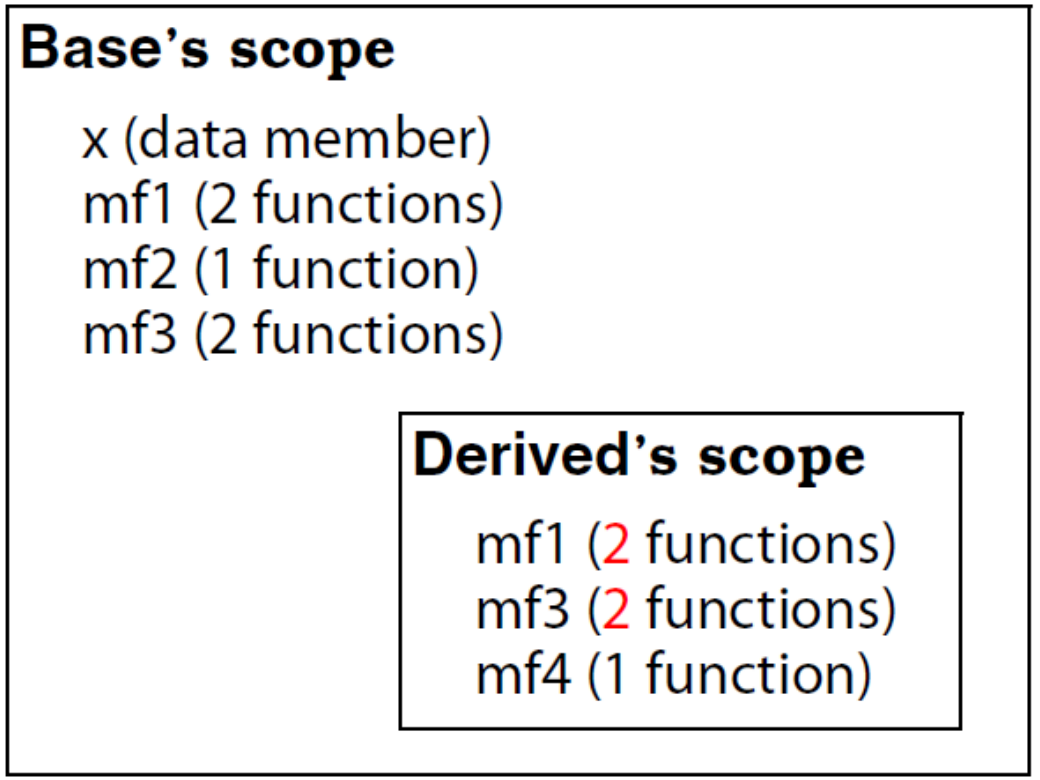
You do it with *using declarations*:

```
class Base {
private:
```

Base's scope

You do it with *using declarations*:

```
class Base {
private:
   int x;

public:
   virtual void mf1() = 0;
   virtual void mf1(int);

   virtual void mf2();

   void mf3();
   void mf3(double);
   ...
};

class Derived: public Base {
public:
   using Base::mf1;          // make all things in Base named mf1 and mf3
   using Base::mf3;          // visible (and public) in Derived's scope

   virtual void mf1();
   void mf3();
   void mf4();
```



```
Derived d;
int x;

...

d.mf1();          // still fine, still calls Derived::mf1
d.mf1(x);         // now okay, calls Base::mf1

d.mf2();          // still fine, still calls Base::mf2

d.mf3();          // fine, calls Derived::mf3
d.mf3(x);         // now okay, calls Base::mf3 (The int x is
                  // implicitly converted to a double so that
                  // the call to Base::mf3 is valid.)
```

This means that if you inherit from a base class with overloaded functions and you want to redefine or override only some of them, you need to include a using declaration for each name you'd otherwise be hiding. If you don't, some of the names you'd like to inherit will be hidden.

It's conceivable that you sometimes won't want to inherit all the functions from your base classes. Under public inheritance, this should never be the case, because, again, it violates public inheritance's is-a relationship between base and derived classes. (That's why the using declarations above are in the public part of the derived class: names that are public in a base class should also be public in a publicly derived class.) Under private inheritance (see Item 39), however, it can make sense. For example, suppose Derived privately inherits from Base, and the only version of mf1 that Derived wants to inherit is the one taking no parameters. A using declaration won't do the trick here, because a using declaration makes *all* inherited functions with a given name visible in the derived class. No, this is a case for a different technique, namely, a simple forwarding function:

```
class Base {
public:
  virtual void mf1() = 0;
  virtual void mf1(int);

  ...                                   // as before
};
```

compilers that (incorrectly) don't support using declarations to import inherited names into the scope of a derived class.

That's the whole story on inheritance and name hiding, but when inheritance is combined with templates, an entirely different form of the "inherited names are hidden" issue arises. For all the angle-bracket-demarcated details, see Item 43.

Things to Remember

- Names in derived classes hide names in base classes. Under public inheritance, this is never desirable.
- To make hidden names visible again, employ using declarations or forwarding functions.

Item 34: Differentiate between inheritance of interface and inheritance of implementation.

The seemingly straightforward notion of (public) inheritance turns out, upon closer examination, to be composed of two separable parts:

s combined with templates, an entirely differe
ed names are hidden" issue arises. For all
arcated details, see Item 43.

emember

derived classes hide names in base classes. Un
e, this is never desirable.

idden names visible again, employ using decla
functions.

Differentiate between inheritance of interface and inheritance of impleme

gly straightforward notion of (public) inherita
oser examination, to be composed of two separ

class ModelB: public Airplane { ... };

To express that all planes have to support a fly function, and in recognition of the fact that different models of plane could, in principle, require different implementations for fly, Airplane::fly is declared virtual. However, in order to avoid writing identical code in the ModelA and ModelB classes, the default flying behavior is provided as the body of Airplane::fly, which both ModelA and ModelB inherit.

This is a classic object-oriented design. Two classes share a common feature (the way they implement fly), so the common feature is moved into a base class, and the feature is inherited by the two classes. This design makes common features explicit, avoids code duplication, facilitates future enhancements, and eases long-term maintenance — all the things for which object-oriented technology is so highly touted. XYZ Airlines should be proud.

Now suppose that XYZ, its fortunes on the rise, decides to acquire a new type of airplane, the Model C. The Model C differs in some ways from the Model A and the Model B. In particular, it is flown differently.

XYZ's programmers add the class for Model C to the hierarchy, but in their haste to get the new model into service, they forget to redefine the fly function:

Now suppose that XYZ, its fortunes on the rise, decides to acquire a new type of airplane, the Model C. The Model C differs in some ways from the Model A and the Model B. In particular, it is flown differently.

XYZ's programmers add the class for Model C to the hierarchy, but in their haste to get the new model into service, they forget to redefine the fly function:

```
class ModelC: public Airplane {
  ...                                   // no fly function is declared
};
```

In their code, then, they have something akin to the following:

```
Airport PDX(...);                       // PDX is the airport near my home
Airplane *pa = new ModelC;
...
pa->fly(PDX);                           // calls Airplane::fly!
```

This is a disaster: an attempt is being made to fly a ModelC object as if it were a ModelA or a ModelB. That's not the kind of behavior that inspires confidence in the traveling public.

The problem here is not that Airplane::fly has default behavior, but that ModelC was allowed to inherit that behavior without explicitly saying that it wanted to. Fortunately, it's easy to offer default behavior to derived classes but not give it to them unless they ask for it. The trick is to sever the connection between the *interface* of the virtual function and its default *implementation*. Here's one way to do it:

```
class Airplane {
public:
   virtual void fly(const Airport& destination) = 0;

   ...

protected:
   void defaultFly(const Airport& destination);
};

void Airplane::defaultFly(const Airport& destination)
{
   default code for flying an airplane to the given destination
}
```

Notice how Airplane::fly has been turned into a *pure* virtual function. That provides the interface for flying. The default implementation is also present in the Airplane class, but now it's in the form of an independent function, defaultFly. Classes like ModelA and ModelB that want to use the default behavior simply make an inline call to defaultFly inside their body of fly (but see Item 30 for information on the interaction of inlining and virtual functions):

```
class ModelA: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }

   ...

};

class ModelB: public Airplane {
public:
   virtual void fly(const Airport& destination)
   { defaultFly(destination); }

   ...

};
```

For the ModelC class, there is no possibility of accidentally inheriting the incorrect implementation of fly, because the pure virtual in Airplane forces ModelC to provide its own version of fly.

```
class ModelC: public Airplane {
public:
  virtual void fly(const Airport& destination);

  ...

};

void ModelC::fly(const Airport& destination)
{
  code for flying a ModelC airplane to the given destination
}
```

When a member function is non-virtual, it's not supposed to behave differently in derived classes. In fact, a non-virtual member function specifies an *invariant over specialization*, because it identifies behavior that is not supposed to change, no matter how specialized a derived class becomes. As such,

- The purpose of declaring a non-virtual function is to have derived classes inherit a function *interface as well as a mandatory implementation*.

You can think of the declaration for Shape::objectID as saying, "Every Shape object has a function that yields an object identifier, and that object identifier is always computed the same way. That way is determined by the definition of Shape::objectID, and no derived class should try to change how it's done." Because a non-virtual function identifies an *invariant* over specialization, it should never be redefined in a derived class, a point that is discussed in detail in Item 36.

The differences in declarations for pure virtual, simple virtual, and non-virtual functions allow you to specify with precision what you want derived classes to inherit: interface only, interface and a default implementation, or interface and a mandatory implementation, respectively. Because these different types of declarations mean fundamen-

We'll begin with an interesting school of thought that argues that virtual functions should almost always be private. Adherents to this school would suggest that a better design would retain healthValue as a public member function but make it non-virtual and have it call a private virtual function to do the real work, say, doHealthValue:

```
class GameCharacter {
public:
  int healthValue() const                  // derived classes do not redefine
  {                                        // this — see Item 36

    ...                                    // do "before" stuff — see below

    int retVal = doHealthValue();          // do the real work

    ...                                    // do "after" stuff — see below

    return retVal;
  }
  ...

private:
  virtual int doHealthValue() const        // derived classes may redefine this
  {
    ...                                    // default algorithm for calculating
  }                                        // character's health
};
```

make it easier to see what is going on. The designs I'm describing are independent of inlining decisions, so don't think it's meaningful that the member functions are defined inside classes. It's not.

This basic design — having clients call private virtual functions indirectly through public non-virtual member functions — is known as the *non-virtual interface (NVI) idiom*. It's a particular manifestation of the more general design pattern called Template Method (a pattern that, unfortunately, has nothing to do with C++ templates). I call the non-virtual function (e.g., healthValue) the virtual function's *wrapper*.

Virtual functions, on the other hand, are dynamically bound (again, see Item 37), so they don't suffer from this problem. If mf were a virtual function, a call to mf through either pB or pD would result in an invocation of D::mf, because what pB and pD *really* point to is an object of type D.

If you are writing class D and you redefine a non-virtual function mf that you inherit from class B, D objects will likely exhibit inconsistent behavior. In particular, any given D object may act like either a B or a D when mf is called, and the determining factor will have nothing to do with the object itself, but with the declared type of the pointer that points to it. References exhibit the same baffling behavior as do pointers.

But that's just a pragmatic argument. What you really want, I know, is some kind of theoretical justification for not redefining inherited non-virtual functions. I am pleased to oblige.

Item 32 explains that public inheritance means is-a, and Item 34 describes why declaring a non-virtual function in a class establishes an invariant over specialization for that class. If you apply these observations to the classes B and D and to the non-virtual member

would invariably redefine an inherited non-virtual function. the base class's destructor. This would be true even for derived classes that declare no destructor, because, as Item 5 explains, the destructor is one of the member functions that compilers generate for you if you don't declare one yourself. In essence, Item 7 is nothing more than a special case of this Item, though it's important enough to merit calling out on its own.

Things to Remember

- Never redefine an inherited non-virtual function.

Item 37: Never redefine a function's inherited default parameter value.

Let's simplify this discussion right from the start. There are only two kinds of functions you can inherit: virtual and non-virtual. However, it's always a mistake to redefine an inherited non-virtual function (see Item 36), so we can safely limit our discussion here to the situation in which you inherit a *virtual* function with a default parameter value.

That being the case, the justification for this Item becomes quite straightforward: virtual functions are dynamically bound, but default

Item 37: Never redefine a function's inherited default parameter value.

Let's simplify this discussion right from the start. There are only two kinds of functions you can inherit: virtual and non-virtual. However, it's always a mistake to redefine an inherited non-virtual function (see Item 36), so we can safely limit our discussion here to the situation in which you inherit a *virtual* function with a default parameter value.

That being the case, the justification for this Item becomes quite straightforward: virtual functions are dynamically bound, but default parameter values are statically bound.

What's that? You say the difference between static and dynamic binding has slipped your already overburdened mind? (For the record, static binding is also known as *early binding*, and dynamic binding is also known as *late binding*.) Let's review, then.

An object's *static type* is the type you declare it to have in the program text. Consider this class hierarchy:

```
// a class for geometric shapes
class Shape {
public:
  enum ShapeColor { Red, Green, Blue };

  // all shapes must offer a function to draw themselves
  virtual void draw(ShapeColor color = Red) const = 0;
```

```
pc->draw(Shape::Red);          // calls Circle::draw(Shape::Red)
pr->draw(Shape::Red);          // calls Rectangle::draw(Shape::Red)
```

This is all old hat, I know; you surely understand virtual functions. The twist comes in when you consider virtual functions with default parameter values, because, as I said above, virtual functions are dynamically bound, but default parameters are statically bound. That means you may end up invoking a virtual function defined in a *derived class* but using a default parameter value from a *base class*:

```
pr->draw();                    // calls Rectangle::draw(Shape::Red)!
```

In this case, pr's dynamic type is Rectangle*, so the Rectangle virtual function is called, just as you would expect. In Rectangle::draw, the default parameter value is Green. Because pr's static type is Shape*, however, the default parameter value for this function call is taken from the Shape class, not the Rectangle class! The result is a call consisting of a strange and almost certainly unanticipated combination of the declarations for draw in both the Shape and Rectangle classes.

The fact that ps, pc, and pr are pointers is of no consequence in this matter. Were they references, the problem would persist. The only important things are that draw is a virtual function, and one of its default parameter values is redefined in a derived class.

classes (see Item 36), this design makes clear that the default value for draw's color parameter should always be Red.

Things to Remember

- Never redefine an inherited default parameter value, because default parameter values are statically bound, while virtual functions — the only functions you should be redefining — are dynamically bound.

Item 38: Model "has-a" or "is-implemented-in-terms-of" through composition.

```
class Student: private Person { ... };   // inheritance is now private

void eat(const Person& p);               // anyone can eat

void study(const Student& s);            // only students study

Person p;                                // p is a Person
Student s;                               // s is a Student

eat(p);                                  // fine, p is a Person

eat(s);                                  // error! a Student isn't a Person
```

Clearly, private inheritance doesn't mean is-a. What does it mean then?

"Whoa!" you say. "Before we get to the meaning, let's cover the behavior. How does private inheritance behave?" Well, the first rule governing private inheritance you've just seen in action: in contrast to public inheritance, compilers will generally *not* convert a derived class object (such as Student) into a base class object (such as Person) if the inheritance relationship between the classes is private. That's why the call to eat fails for the object s. The second rule is that members inherited from a private base class become private members of the derived class, even if they were protected or public in the base class.

So much for behavior. That brings us to meaning. Private inheritance

tions adds a vptr to each object — see Item 7); and no virtual base classes (because such base classes also incur a size overhead — see Item 40). Conceptually, objects of such *empty classes* should use no space, because there is no per-object data to be stored. However, there are technical reasons for C++ decreeing that freestanding objects must have non-zero size, so if you do this,

```
class Empty {};                          // has no data, so objects should
                                         // use no memory

class HoldsAnInt {                       // should need only space for an int
private:
   int x;
   Empty e;                              // should require no memory
};
```

you'll find that sizeof(HoldsAnInt) > sizeof(int); an Empty data member requires memory. With most compilers, sizeof(Empty) is 1, because

C++'s edict against zero-size freestanding objects is typically satisfied by the silent insertion of a char into "empty" objects. However, alignment requirements (see Item 50) may cause compilers to add padding to classes like HoldsAnInt, so it's likely that HoldsAnInt objects wouldn't gain just the size of a char, they would actually enlarge enough to hold a second int. (On all the compilers I tested, that's exactly what happened.)

But perhaps you've noticed that I've been careful to say that "freestanding" objects mustn't have zero size. This constraint doesn't apply to base class parts of derived class objects, because they're not freestanding. If you *inherit* from Empty instead of containing an object of that type,

```
class HoldsAnInt: private Empty {
private:
   int x;
};
```

But perhaps you've noticed that I've been careful to say that "free-standing" objects mustn't have zero size. This constraint doesn't apply to base class parts of derived class objects, because they're not free-standing. If you *inherit* from Empty instead of containing an object of that type,

```
class HoldsAnInt: private Empty {
private:
   int x;
};
```

you're almost sure to find that sizeof(HoldsAnInt) == sizeof(int). This is known as the *empty base optimization* (EBO), and it's implemented by all the compilers I tested. If you're a library developer whose clients care about space, the EBO is worth knowing about. Also worth knowing is that the EBO is generally viable only under single inheritance. The rules governing C++ object layout generally mean that the EBO can't be applied to derived classes that have more than one base.

In practice, "empty" classes aren't truly empty. Though they never have non-static data members, they often contain typedefs, enums, static data members, or non-virtual functions. The STL has many technically empty classes that contain useful members (usually type-

seen that a mixture of public inheritance and containment can often yield the behavior you want, albeit with greater design complexity. Using private inheritance *judiciously* means employing it when, having considered all the alternatives, it's the best way to express the relationship between two classes in your software.

Things to Remember

- Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions.
- Unlike composition, private inheritance can enable the empty base optimization. This can be important for library developers who strive to minimize object sizes.

Item 40: Use multiple inheritance judiciously.

```
class BorrowableItem {                  // something a library lets you borrow
public:
   void checkOut();                     // check the item out from the library
   ...
};

class ElectronicGadget {
private:
   bool checkOut() const;               // perform self-test, return whether
   ...                                  // test succeeds
};

class MP3Player:                        // note MI here
   public BorrowableItem,               // (some libraries loan MP3 players)
   public ElectronicGadget
{ ... };                                // class definition is unimportant

MP3Player mp;

mp.checkOut();                          // ambiguous! which checkOut?
```

Note that in this example, the call to checkOut is ambiguous, even though only one of the two functions is accessible. (checkOut is public in BorrowableItem but private in ElectronicGadget.) That's in accord with the C++ rules for resolving calls to overloaded functions: before seeing whether a function is accessible, C++ first identifies the function that's the best match for the call. It checks accessibility only after finding the best-match function. In this case, the name checkOut is ambiguous during name lookup, so neither function overload resolution nor best match determination takes place. The accessibility of ElectronicGadget::checkOut is therefore never examined.

To resolve the ambiguity, you must specify which base class's function to call:

```
mp.BorrowableItem::checkOut();          // ah, that checkOut...
```

class all the time, because it's easier to type. Others (including me) prefer typename, because it suggests that the parameter need not be a class type. A few developers employ typename when any type is allowed and reserve class for when only user-defined types are acceptable. But from C++'s point of view, class and typename mean exactly the same thing when declaring a template parameter.

C++ doesn't always view class and typename as equivalent, however. Sometimes you must use typename. To understand when, we have to talk about two kinds of names you can refer to in a template.

Suppose we have a template for a function that takes an STL-compatible container holding objects that can be assigned to ints. Further

```
template<typename C>                          // print 2nd element in
void print2nd(const C& container)             // container;
{                                             // this is not valid C++!
  if (container.size() >= 2) {
    C::const_iterator iter(container.begin()); // get iterator to 1st element
    ++iter;                                   // move iter to 2nd element
    int value = *iter;                        // copy that element to an int
    std::cout << value;                       // print the int
  }
}
```

I've highlighted the two local variables in this function, iter and value. The type of iter is C::const_iterator, a type that depends on the template parameter C. Names in a template that are dependent on a template parameter are called *dependent names*. When a dependent name is nested inside a class, I call it a *nested dependent name*. C::const_iterator is a nested dependent name. In fact, it's a *nested dependent type name*, i.e., a nested dependent name that refers to a type.

The other local variable in print2nd, value, has type int. int is a name that does not depend on any template parameter. Such names are known as *non-dependent names*, (I have no idea why they're not called independent names. If, like me, you find the term "non-dependent" an abomination, you have my sympathies, but "non-dependent" is the term for these kinds of names, so, like me, roll your eyes and resign yourself to it.)

able? In that case, the code above wouldn't declare a local variable, it would be a multiplication of C::const_iterator by x! Sure, that sounds crazy, but it's *possible*, and people who write C++ parsers have to worry about all possible inputs, even the crazy ones.

Until C is known, there's no way to know whether C::const_iterator is a type or isn't, and when the template print2nd is parsed, C isn't known. C++ has a rule to resolve this ambiguity: if the parser encounters a nested dependent name in a template, it assumes that the name is *not* a type unless you tell it otherwise. By default, nested dependent names are *not* types. (There is an exception to this rule that I'll get to in a moment.)

With that in mind, look again at the beginning of print2nd:

```
template<typename C>
void print2nd(const C& container)
{
  if (container.size() >= 2) {
```

in a moment.)

With that in mind, look again at the beginning of print2nd:

```
template<typename C>
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    C::const_iterator iter(container.begin());      // this name is assumed to
    ...                                             // not be a type
```

Now it should be clear why this isn't valid C++. The declaration of iter makes sense only if C::const_iterator is a type, but we haven't told C++ that it is, and C++ assumes that it's not. To rectify the situation, we have to tell C++ that C::const_iterator is a type. We do that by putting typename immediately in front of it:

```
template<typename C>                                // this is valid C++
void print2nd(const C& container)
{
  if (container.size() >= 2) {
    typename C::const_iterator iter(container.begin());

    ...

  }
}
```

The general rule is simple: anytime you refer to a nested dependent

zMsgSender.sendClearMsg(msgData); // error! won't compile

the call to sendClearMsg won't compile, because at this point, compilers know that the base class is the template specialization MsgSender<CompanyZ>, and they know that class doesn't offer the sendClear function that sendClearMsg is trying to call.

Fundamentally, the issue is whether compilers will diagnose invalid references to base class members sooner (when derived class template definitions are parsed) or later (when those templates are instantiated with specific template arguments). C++'s policy is to prefer early diagnoses, and that's why it assumes it knows nothing about the contents of base classes when those classes are instantiated from templates.

Things to Remember

- In derived class templates, refer to names in base class templates via a "this->" prefix, via using declarations, or via an explicit base class qualification.

Item 44: Factor parameter-independent code out of templates.

Templates are a wonderful way to save time and avoid code replication. Instead of typing 20 similar classes, each with 15 member func-

plates that do the same thing.

Things to Remember

- Templates generate multiple classes and multiple functions, so any template code not dependent on a template parameter causes bloat.
- Bloat due to non-type template parameters can often be eliminated by replacing template parameters with function parameters or class data members.
- Bloat due to type parameters can be reduced by sharing implementations for instantiation types with identical binary representations.

```
class Top { ... };
class Middle: public Top { ... };
class Bottom: public Middle { ... };
Top *pt1 = new Middle;                  // convert Middle* ⇒ Top*
Top *pt2 = new Bottom;                  // convert Bottom* ⇒ Top*
const Top *pct2 = pt1;                  // convert Top* ⇒ const Top*
```

Emulating such conversions in user-defined smart pointer classes is tricky. We'd need the following code to compile:

```
template<typename T>
class SmartPtr {
public:                                 // smart pointers are typically
  explicit SmartPtr(T *realPtr);        // initialized by built-in pointers
  ...
};

SmartPtr<Top> pt1 =                     // convert SmartPtr<Middle> ⇒
  SmartPtr<Middle>(new Middle);         //    SmartPtr<Top>

SmartPtr<Top> pt2 =                     // convert SmartPtr<Bottom> ⇒
  SmartPtr<Bottom>(new Bottom);         //    SmartPtr<Top>

SmartPtr<const Top> pct2 = pt1;         // convert SmartPtr<Top> ⇒
                                        //    SmartPtr<const Top>
```

// SmartPtr<const Top>

There is no inherent relationship among different instantiations of the same template, so compilers view SmartPtr<Middle> and SmartPtr<Top> as completely different classes, no more closely related than, say, vector<float> and Widget. To get the conversions among SmartPtr classes that we want, we have to program them explicitly.

In the smart pointer sample code above, each statement creates a new smart pointer object, so for now we'll focus on how to write smart pointer constructors that behave the way we want. A key observation is that there is no way to write out all the constructors we need. In the hierarchy above, we can construct a SmartPtr<Top> from a SmartPtr<Middle> or a SmartPtr<Bottom>, but if the hierarchy is extended in the

dle> or a SmartPtr<Bottom>, but if the hierarchy is extended in the future, SmartPtr<Top> objects will have to be constructible from other smart pointer types. For example, if we later add

```
class BelowBottom: public Bottom { ... };
```

we'll need to support the creation of SmartPtr<Top> objects from SmartPtr<BelowBottom> objects, and we certainly won't want to have to modify the SmartPtr template to do it.

In principle, the number of constructors we need is unlimited. Since a template can be instantiated to generate an unlimited number of functions, it seems that we don't need a constructor *function* for SmartPtr, we need a constructor *template*. Such templates are examples of *member function templates* (often just known as *member templates*) — templates that generate member functions of a class:

```
template<typename T>
class SmartPtr {
public:
  template<typename U>                          // member template
  SmartPtr(const SmartPtr<U>& other);           // for a "generalized
  ...                                           // copy constructor"
};
```

This says that for every type T and every type U, a SmartPtr<T> can be created from a SmartPtr<U>, because SmartPtr<T> has a constructor

```
public:
  template<typename U>                          // member template
  SmartPtr(const SmartPtr<U>& other);           // for a "generalized
  ...                                           // copy constructor"
};
```

This says that for every type T and every type U, a SmartPtr<T> can be created from a SmartPtr<U>, because SmartPtr<T> has a constructor that takes a SmartPtr<U> parameter. Constructors like this — ones that create one object from another object whose type is a different instantiation of the same template (e.g., create a SmartPtr<T> from a SmartPtr<U>) — are sometimes known as *generalized copy constructors*.

The generalized copy constructor above is not declared explicit. That's deliberate. Type conversions among built-in pointer types (e.g., from derived to base class pointers) are implicit and require no cast, so it's reasonable for smart pointers to emulate that behavior. Omitting explicit on the templatized constructor does just that.

As declared, the generalized copy constructor for SmartPtr offers more than we want. Yes, we want to be able to create a SmartPtr<Top> from a SmartPtr<Bottom>, but we don't want to be able to create a SmartPtr<Bottom> from a SmartPtr<Top>, as that's contrary to the meaning of public inheritance (see Item 32). We also don't want to be

```
template<class T> class shared_ptr {
public:
  shared_ptr(shared_ptr const& r);                  // copy constructor
  template<class Y>                                 // generalized
    shared_ptr(shared_ptr<Y> const& r);             // copy constructor
  shared_ptr& operator=(shared_ptr const& r);       // copy assignment
  template<class Y>                                 // generalized
    shared_ptr& operator=(shared_ptr<Y> const& r);  // copy assignment
  ...
};
```


Things to Remember

- Use member function templates to generate functions that accept all compatible types.
- If you declare member templates for generalized copy construction or generalized assignment, you'll still need to declare the normal copy constructor and copy assignment operator, too.

// move iter backward

Conceptually, advance just does iter += d, but advance can't be implemented that way, because only random access iterators support the += operation. Less powerful iterator types have to implement advance by iteratively applying ++ or -- d times.

Um, you don't remember your STL iterator categories? No problem, we'll do a mini-review. There are five categories of iterators, corresponding to the operations they support. *Input iterators* can move only forward, can move only one step at a time, can only read what they point to, and can read what they're pointing to only once. They're modeled on the read pointer into an input file; the C++ library's istream_iterators are representative of this category. *Output iterators* are analogous, but for output: they move only forward, move only one step at a time, can only write what they point to, and can write it only once. They're modeled on the write pointer into an output file; ostream_iterators epitomize this category. These are the two least powerful iterator categories. Because input and output iterators can move only forward and can read or write what they point to at most once, they are suitable only for one-pass algorithms.

A more powerful iterator category consists of *forward iterators*. Such iterators can do everything input and output iterators can do, plus they can read or write what they point to more than once. This makes them viable for multi-pass algorithms. The STL offers no singly linked list, but some libraries offer one (usually called slist), and iterators into such containers are forward iterators. Iterators into TR1's hashed

they are suitable only for one-pass algorithms.

A more powerful iterator category consists of *forward iterators*. Such iterators can do everything input and output iterators can do, plus they can read or write what they point to more than once. This makes them viable for multi-pass algorithms. The STL offers no singly linked list, but some libraries offer one (usually called slist), and iterators into such containers are forward iterators. Iterators into TR1's hashed containers (see Item 54) may also be in the forward category.

Bidirectional iterators add to forward iterators the ability to move backward as well as forward. Iterators for the STL's list are in this category, as are iterators for set, multiset, map, and multimap.

The most powerful iterator category is that of *random access iterators*. These kinds of iterators add to bidirectional iterators the ability to perform "iterator arithmetic," i.e., to jump forward or backward an arbitrary distance in constant time. Such arithmetic is analogous to pointer arithmetic, which is not surprising, because random access iterators are modeled on built-in pointers, and built-in pointers can act as random access iterators. Iterators for vector, deque, and string are random access iterators.

For each of the five iterator categories, C++ has a "tag struct" in the standard library that serves to identify it:

are random access iterators.

For each of the five iterator categories, C++ has a "tag struct" in the standard library that serves to identify it:


```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

The inheritance relationships among these structs are valid is-a relationships (see Item 32): it's true that all forward iterators are also input iterators, etc. We'll see the utility of this inheritance shortly.

As you can see, iterator_traits is a struct. By convention, traits are always implemented as structs. Another convention is that the structs used to implement traits are known as — I am not making this up — traits *classes*.

The way iterator_traits works is that for each type IterT, a typedef named iterator_category is declared in the struct iterator_traits<IterT>. This typedef identifies the iterator category of IterT.

iterator_traits implements this in two parts. First, it imposes the requirement that any user-defined iterator type must contain a nested typedef named iterator_category that identifies the appropriate tag struct. deque's iterators are random access, for example, so a class for deque iterators would look something like this:

```
template < ... >                                    // template params elided
class deque {
public:
  class iterator {
  public:
    typedef random_access_iterator_tag iterator_category;
    ...
  };
  ...
};
```

list's iterators are bidirectional, however, so they'd do things this way:

```
template < ... >
class list {
```

} // category

We can now summarize how to use a traits class:

- Create a set of overloaded "worker" functions or function templates (e.g., doAdvance) that differ in a traits parameter. Implement each function in accord with the traits information passed.
- Create a "master" function or function template (e.g., advance) that calls the workers, passing information provided by a traits class.

Traits are widely used in the standard library. There's iterator_traits, of course, which, in addition to iterator_category, offers four other pieces of information about iterators (the most useful of which is value_type — Item 42 shows an example of its use). There's also char_traits, which holds information about character types, and numeric_limits, which serves up information about numeric types, e.g., their minimum and maximum representable values, etc. (The name numeric_limits is a bit of a surprise, because the more common convention is for traits classes to end with "traits," but numeric_limits is what it's called, so numeric_limits is the name we use.)

TR1 (see Item 54) introduces a slew of new traits classes that give information about types, including is_fundamental<T> (whether T is a built-in type), is_array<T> (whether T is an array type), and is_base_of<T1, T2> (whether T1 is the same as or is a base class of T2). All told, TR1 adds over 50 traits classes to standard C++.

Things to Remember

classes to end with "traits," but numeric_limits is what it's called, so numeric_limits is the name we use.)

TR1 (see Item 54) introduces a slew of new traits classes that give information about types, including is_fundamental<T> (whether T is a built-in type), is_array<T> (whether T is an array type), and is_base_of<T1, T2> (whether T1 is the same as or is a base class of T2). All told, TR1 adds over 50 traits classes to standard C++.

Things to Remember

- Traits classes make information about types available during compilation. They're implemented using templates and template specializations.
- In conjunction with overloading, traits classes make it possible to perform compile-time if...else tests on types.

When operator new is unable to fulfill a memory request, it calls the new-handler function repeatedly until it *can* find enough memory. The code giving rise to these repeated calls is shown in Item 51, but this high-level description is enough to conclude that a well-designed new-handler function must do one of the following:

- **Make more memory available**. This may allow the next memory allocation attempt inside operator new to succeed. One way to implement this strategy is to allocate a large block of memory at program start-up, then release it for use in the program the first time the new-handler is invoked.
- **Install a different new-handler**. If the current new-handler can't make any more memory available, perhaps it knows of a different new-handler that can. If so, the current new-handler can install the other new-handler in its place (by calling set_new_handler). The next time operator new calls the new-handler function, it will get the one most recently installed. (A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static, namespace-specific, or global data that affects the new-handler's behavior.)
- **Deinstall the new-handler**, i.e., pass the null pointer to set_new_handler. With no new-handler installed, operator new will throw an exception when memory allocation is unsuccessful.

the one most recently installed. (A variation on this theme is for a new-handler to modify its *own* behavior, so the next time it's invoked, it does something different. One way to achieve this is to have the new-handler modify static, namespace-specific, or global data that affects the new-handler's behavior.)

- **Deinstall the new-handler**, i.e., pass the null pointer to set_new_handler. With no new-handler installed, operator new will throw an exception when memory allocation is unsuccessful.
- **Throw an exception** of type bad_alloc or some type derived from bad_alloc. Such exceptions will not be caught by operator new, so they will propagate to the site originating the request for memory.
- **Not return**, typically by calling abort or exit.

These choices give you considerable flexibility in implementing new-handler functions.

Until 1993, C++ required that operator new return null when it was unable to allocate the requested memory. operator new is now specified to throw a bad_alloc exception, but a lot of C++ was written before compilers began supporting the revised specification. The C++ standardization committee didn't want to abandon the test-for-null code base, so they provided alternative forms of operator new that offer the traditional failure-yields-null behavior. These forms are called "nothrow" forms, in part because they employ nothrow objects (defined in the header <new>) at the point where new is used:

```
class Widget { ... };
Widget *pw1 = new Widget;                       // throws bad_alloc if
                                                // allocation fails
if (pw1 == 0) ...                               // this test must fail

Widget *pw2 = new (std::nothrow) Widget;        // returns 0 if allocation for
                                                // the Widget fails
if (pw2 == 0) ...                               // this test may succeed
```

Nothrow new offers a less compelling guarantee about exceptions than is initially apparent. In the expression "new (std::nothrow) Widget," two things happen. First, the nothrow version of operator new is called to allocate enough memory for a Widget object. If that allocation fails,