"call" objects of such classes just as if they were functions. We'll also look at new library facilities that make it easy to use different types of callable objects in a uniform way.

This chapter concludes by looking at another special kind of class member function —conversion operators. These operators define implicit conversions from objects of class type. The compiler applies these conversions in the same contexts—and for the same reasons—as it does with conversions among the built-in types.

The last two chapters in this part cover how C++ supports object-oriented and generic programming.

Chapter 15 covers inheritance and dynamic binding. Along with data abstraction, inheritance and dynamic binding are fundamental to object-oriented programming. Inheritance makes it easier for us to define related types and dynamic binding lets us write type-indepenent code that can ignore the differences among types that are related by inheritance.

Chapter 16 covers function and class templates. Templates let us write generic classes and functions that are type-independent. A number of new template-related features were introduced by the new standard: variadic templates, template type aliases, and new ways to control instantiation.

Writing our own object-oriented or generic types requires a fairly good understanding of C++. Fortunately, we can use object-oriented and generic types without understanding the details of how to build them. For example, the standard library uses the facilities we'll study in Chapters 15 and 16 extensively, and we've used the library types and algorithms without needing to know how they are implemented.

Readers, therefore, should understand that Part III covers fairly advanced topics. Writing templates or object-oriented classes requires a good understanding of the basics of C++ and a good grasp of how to define more basic classes.

Chapter 13. Copy Control

Contents

Defined Terms

As we saw in Chapter 7, each class defines a new type and defines the operations that objects of that type can perform. In that chapter, we also learned that classes can define constructors, which control what happens when objects of the class type are created.

In this chapter we'll learn how classes can control what happens when objects of the class type are copied, assigned, moved, or destroyed. Classes control these actions through special member functions: the copy constructor, move constructor, copy-assignment operator, move-assignment operator, and destructor.

When we define a class, we specify—explicitly or implicitly—what happens when objects of that class type are copied, moved, assigned, and destroyed. A class controls these operations by defining five special member functions: **copy constructor**, **copy-assignment operator**, **move constructor**, **move-assignment operator**, and **destructor**. The copy and move constructors define what happens when an object is initialized from another object of the same type. The copy- and move-assignment operators define what happens when we assign an object of a class type to another object of that same class type. The destructor defines what happens when an object of the type ceases to exist. Collectively, we'll refer to these operations as **copy control**.

If a class does not define all of the copy-control members, the compiler automatically defines the missing operations. As a result, many classes can ignore copy control (§ 7.1.5, p. 267). However, for some classes, relying on the default definitions leads to disaster. Frequently, the hardest part of implementing copy-control operations is recognizing when we need to define them in the first place.

Warning

Copy control is an essential part of defining any C++ class. Programmers new to C++ are often confused by having to define what happens when objects are copied, moved, assigned, or destroyed. This confusion is compounded because if we do not explicitly define these operations, the compiler defines them for us—although the compiler-defined versions might not behave as we intend.

13.1. Copy, Assign, and Destroy

We'll start by covering the most basic operations, which are the copy constructor, copy-assignment operator, and destructor. We'll cover the move operations (which were introduced by the new standard) in § 13.6 (p. 531).

13.1.1. The Copy Constructor

A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values:

Click here to view code image

```
class Foo {
public:
   Foo();              // default constructor
   Foo(const Foo&);    // copy constructor
   // ...
};
```

For reasons we'll explain shortly, the first parameter must be a reference type. That parameter is almost always a reference to `const`, although we can define the copy constructor to take a reference to non`const`. The copy constructor is used implicitly in several circumstances. Hence, the copy constructor usually should not be `explicit` (§ 7.5.4, p. 296).

The Synthesized Copy Constructor

When we do not define a copy constructor for a class, the compiler synthesizes one for us. Unlike the synthesized default constructor (§ 7.1.4, p. 262), a copy constructor is synthesized even if we define other constructors.

As we'll see in § 13.1.6 (p. 508), the **synthesized copy constructor** for some classes prevents us from copying objects of that class type. Otherwise, the synthesized copy constructor **memberwise copies** the members of its argument into the object being created (§ 7.1.5, p. 267). The compiler copies each non`static` member in turn from the given object into the one being created.

The type of each member determines how that member is copied: Members of class type are copied by the copy constructor for that class; members of built-in type are copied directly. Although we cannot directly copy an array (§ 3.5.1, p. 114), the synthesized copy constructor copies members of array type by copying each element. Elements of class type are copied by using the elements' copy constructor.

As an example, the synthesized copy constructor for our `Sales_data` class is equivalent to:

Click here to view code image

```
class Sales_data {
public:
    // other members and constructors as before
    // declaration equivalent to the synthesized copy constructor
    Sales_data(const Sales_data&);
private:
    std::string bookNo;
    int units_sold = 0;
    double revenue = 0.0;
};
// equivalent to the copy constructor that would be synthesized for Sales_data
Sales_data::Sales_data(const Sales_data &orig):
    bookNo(orig.bookNo),         // uses the string copy constructor
    units_sold(orig.units_sold), // copies orig.units_sold
    revenue(orig.revenue)        // copies orig.revenue
    {   }                        // empty body
```

Copy Initialization

We are now in a position to fully understand the differences between direct initialization and copy initialization (§ 3.2.1, p. 84):

Click here to view code image

```
    string dots(10, '.');               // direct initialization
    string s(dots);                     // direct initialization
    string s2 = dots;                   // copy initialization
    string null_book = "9-999-99999-9"; // copy initialization
    string nines = string(100, '9');    // copy initialization
```

When we use direct initialization, we are asking the compiler to use ordinary function matching (§ 6.4, p. 233) to select the constructor that best matches the arguments we provide. When we use **copy initialization**, we are asking the compiler to copy the right-hand operand into the object being created, converting that operand if necessary (§ 7.5.4, p. 294).

Copy initialization ordinarily uses the copy constructor. However, as we'll see in § 13.6.2 (p. 534), if a class has a move constructor, then copy initialization sometimes uses the move constructor instead of the copy constructor. For now, what's useful to know is when copy initialization happens and that copy initialization requires either the copy constructor or the move constructor.

Copy initialization happens not only when we define variables using an =, but also when we

- Pass an object as an argument to a parameter of nonreference type
- Return an object from a function that has a nonreference return type
- Brace initialize the elements in an array or the members of an aggregate class (§ 7.5.5, p. 298)

Some class types also use copy initialization for the objects they allocate. For example, the library containers copy initialize their elements when we initialize the container, or when we call an `insert` or `push` member (§ 9.3.1, p. 342). By contrast, elements created by an `emplace` member are direct initialized (§ 9.3.1, p. 345).

Parameters and Return Values

During a function call, parameters that have a nonreference type are copy initialized (§ 6.2.1, p. 209). Similarly, when a function has a nonreference return type, the return value is used to copy initialize the result of the call operator at the call site (§ 6.3.2, p. 224).

The fact that the copy constructor is used to initialize nonreference parameters of class type explains why the copy constructor's own parameter must be a reference. If that parameter were not a reference, then the call would never succeed—to call the copy constructor, we'd need to use the copy constructor to copy the argument, but to copy the argument, we'd need to call the copy constructor, and so on indefinitely.

Constraints on Copy Initialization

As we've seen, whether we use copy or direct initialization matters if we use an initializer that requires conversion by an `explicit` constructor (§ 7.5.4, p. 296):

Click here to view code image

```
vector<int> v1(10);   // ok: direct initialization
vector<int> v2 = 10; // error: constructor that takes a size is explicit
void f(vector<int>); // f's parameter is copy initialized
f(10); // error: can't use an explicit constructor to copy an argument
f(vector<int>(10));  // ok: directly construct a temporary vector from an int
```

Directly initializing `v1` is fine, but the seemingly equivalent copy initialization of `v2` is an error, because the `vector` constructor that takes a single size parameter is `explicit`. For the same reasons that we cannot copy initialize `v2`, we cannot implicitly use an `explicit` constructor when we pass an argument or return a value from a function. If we want to use an `explicit` constructor, we must do so explicitly, as in the last line of the example above.

The Compiler Can Bypass the Copy Constructor

During copy initialization, the compiler is permitted (but not obligated) to skip the copy/move constructor and create the object directly. That is, the compiler is permitted to rewrite

Click here to view code image

```
string null_book = "9-999-99999-9"; // copy initialization
```

into

Click here to view code image

`string null_book("9-999-99999-9");` *// compiler omits the copy constructor*

However, even if the compiler omits the call to the copy/move constructor, the copy/move constructor must exist and must be accessible (e.g., not `private`) at that point in the program.

Exercises Section 13.1.1

Exercise 13.1: What is a copy constructor? When is it used?

Exercise 13.2: Explain why the following declaration is illegal:

Click here to view code image

```
Sales_data::Sales_data(Sales_data rhs);
```

Exercise 13.3: What happens when we copy a `StrBlob`? What about `StrBlobPtr`s?

Exercise 13.4: Assuming `Point` is a class type with a `public` copy constructor, identify each use of the copy constructor in this program fragment:

Click here to view code image

```
Point global;
Point foo_bar(Point arg)
{
    Point local = arg, *heap = new Point(global);
    *heap = local;
    Point pa[ 4 ] = { local, *heap };
    return *heap;
}
```

Exercise 13.5: Given the following sketch of a class, write a copy constructor that copies all the members. Your constructor should dynamically allocate a new `string` (§ 12.1.2, p. 458) and copy the object to which `ps` points, rather than copying `ps` itself.

Click here to view code image

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
private:
    std::string *ps;
    int    i;
};
```

13.1.2. The Copy-Assignment Operator

Just as a class controls how objects of that class are initialized, it also controls how objects of its class are assigned:

Click here to view code image

```
Sales_data trans, accum;
trans = accum; // uses the Sales_data copy-assignment operator
```

As with the copy constructor, the compiler synthesizes a copy-assignment operator if the class does not define its own.

Introducing Overloaded Assignment

Before we look at the synthesized assignment operator, we need to know a bit about **overloaded operators**, which we cover in detail in Chapter 14.

Overloaded operators are functions that have the name `operator` followed by the symbol for the operator being defined. Hence, the assignment operator is a function named `operator=`. Like any other function, an operator function has a return type and a parameter list.

The parameters in an overloaded operator represent the operands of the operator. Some operators, assignment among them, must be defined as member functions. When an operator is a member function, the left-hand operand is bound to the implicit `this` parameter (§ 7.1.2, p. 257). The right-hand operand in a binary operator, such as assignment, is passed as an explicit parameter.

The copy-assignment operator takes an argument of the same type as the class:

Click here to view code image

```
class Foo {
public:
    Foo& operator=(const Foo&); // assignment operator
   // ...
};
```

To be consistent with assignment for the built-in types (§ 4.4, p. 145), assignment operators usually return a reference to their left-hand operand. It is also worth noting that the library generally requires that types stored in a container have assignment operators that return a reference to the left-hand operand.

Best Practices

Assignment operators ordinarily should return a reference to their left-hand operand.

The Synthesized Copy-Assignment Operator

Just as it does for the copy constructor, the compiler generates a **synthesized copy-assignment operator** for a class if the class does not define its own. Analogously to the copy constructor, for some classes the synthesized copy-assignment operator disallows assignment (§ 13.1.6, p. 508). Otherwise, it assigns each nonstatic member of the right-hand object to the corresponding member of the left-hand object using the copy-assignment operator for the type of that member. Array members are assigned by assigning each element of the array. The synthesized copy-assignment operator returns a reference to its left-hand object.

As an example, the following is equivalent to the synthesized `Sales_data` copy-assignment operator:

Click here to view code image

```
// equivalent to the synthesized copy-assignment operator
Sales_data&
Sales_data::operator=(const Sales_data &rhs)
{
    bookNo = rhs.bookNo;          // calls the string::operator=
    units_sold = rhs.units_sold;  // uses the built-in int assignment
     revenue = rhs.revenue;          // uses the built-in double
assignment
    return *this;                 // return a reference to this object
}
```

Exercises Section 13.1.2

Exercise 13.6: What is a copy-assignment operator? When is this operator used? What does the synthesized copy-assignment operator do? When is it synthesized?

Exercise 13.7: What happens when we assign one `StrBlob` to another? What about `StrBlobPtrs`?

Exercise 13.8: Write the assignment operator for the `HasPtr` class from exercise 13.5 in § 13.1.1 (p. 499). As with the copy constructor, your assignment operator should copy the object to which `ps` points.

13.1.3. The Destructor

The destructor operates inversely to the constructors: Constructors initialize the `nonstatic` data members of an object and may do other work; destructors do whatever work is needed to free the resources used by an object and destroy the `nonstatic` data members of the object.

The destructor is a member function with the name of the class prefixed by a tilde (~). It has no return value and takes no parameters:

```
class Foo {
public:
    ~Foo();     // destructor
   // ...
};
```

Because it takes no parameters, it cannot be overloaded. There is always only one destructor for a given class.

What a Destructor Does

Just as a constructor has an initialization part and a function body (§ 7.5.1, p. 288), a destructor has a function body and a destruction part. In a constructor, members are initialized before the function body is executed, and members are initialized in the same order as they appear in the class. In a destructor, the function body is executed first and then the members are destroyed. Members are destroyed in reverse order from the order in which they were initialized.

The function body of a destructor does whatever operations the class designer wishes to have executed subsequent to the last use of an object. Typically, the destructor frees resources an object allocated during its lifetime.

In a destructor, there is nothing akin to the constructor initializer list to control how members are destroyed; the destruction part is implicit. What happens when a member is destroyed depends on the type of the member. Members of class type are destroyed by running the member's own destructor. The built-in types do not have destructors, so nothing is done to destroy members of built-in type.

Note

The implicit destruction of a member of built-in pointer type does *not* `delete` the object to which that pointer points.

Unlike ordinary pointers, the smart pointers (§ 12.1.1, p. 452) are class types and have destructors. As a result, unlike ordinary pointers, members that are smart pointers are automatically destroyed during the destruction phase.

When a Destructor Is Called

The destructor is used automatically whenever an object of its type is destroyed:

- Variables are destroyed when they go out of scope.
- Members of an object are destroyed when the object of which they are a part is destroyed.
- Elements in a container—whether a library container or an array—are destroyed when the container is destroyed.
- Dynamically allocated objects are destroyed when the `delete` operator is applied to a pointer to the object (§ 12.1.2, p. 460).
- Temporary objects are destroyed at the end of the full expression in which the temporary was created.

Because destructors are run automatically, our programs can allocate resources and (usually) not worry about when those resources are released.

For example, the following fragment defines four `Sales_data` objects:

Click here to view code image

```
{ // new scope
    // p and p2 point to dynamically allocated objects
    Sales_data  *p = new Sales_data;      // p is a built-in pointer
    auto p2 = make_shared<Sales_data>(); // p2 is a shared_ptr
    Sales_data item(*p);      // copy constructor copies *p into item
    vector<Sales_data> vec;   // local object
    vec.push_back(*p2);       // copies the object to which p2 points
    delete p;                  // destructor called on the object pointed to
by p
} // exit local scope; destructor called on item, p2, and vec
  // destroying p2 decrements its use count; if the count goes to 0, the object is freed
  // destroying vec destroys the elements in vec
```

Each of these objects contains a `string` member, which allocates dynamic memory to contain the characters in its `bookNo` member. However, the only memory our code has to manage directly is the object we directly allocated. Our code directly frees only the dynamically allocated object bound to `p`.

The other `Sales_data` objects are automatically destroyed when they go out of scope. When the block ends, `vec`, `p2`, and `item` all go out of scope, which means that the `vector`, `shared_ptr`, and `Sales_data` destructors will be run on those objects, respectively. The `vector` destructor will destroy the element we pushed onto `vec`. The `shared_ptr` destructor will decrement the reference count of the object to which `p2` points. In this example, that count will go to zero, so the `shared_ptr` destructor will `delete` the `Sales_data` object that `p2` allocated.

In all cases, the `Sales_data` destructor implicitly destroys the `bookNo` member. Destroying `bookNo` runs the `string` destructor, which frees the memory used to store the ISBN.

Note

The destructor is *not* run when a reference or a pointer to an object goes out of scope.

The Synthesized Destructor

The compiler defines a **synthesized destructor** for any class that does not define its own destructor. As with the copy constructor and the copy-assignment operator, for some classes, the synthesized destructor is defined to disallow objects of the type from being destroyed (§ 13.1.6, p. 508). Otherwise, the synthesized destructor has an empty function body.

For example, the synthesized `Sales_data` destructor is equivalent to:

Click here to view code image

```
class Sales_data {
public:
    // no work to do other than destroying the members, which happens automatically
     ~Sales_data() { }
    // other members as before
};
```

The members are automatically destroyed after the (empty) destructor body is run. In particular, the `string` destructor will be run to free the memory used by the `bookNo` member.

It is important to realize that the destructor body does not directly destroy the members themselves. Members are destroyed as part of the implicit destruction phase that follows the destructor body. A destructor body executes *in addition to* the memberwise destruction that takes place as part of destroying an object.

13.1.4. The Rule of Three/Five

As we’ve seen, there are three basic operations to control copies of class objects: the copy constructor, copy-assignment operator, and destructor. Moreover, as we’ll see in § 13.6 (p. 531), under the new standard, a class can also define a move constructor and move-assignment operator.

Exercises Section 13.1.3

Exercise 13.9: What is a destructor? What does the synthesized destructor do? When is a destructor synthesized?

Exercise 13.10: What happens when a `StrBlob` object is destroyed? What about a `StrBlobPtr`?

Exercise 13.11: Add a destructor to your `HasPtr` class from the previous exercises.

Exercise 13.12: How many destructor calls occur in the following code fragment?

Click here to view code image

```
bool fcn(const Sales_data *trans, Sales_data accum)
{
    Sales_data item1(*trans), item2(accum);
    return item1.isbn() != item2.isbn();
}
```

Exercise 13.13: A good way to understand copy-control members and constructors is to define a simple class with these members in which each member prints its name:

Click here to view code image

```
struct X {
    X() {std::cout << "X()" << std::endl;}
        X(const  X&)  {std::cout  <<  "X(const  X&)"  <<
std::endl;}
};
```

Add the copy-assignment operator and destructor to `X` and write a program using `X` objects in various ways: Pass them as nonreference and reference parameters; dynamically allocate them; put them in containers; and so forth. Study the output until you are certain you understand when and why each copy-control member is used. As you read the output, remember that the compiler can omit calls to the copy constructor.

There is no requirement that we define all of these operations: We can define one or two of them without having to define all of them. However, ordinarily these operations should be thought of as a unit. In general, it is unusual to need one without needing to define them all.

Classes That Need Destructors Need Copy and Assignment

One rule of thumb to use when you decide whether a class needs to define its own versions of the copy-control members is to decide first whether the class needs a destructor. Often, the need for a destructor is more obvious than the need for the copy constructor or assignment operator. If the class needs a destructor, it almost surely needs a copy constructor and copy-assignment operator as well.

The `HasPtr` class that we have used in the exercises is a good example (§ 13.1.1, p. 499). That class allocates dynamic memory in its constructor. The synthesized destructor will not `delete` a data member that is a pointer. Therefore, this class needs to define a destructor to free the memory allocated by its constructor.

What may be less clear—but what our rule of thumb tells us—is that `HasPtr` also needs a copy constructor and copy-assignment operator.

Consider what would happen if we gave `HasPtr` a destructor but used the synthesized versions of the copy constructor and copy-assignment operator:

Click here to view code image

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    ~HasPtr() { delete ps; }
    // WRONG: HasPtr needs a copy constructor and copy-assignment operator
    // other members as before
};
```

In this version of the class, the memory allocated in the constructor will be freed when a `HasPtr` object is destroyed. Unfortunately, we have introduced a serious bug! This version of the class uses the synthesized versions of copy and assignment. Those functions copy the pointer member, meaning that multiple `HasPtr` objects may be pointing to the same memory:

Click here to view code image

```
HasPtr f(HasPtr hp)   // HasPtr passed by value, so it is copied
{
    HasPtr ret = hp; // copies the given HasPtr
    // process ret
    return ret;       // ret and hp are destroyed
}
```

When `f` returns, both `hp` and `ret` are destroyed and the `HasPtr` destructor is run on each of these objects. That destructor will `delete` the pointer member in `ret` and in `hp`. But these objects contain the same pointer value. This code will `delete` that pointer twice, which is an error (§ 12.1.2, p. 462). What happens is undefined.

In addition, the caller of `f` may still be using the object that was passed to `f`:

Click here to view code image

```
HasPtr p("some values");
f(p);          // when f completes, the memory to which p.ps points is freed
HasPtr q(p);  // now both p and q point to invalid memory!
```

The memory to which `p` (and `q`) points is no longer valid. It was returned to the system when `hp` (or `ret!`) was destroyed.

Tip

If a class needs a destructor, it almost surely also needs the copy-assignment operator and a copy constructor.

Classes That Need Copy Need Assignment, and Vice Versa

Although many classes need to define all of (or none of) the copy-control members, some classes have work that needs to be done to copy or assign objects but has no need for the destructor.

As an example, consider a class that gives each object its own, unique serial number. Such a class would need a copy constructor to generate a new, distinct serial number for the object being created. That constructor would copy all the other data members from the given object. This class would also need its own copy-assignment operator to avoid assigning to the serial number of the left-hand object. However, this class would have no need for a destructor.

This example gives rise to a second rule of thumb: If a class needs a copy constructor, it almost surely needs a copy-assignment operator. And vice versa—if the class needs an assignment operator, it almost surely needs a copy constructor as well. Nevertheless, needing either the copy constructor or the copy-assignment operator does not (necessarily) indicate the need for a destructor.

Exercises Section 13.1.4

Exercise 13.14: Assume that `numbered` is a class with a default constructor that generates a unique serial number for each object, which is stored in a data member named `mysn`. Assuming `numbered` uses the synthesized copy-control members and given the following function:

Click here to view code image

```
void f (numbered s) { cout << s.mysn << endl; }
```

what output does the following code produce?

Click here to view code image

```
numbered a, b = a, c = b;
f(a); f(b); f(c);
```

Exercise 13.15: Assume `numbered` has a copy constructor that generates a new serial number. Does that change the output of the calls in the previous exercise? If so, why? What output gets generated?

Exercise 13.16: What if the parameter in `f` were `const numbered&`? Does that change the output? If so, why? What output gets generated?

Exercise 13.17: Write versions of `numbered` and `f` corresponding to the previous three exercises and check whether you correctly predicted the output.

13.1.5. Using `= default`

C++ 11

We can explicitly ask the compiler to generate the synthesized versions of the copy-control members by defining them as `= default` (§ 7.1.4, p. 264):

Click here to view code image

```
class Sales_data {
public:
    // copy control; use defaults
    Sales_data() = default;
    Sales_data(const Sales_data&) = default;
    Sales_data& operator=(const Sales_data &);
    ~Sales_data() = default;
    // other members as before
};
Sales_data&   Sales_data::operator=(const   Sales_data&)   =
default;
```

When we specify `= default` on the declaration of the member inside the class body, the synthesized function is implicitly inline (just as is any other member function defined in the body of the class). If we do not want the synthesized member to be an inline function, we can specify `= default` on the member's definition, as we do in the definition of the copy-assignment operator.

Note

We can use `= default` only on member functions that have a synthesized version (i.e., the default constructor or a copy-control member).

13.1.6. Preventing Copies

Best Practices

Most classes should define—either implicitly or explicitly—the default and copy constructors and the copy-assignment operator.

Although most classes should (and generally do) define a copy constructor and a copy-assignment operator, for some classes, there really is no sensible meaning for these operations. In such cases, the class must be defined so as to prevent copies or assignments from being made. For example, the `iostream` classes prevent copying to avoid letting multiple objects write to or read from the same IO buffer. It might seem that we could prevent copies by not defining the copy-control members. However, this strategy doesn't work: If our class doesn't define these operations, the compiler will synthesize them.

Defining a Function as Deleted

Under the new standard, we can prevent copies by defining the copy constructor and copy-assignment operator as **deleted functions**. A deleted function is one that is declared but may not be used in any other way. We indicate that we want to define a function as deleted by following its parameter list with `= delete`:

Click here to view code image

```
struct NoCopy {
    NoCopy() = default;    // use the synthesized default constructor
    NoCopy(const NoCopy&) = delete;            // no copy
    NoCopy &operator=(const NoCopy&) = delete; // no assignment
    ~NoCopy() = default;   // use the synthesized destructor
    // other members
};
```

The `= delete` signals to the compiler (and to readers of our code) that we are intentionally *not defining* these members.

Unlike `= default`, `= delete` must appear on the first declaration of a deleted function. This difference follows logically from the meaning of these declarations. A defaulted member affects only what code the compiler generates; hence the `= default` is not needed until the compiler generates code. On the other hand, the compiler needs to know that a function is deleted in order to prohibit operations that attempt to use it.

Also unlike `= default`, we can specify `= delete` on any function (we can use =

`default` only on the default constructor or a copy-control member that the compiler can synthesize). Although the primary use of deleted functions is to suppress the copy-control members, deleted functions are sometimes also useful when we want to guide the function-matching process.

The Destructor Should Not be a Deleted Member

It is worth noting that we did not delete the destructor. If the destructor is deleted, then there is no way to destroy objects of that type. The compiler will not let us define variables or create temporaries of a type that has a deleted destructor. Moreover, we cannot define variables or temporaries of a class that has a member whose type has a deleted destructor. If a member has a deleted destructor, then that member cannot be destroyed. If a member can't be destroyed, the object as a whole can't be destroyed.

Although we cannot define variables or members of such types, we can dynamically allocate objects with a deleted destructor. However, we cannot free them:

Click here to view code image

```
struct NoDtor {
    NoDtor() =  default;   // use the synthesized default constructor
    ~NoDtor() = delete;   // we can't destroy objects of type NoDtor
};
NoDtor nd;  // error: NoDtor destructor is deleted
NoDtor *p = new NoDtor();    // ok: but we can't delete p
delete p; // error: NoDtor destructor is deleted
```

Warning

It is not possible to define an object or delete a pointer to a dynamically allocated object of a type with a deleted destructor.

The Copy-Control Members May Be Synthesized as Deleted

As we've seen, if we do not define the copy-control members, the compiler defines them for us. Similarly, if a class defines no constructors, the compiler synthesizes a default constructor for that class (§ 7.1.4, p. 262). For some classes, the compiler defines these synthesized members as deleted functions:

- The synthesized destructor is defined as deleted if the class has a member whose own destructor is deleted or is inaccessible (e.g., `private`).
- The synthesized copy constructor is defined as deleted if the class has a member whose own copy constructor is deleted or inaccessible. It is also deleted

if the class has a member with a deleted or inaccessible destructor.

- The synthesized copy-assignment operator is defined as deleted if a member has a deleted or inaccessible copy-assignment operator, or if the class has a `const` or reference member.
- The synthesized default constructor is defined as deleted if the class has a member with a deleted or inaccessible destructor; or has a reference member that does not have an in-class initializer (§ 2.6.1, p. 73); or has a `const` member whose type does not explicitly define a default constructor and that member does not have an in-class initializer.

In essence, these rules mean that if a class has a data member that cannot be default constructed, copied, assigned, or destroyed, then the corresponding member will be a deleted function.

It may be surprising that a member that has a deleted or inaccessible destructor causes the synthesized default and copy constructors to be defined as deleted. The reason for this rule is that without it, we could create objects that we could not destroy.

It should not be surprising that the compiler will not synthesize a default constructor for a class with a reference member or a `const` member that cannot be default constructed. Nor should it be surprising that a class with a `const` member cannot use the synthesized copy-assignment operator: After all, that operator attempts to assign to every member. It is not possible to assign a new value to a `const` object.

Although we can assign a new value to a reference, doing so changes the value of the object to which the reference refers. If the copy-assignment operator were synthesized for such classes, the left-hand operand would continue to refer to the same object as it did before the assignment. It would not refer to the same object as the right-hand operand. Because this behavior is unlikely to be desired, the synthesized copy-assignment operator is defined as deleted if the class has a reference member.

We'll see in § 13.6.2 (p. 539), § 15.7.2 (p. 624), and § 19.6 (p. 849) that there are other aspects of a class that can cause its copy members to be defined as deleted.

Note

In essence, the copy-control members are synthesized as deleted when it is impossible to copy, assign, or destroy a member of the class.

`private` Copy Control

Prior to the new standard, classes prevented copies by declaring their copy constructor and copy-assignment operator as `private`:

Click here to view code image

```
class PrivateCopy {
    // no access specifier; following members are private by default; see § 7.2 (p. 268)
    // copy control is private and so is inaccessible to ordinary user code
    PrivateCopy(const PrivateCopy&);
    PrivateCopy &operator=(const PrivateCopy&);
    // other members
public:
    PrivateCopy() = default; // use the synthesized default constructor
    ~PrivateCopy(); // users can define objects of this type but not copy them
};
```

Because the destructor is `public`, users will be able to define `PrivateCopy` objects. However, because the copy constructor and copy-assignment operator are `private`, user code will not be able to copy such objects. However, friends and members of the class can still make copies. To prevent copies by friends and members, we declare these members as `private` but do not define them.

With one exception, which we'll cover in § 15.2.1 (p. 594), it is legal to declare, but not define, a member function (§ 6.1.2, p. 206). An attempt to *use* an undefined member results in a link-time failure. By declaring (but not defining) a `private` copy constructor, we can forestall any attempt to copy an object of the class type: User code that tries to make a copy will be flagged as an error at compile time; copies made in member functions or friends will result in an error at link time.

★ Best Practices

Classes that want to prevent copying should define their copy constructor and copy-assignment operators using `= delete` rather than making those members `private`.

Exercises Section 13.1.6

Exercise 13.18: Define an `Employee` class that contains an employee name and a unique employee identifier. Give the class a default constructor and a constructor that takes a `string` representing the employee's name. Each constructor should generate a unique ID by incrementing a `static` data member.

Exercise 13.19: Does your `Employee` class need to define its own versions of the copy-control members? If so, why? If not, why not? Implement whatever copy-control members you think `Employee` needs.

Exercise 13.20: Explain what happens when we copy, assign, or destroy objects of our `TextQuery` and `QueryResult` classes from § 12.3 (p. 484).

Exercise 13.21: Do you think the `TextQuery` and `QueryResult` classes need to define their own versions of the copy-control members? If so, why? If not, why not? Implement whichever copy-control operations you think these classes require.

13.2. Copy Control and Resource Management

Ordinarily, classes that manage resources that do not reside in the class must define the copy-control members. As we saw in § 13.1.4 (p. 504), such classes will need destructors to free the resources allocated by the object. Once a class needs a destructor, it almost surely needs a copy constructor and copy-assignment operator as well.

In order to define these members, we first have to decide what copying an object of our type will mean. In general, we have two choices: We can define the copy operations to make the class behave like a value or like a pointer.

Classes that behave like values have their own state. When we copy a valuelike object, the copy and the original are independent of each other. Changes made to the copy have no effect on the original, and vice versa.

Classes that act like pointers share state. When we copy objects of such classes, the copy and the original use the same underlying data. Changes made to the copy also change the original, and vice versa.

Of the library classes we've used, the library containers and `string` class have valuelike behavior. Not surprisingly, the `shared_ptr` class provides pointerlike behavior, as does our `StrBlob` class (§ 12.1.1, p. 456). The IO types and `unique_ptr` do not allow copying or assignment, so they provide neither valuelike nor pointerlike behavior.

To illustrate these two approaches, we'll define the copy-control members for the `HasPtr` class used in the exercises. First, we'll make the class act like a value; then we'll reimplement the class making it behave like a pointer.

Our `HasPtr` class has two members, an `int` and a pointer to `string`. Ordinarily, classes copy members of built-in type (other than pointers) directly; such members are values and hence ordinarily ought to behave like values. What we do when we copy the pointer member determines whether a class like `HasPtr` has valuelike or pointerlike behavior.

Exercises Section 13.2

Exercise 13.22: Assume that we want `HasPtr` to behave like a value. That is, each object should have its own copy of the string to which the objects point. We'll show the definitions of the copy-control members in the next section. However, you already know everything you need to know to implement these members. Write the HasPtr copy constructor and copy-assignment operator before reading on.

13.2.1. Classes That Act Like Values

To provide valuelike behavior, each object has to have its own copy of the resource that the class manages. That means each HasPtr object must have its own copy of the string to which ps points. To implement valuelike behavior HasPtr needs

- A copy constructor that copies the string, not just the pointer
- A destructor to free the string
- A copy-assignment operator to free the object's existing string and copy the string from its right-hand operand

The valuelike version of HasPtr is

Click here to view code image

```
class HasPtr {
public:
    HasPtr(const std::string &s = std::string()):
        ps(new std::string(s)), i(0) { }
    // each HasPtr has its own copy of the string to which ps points
    HasPtr(const HasPtr &p):
        ps(new std::string(*p.ps)), i(p.i) { }
    HasPtr& operator=(const HasPtr &);
    ~HasPtr() { delete ps; }
private:
    std::string *ps;
    int    i;
};
```

Our class is simple enough that we've defined all but the assignment operator in the class body. The first constructor takes an (optional) string argument. That constructor dynamically allocates its own copy of that string and stores a pointer to that string in ps. The copy constructor also allocates its own, separate copy of the string. The destructor frees the memory allocated in its constructors by executing delete on the pointer member, ps.

Valuelike Copy-Assignment Operator

Assignment operators typically combine the actions of the destructor and the copy constructor. Like the destructor, assignment destroys the left-hand operand's resources. Like the copy constructor, assignment copies data from the right-hand operand. However, it is crucially important that these actions be done in a sequence that is correct even if an object is assigned to itself. Moreover, when possible, we should also write our assignment operators so that they will leave the left-hand operand in a sensible state should an exception occur (§ 5.6.2, p. 196).

In this case, we can handle self-assignment—and make our code safe should an exception happen—by first copying the right-hand side. After the copy is made, we'll free the left-hand side and update the pointer to point to the newly allocated `string`:

Click here to view code image

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    auto newp = new string(*rhs.ps);   // copy the underlying string
    delete ps;       // free the old memory
    ps = newp;       // copy data from rhs into this object
    i = rhs.i;
    return *this;    // return this object
}
```

In this assignment operator, we quite clearly first do the work of the constructor: The initializer of `newp` is identical to the initializer of `ps` in `HasPtr`'s copy constructor. As in the destructor, we next `delete` the `string` to which `ps` currently points. What remains is to copy the pointer to the newly allocated `string` and the `int` value from `rhs` into this object.

Key Concept: Assignment Operators

There are two points to keep in mind when you write an assignment operator:

- Assignment operators must work correctly if an object is assigned to itself.
- Most assignment operators share work with the destructor and copy constructor.

A good pattern to use when you write an assignment operator is to first copy the right-hand operand into a local temporary. *After* the copy is done, it is safe to destroy the existing members of the left-hand operand. Once the left-hand operand is destroyed, copy the data from the temporary into the members of the left-hand operand.

To illustrate the importance of guarding against self-assignment, consider what would happen if we wrote the assignment operator as

Click here to view code image

```
// WRONG way to write an assignment operator!
HasPtr&
HasPtr::operator=(const HasPtr &rhs)
{
    delete ps;    // frees the string to which this object points
    // if rhs and *this are the same object, we're copying from deleted memory!
    ps = new string(*(rhs.ps));
    i = rhs.i;
    return *this;
}
```

If `rhs` and this object are the same object, deleting `ps` frees the `string` to which both `*this` and `rhs` point. When we attempt to copy `* (rhs.ps)` in the `new` expression, that pointer points to invalid memory. What happens is undefined.

Warning

It is crucially important for assignment operators to work correctly, even when an object is assigned to itself. A good way to do so is to copy the right-hand operand before destroying the left-hand operand.

Exercises Section 13.2.1

Exercise 13.23: Compare the copy-control members that you wrote for the solutions to the previous section's exercises to the code presented here. Be sure you understand the differences, if any, between your code and ours.

Exercise 13.24: What would happen if the version of `HasPtr` in this section didn't define a destructor? What if `HasPtr` didn't define the copy constructor?

Exercise 13.25: Assume we want to define a version of `StrBlob` that acts like a value. Also assume that we want to continue to use a `shared_ptr` so that our `StrBlobPtr` class can still use a `weak_ptr` to the `vector`. Your revised class will need a copy constructor and copy-assignment operator but will not need a destructor. Explain what the copy constructor and copy-assignment operators must do. Explain why the class does not need a destructor.

Exercise 13.26: Write your own version of the `StrBlob` class described in the previous exercise.

13.2.2. Defining Classes That Act Like Pointers

For our HasPtr class to act like a pointer, we need the copy constructor and copy-assignment operator to copy the pointer member, not the string to which that pointer points. Our class will still need its own destructor to free the memory allocated by the constructor that takes a string (§ 13.1.4, p. 504). In this case, though, the destructor cannot unilaterally free its associated string. It can do so only when the last HasPtr pointing to that string goes away.

The easiest way to make a class act like a pointer is to use shared_ptrs to manage the resources in the class. Copying (or assigning) a shared_ptr copies (assigns) the pointer to which the shared_ptr points. The shared_ptr class itself keeps track of how many users are sharing the pointed-to object. When there are no more users, the shared_ptr class takes care of freeing the resource.

However, sometimes we want to manage a resource directly. In such cases, it can be useful to use a **reference count** (§ 12.1.1, p. 452). To show how reference counting works, we'll redefine HasPtr to provide pointerlike behavior, but we will do our own reference counting.

Reference Counts

Reference counting works as follows:

- In addition to initializing the object, each constructor (other than the copy constructor) creates a counter. This counter will keep track of how many objects share state with the object we are creating. When we create an object, there is only one such object, so we initialize the counter to 1.
- The copy constructor does not allocate a new counter; instead, it copies the data members of its given object, including the counter. The copy constructor increments this shared counter, indicating that there is another user of that object's state.
- The destructor decrements the counter, indicating that there is one less user of the shared state. If the count goes to zero, the destructor deletes that state.
- The copy-assignment operator increments the right-hand operand's counter and decrements the counter of the left-hand operand. If the counter for the left-hand operand goes to zero, there are no more users. In this case, the copy-assignment operator must destroy the state of the left-hand operand.

The only wrinkle is deciding where to put the reference count. The counter cannot be a direct member of a HasPtr object. To see why, consider what happens in the following example:

Click here to view code image

```
HasPtr p1("Hiya!");
HasPtr p2(p1);  // p1 and p2 point to the same string
HasPtr p3(p1);  // p1, p2, and p3 all point to the same string
```

If the reference count is stored in each object, how can we update it correctly when p3 is created? We could increment the count in p1 and copy that count into p3, but how would we update the counter in p2?

One way to solve this problem is to store the counter in dynamic memory. When we create an object, we'll also allocate a new counter. When we copy or assign an object, we'll copy the pointer to the counter. That way the copy and the original will point to the same counter.

Defining a Reference-Counted Class

Using a reference count, we can write the pointerlike version of `HasPtr` as follows:

Click here to view code image

```
class HasPtr {
public:
    // constructor allocates a new string and a new counter, which it sets to 1
    HasPtr(const std::string &s = std::string()):
       ps(new std::string(s)), i(0), use(new std::size_t(1))
{}
    // copy constructor copies all three data members and increments the counter
    HasPtr(const HasPtr &p):
        ps(p.ps), i(p.i), use(p.use) { ++*use; }
    HasPtr& operator=(const HasPtr&);
    ~HasPtr();
private:
    std::string *ps;
    int    i;
    std::size_t *use;    // member to keep track of how many objects share
*ps
};
```

Here, we've added a new data member named `use` that will keep track of how many objects share the same `string`. The constructor that takes a `string` allocates this counter and initializes it to `1`, indicating that there is one user of this object's `string` member.

Pointerlike Copy Members "Fiddle" the Reference Count

When we copy or assign a `HasPtr` object, we want the copy and the original to point to the same `string`. That is, when we copy a `HasPtr`, we'll copy `ps` itself, not the `string` to which `ps` points. When we make a copy, we also increment the counter associated with that `string`.

The copy constructor (which we defined inside the class) copies all three members from its given `HasPtr`. This constructor also increments the `use` member, indicating that there is another user for the `string` to which `ps` and `p.ps` point.

The destructor cannot unconditionally `delete ps`—there might be other objects pointing to that memory. Instead, the destructor decrements the reference count, indicating that one less object shares the `string`. If the counter goes to zero, then the destructor frees the memory to which both `ps` and `use` point:

Click here to view code image

```
HasPtr::~HasPtr()
{
    if (--*use == 0) {      // if the reference count goes to 0
        delete ps;          // delete the string
        delete use;         // and the counter
    }
}
```

The copy-assignment operator, as usual, does the work common to the copy constructor and to the destructor. That is, the assignment operator must increment the counter of the right-hand operand (i.e., the work of the copy constructor) and decrement the counter of the left-hand operand, deleting the memory used if appropriate (i.e., the work of the destructor).

Also, as usual, the operator must handle self-assignment. We do so by incrementing the count in `rhs` before decrementing the count in the left-hand object. That way if both objects are the same, the counter will have been incremented before we check to see if `ps` (and `use`) should be deleted:

Click here to view code image

```
HasPtr& HasPtr::operator=(const HasPtr &rhs)
{
    ++*rhs.use;   // increment the use count of the right-hand operand
    if (--*use == 0) {   // then decrement this object's counter
        delete ps;       // if no other users
        delete use;      // free this object's allocated members
    }
    ps = rhs.ps;         // copy data from rhs into this object
    i = rhs.i;
    use = rhs.use;
    return *this;        // return this object
}
```

Exercises Section 13.2.2

Exercise 13.27: Define your own reference-counted version of `HasPtr`.

Exercise 13.28: Given the following classes, implement a default constructor and the necessary copy-control members.

(a)

```
class TreeNode {
```

```
private:
    std::string value;
    int         count;
    TreeNode    *left;
    TreeNode    *right;
};
```

(b)

```
class BinStrTree {
      private:
           TreeNode *root;
      };
```

13.3. Swap

In addition to defining the copy-control members, classes that manage resources often also define a function named `swap` (§ 9.2.5, p. 339). Defining `swap` is particularly important for classes that we plan to use with algorithms that reorder elements (§ 10.2.3, p. 383). Such algorithms call `swap` whenever they need to exchange two elements.

If a class defines its own `swap`, then the algorithm uses that class-specific version. Otherwise, it uses the `swap` function defined by the library. Although, as usual, we don't know how `swap` is implemented, conceptually it's easy to see that swapping two objects involves a copy and two assignments. For example, code to swap two objects of our valuelike `HasPtr` class (§ 13.2.1, p. 511) might look something like:

Click here to view code image

```
HasPtr temp = v1; // make a temporary copy of the value of v1
v1 = v2;          // assign the value of v2 to v1
v2 = temp;        // assign the saved value of v1 to v2
```

This code copies the `string` that was originally in `v1` twice—once when the `HasPtr` copy constructor copies `v1` into `temp` and again when the assignment operator assigns `temp` to `v2`. It also copies the `string` that was originally in `v2` when it assigns `v2` to `v1`. As we've seen, copying a valuelike `HasPtr` allocates a new `string` and copies the `string` to which the `HasPtr` points.

In principle, none of this memory allocation is necessary. Rather than allocating new copies of the `string`, we'd like `swap` to swap the pointers. That is, we'd like swapping two `HasPtr`s to execute as:

Click here to view code image

```
string *temp = v1.ps; // make a temporary copy of the pointer in v1.ps
v1.ps = v2.ps;        // assign the pointer in v2.ps to v1.ps
```

```
    v2.ps = temp;           // assign the saved pointer in v1.ps to v2.ps
```

Writing Our Own swap Function

We can override the default behavior of swap by defining a version of swap that operates on our class. The typical implementation of swap is:

Click here to view code image

```
class HasPtr {
    friend void swap(HasPtr&, HasPtr&);
    // other members as in § 13.2.1 (p. 511)
};
inline
void swap(HasPtr &lhs, HasPtr &rhs)
{
    using std::swap;
    swap(lhs.ps, rhs.ps); // swap the pointers, not the string data
    swap(lhs.i, rhs.i);   // swap the int members
}
```

We start by declaring swap as a friend to give it access to HasPtr's (private) data members. Because swap exists to optimize our code, we've defined swap as an inline function (§ 6.5.2, p. 238). The body of swap calls swap on each of the data members of the given object. In this case, we first swap the pointers and then the int members of the objects bound to rhs and lhs.

Note

Unlike the copy-control members, swap is never necessary. However, defining swap can be an important optimization for classes that allocate resources.

swap Functions Should Call swap, Not std::swap

There is one important subtlety in this code: Although it doesn't matter in this particular case, it is essential that swap functions call swap and not std::swap. In the HasPtr function, the data members have built-in types. There is no type-specific version of swap for the built-in types. In this case, these calls will invoke the library std::swap.

However, if a class has a member that has its own type-specific swap function, calling std::swap would be a mistake. For example, assume we had another class named Foo that has a member named h, which has type HasPtr. If we did not write a Foo version of swap, then the library version of swap would be used. As we've already seen, the library swap makes unnecessary copies of the strings managed by HasPtr.

We can avoid these copies by writing a swap function for Foo. However, if we wrote the Foo version of swap as:

Click here to view code image

```
void swap(Foo &lhs, Foo &rhs)
{
    // WRONG: this function uses the library version of swap, not the HasPtr version
    std::swap(lhs.h, rhs.h);
    // swap other members of type Foo
}
```

this code would compile and execute. However, there would be no performance difference between this code and simply using the default version of swap. The problem is that we've explicitly requested the library version of swap. However, we don't want the version in std; we want the one defined for HasPtr objects.

The right way to write this swap function is:

Click here to view code image

```
void swap(Foo &lhs, Foo &rhs)
{
    using std::swap;
    swap(lhs.h, rhs.h); // uses the HasPtr version of swap
    // swap other members of type Foo
}
```

Each call to swap must be unqualified. That is, each call should be to swap, not std::swap. For reasons we'll explain in § 16.3 (p. 697), if there is a type-specific version of swap, that version will be a better match than the one defined in std. As a result, if there is a type-specific version of swap, calls to swap will match that type-specific version. If there is no type-specific version, then—assuming there is a using declaration for swap in scope—calls to swap will use the version in std.

Very careful readers may wonder why the using declaration inside swap does not hide the declarations for the HasPtr version of swap (§ 6.4.1, p. 234). We'll explain the reasons for why this code works in § 18.2.3 (p. 798).

Using swap in Assignment Operators

Classes that define swap often use swap to define their assignment operator. These operators use a technique known as **copy and swap**. This technique *swaps* the left-hand operand with a *copy* of the right-hand operand:

Click here to view code image

```
// note rhs is passed by value, which means the HasPtr copy constructor
// copies the string in the right-hand operand into rhs
HasPtr& HasPtr::operator=(HasPtr rhs)
{
    // swap the contents of the left-hand operand with the local variable rhs
    swap(*this, rhs); // rhs now points to the memory this object had used
    return *this;     // rhs is destroyed, which deletes the pointer in rhs
}
```

In this version of the assignment operator, the parameter is not a reference. Instead, we pass the right-hand operand by value. Thus, `rhs` is a copy of the right-hand operand. Copying a `HasPtr` allocates a new copy of that object's `string`.

In the body of the assignment operator, we call `swap`, which swaps the data members of `rhs` with those in `*this`. This call puts the pointer that had been in the left-hand operand into `rhs`, and puts the pointer that was in `rhs` into `*this`. Thus, after the `swap`, the pointer member in `*this` points to the newly allocated `string` that is a copy of the right-hand operand.

When the assignment operator finishes, `rhs` is destroyed and the `HasPtr` destructor is run. That destructor `delete`s the memory to which `rhs` now points, thus freeing the memory to which the left-hand operand had pointed.

The interesting thing about this technique is that it automatically handles self assignment and is automatically exception safe. By copying the right-hand operand before changing the left-hand operand, it handles self assignment in the same was as we did in our original assignment operator (§ 13.2.1, p. 512). It manages exception safety in the same way as the original definition as well. The only code that might throw is the `new` expression inside the copy constructor. If an exception occurs, it will happen before we have changed the left-hand operand.

Tip

Assignment operators that use copy and swap are automatically exception safe and correctly handle self-assignment.

Exercises Section 13.3

Exercise 13.29: Explain why the calls to `swap` inside `swap(HasPtr&, HasPtr&)` do not cause a recursion loop.

Exercise 13.30: Write and test a `swap` function for your valuelike version of `HasPtr`. Give your `swap` a print statement that notes when it is executed.

Exercise 13.31: Give your class a `<` operator and define a `vector` of `HasPtr`s. Give that `vector` some elements and then `sort` the `vector`.

Note when `swap` is called.

Exercise 13.32: Would the pointerlike version of `HasPtr` benefit from defining a `swap` function? If so, what is the benefit? If not, why not?

13.4. A Copy-Control Example

Although copy control is most often needed for classes that allocate resources, resource management is not the only reason why a class might need to define these members. Some classes have bookkeeping or other actions that the copy-control members must perform.

As an example of a class that needs copy control in order to do some bookkeeping, we'll sketch out two classes that might be used in a mail-handling application. These classes, `Message` and `Folder`, represent, respectively, email (or other kinds of) messages, and directories in which a message might appear. Each `Message` can appear in multiple `Folders`. However, there will be only one copy of the contents of any given `Message`. That way, if the contents of a `Message` are changed, those changes will appear when we view that `Message` from any of its `Folders`.

To keep track of which `Messages` are in which `Folders`, each `Message` will store a `set` of pointers to the `Folders` in which it appears, and each `Folder` will contain a `set` of pointers to its `Messages`. Figure 13.1 illustrates this design.

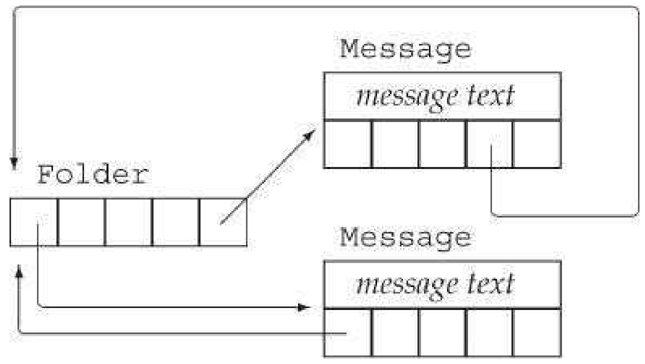


Figure 13.1. Message and Folder Class Design

Our `Message` class will provide `save` and `remove` operations to add or remove a `Message` from a specified `Folder`. To create a new `Message`, we will specify the contents of the message but no `Folder`. To put a `Message` in a particular `Folder`, we must call `save`.

When we copy a `Message`, the copy and the original will be distinct `Messages`, but both `Messages` should appear in the same `set` of `Folders`. Thus, copying a `Message` will copy the contents and the `set` of `Folder` pointers. It must also add a pointer to the newly created `Message` to each of those `Folders`.

When we destroy a `Message`, that `Message` no longer exists. Therefore, destroying a `Message` must remove pointers to that `Message` from the `Folders` that had contained that `Message`.

When we assign one `Message` to another, we'll replace the `contents` of the left-hand `Message` with those in the right-hand side. We must also update the `set` of `Folders`, removing the left-hand `Message` from its previous `Folders` and adding that `Message` to the `Folders` in which the right-hand `Message` appears.

Looking at this list of operations, we can see that both the destructor and the copy-assignment operator have to remove this `Message` from the `Folders` that point to it. Similarly, both the copy constructor and the copy-assignment operator add a `Message` to a given list of `Folders`. We'll define a pair of `private` utility functions to do these tasks.

Best Practices

The copy-assignment operator often does the same work as is needed in the copy constructor and destructor. In such cases, the common work should be put in `private` utility functions.

The `Folder` class will need analogous copy control members to add or remove itself from the `Messages` it stores.

We'll leave the design and implementation of the `Folder` class as an exercise. However, we'll assume that the `Folder` class has members named `addMsg` and `remMsg` that do whatever work is need to add or remove this `Message`, respectively, from the set of messages in the given `Folder`.

The Message Class

Given this design, we can write our `Message` class as follows:

Click here to view code image

```
class Message {
    friend class Folder;
public:
    // folders is implicitly initialized to the empty set
    explicit Message(const std::string &str = ""):
        contents(str) { }
    // copy control to manage pointers to this Message
    Message(const Message&);            // copy constructor
    Message& operator=(const Message&); // copy assignment
    ~Message();                          // destructor
    // add/remove this Message from the specified Folder's set of messages
    void save(Folder&);
    void remove(Folder&);
private:
    std::string contents;      // actual message text
    std::set<Folder*> folders; // Folders that have this Message
    // utility functions used by copy constructor, assignment, and destructor
    // add this Message to the Folders that point to the parameter
    void add_to_Folders(const Message&);
    // remove this Message from every Folder in folders
    void remove_from_Folders();
};
```

The class defines two data members: `contents`, to store the message text, and `folders`, to store pointers to the `Folder`s in which this `Message` appears. The constructor that takes a `string` copies the given `string` into `contents` and (implicitly) initializes `folders` to the empty set. Because this constructor has a default argument, it is also the `Message` default constructor (§ 7.5.1, p. 290).

The save and remove Members

Aside from copy control, the `Message` class has only two `public` members: `save`, which puts the `Message` in the given `Folder`, and `remove`, which takes it out:

Click here to view code image

```
void Message::save(Folder &f)
{
    folders.insert(&f); // add the given Folder to our list of Folders
    f.addMsg(this);     // add this Message to f's set of Messages
}
void Message::remove(Folder &f)
{
    folders.erase(&f); // take the given Folder out of our list of Folders
    f.remMsg(this);    // remove this Message to f's set of Messages
}
```

To save (or remove) a `Message` requires updating the `folders` member of the `Message`. When we `save` a `Message`, we store a pointer to the given `Folder`; when we `remove` a `Message`, we remove that pointer.

These operations must also update the given `Folder`. Updating a `Folder` is a job that the `Folder` class controls through its `addMsg` and `remMsg` members, which will add or remove a pointer to a given `Message`, respectively.

Copy Control for the Message Class

When we copy a `Message`, the copy should appear in the same `Folder`s as the original `Message`. As a result, we must traverse the `set` of `Folder` pointers adding a pointer to the new `Message` to each `Folder` that points to the original `Message`. Both the copy constructor and the copy-assignment operator will need to do this work, so we'll define a function to do this common processing:

Click here to view code image

```
// add this Message to Folders that point to m
void Message::add_to_Folders(const Message &m)
{
    for (auto f : m.folders) // for each Folder that holds m
        f->addMsg(this); // add a pointer to this Message to that Folder
}
```

Here we call `addMsg` on each `Folder` in `m.folders`. The `addMsg` function will add a pointer to this `Message` to that `Folder`.

The `Message` copy constructor copies the data members of the given object:

Click here to view code image

```
Message::Message(const Message &m):
    contents(m.contents), folders(m.folders)
{
    add_to_Folders(m); // add this Message to the Folders that point to m
}
```

and calls `add_to_Folders` to add a pointer to the newly created `Message` to each `Folder` that contains the original `Message`.

The `Message` Destructor

When a `Message` is destroyed, we must remove this `Message` from the `Folders` that point to it. This work is shared with the copy-assignment operator, so we'll define a common function to do it:

Click here to view code image

```
// remove this Message from the corresponding Folders
void Message::remove_from_Folders()
{
    for (auto f : folders) // for each pointer in folders
        f->remMsg(this);    // remove this Message from that Folder
}
```

The implementation of the `remove_from_Folders` function is similar to that of `add_to_Folders`, except that it uses `remMsg` to remove the current `Message`.

Given the `remove_from_Folders` function, writing the destructor is trivial:

Click here to view code image

```
Message::~Message()
{
    remove_from_Folders();
    }
```

The call to `remove_from_Folders` ensures that no `Folder` has a pointer to the `Message` we are destroying. The compiler automatically invokes the `string` destructor to free `contents` and the `set` destructor to clean up the memory used by those members.

`Message` Copy-Assignment Operator

In common with most assignment operators, our `Folder` copy-assignment operator must do the work of the copy constructor and the destructor. As usual, it is crucial that we structure our code to execute correctly even if the left- and right-hand operands happen to be the same object.

In this case, we protect against self-assignment by removing pointers to this `Message` from the `folders` of the left-hand operand before inserting pointers in the `folders` in the right-hand operand:

Click here to view code image

```
    Message& Message::operator=(const Message &rhs)
    {
        // handle self-assignment by removing pointers before inserting them
        remove_from_Folders();    // update existing Folders
        contents = rhs.contents; // copy message contents from rhs
        folders = rhs.folders;    // copy Folder pointers from rhs
        add_to_Folders(rhs);      // add this Message to those Folders
        return *this;
    }
```

If the left- and right-hand operands are the same object, then they have the same address. Had we called `remove_from_folders` after calling `add_to_folders`, we would have removed this `Message` from all of its corresponding `Folders`.

A `swap` Function for `Message`

The library defines versions of `swap` for both `string` and `set` (§ 9.2.5, p. 339). As a result, our `Message` class will benefit from defining its own version of `swap`. By defining a `Message`-specific version of `swap`, we can avoid extraneous copies of the `contents` and `folders` members.

However, our `swap` function must also manage the `Folder` pointers that point to the swapped `Messages`. After a call such as `swap(m1, m2)`, the `Folders` that had pointed to `m1` must now point to `m2`, and vice versa.

We'll manage the `Folder` pointers by making two passes through each of the `folders` members. The first pass will remove the `Messages` from their respective `Folders`. We'll next call `swap` to swap the data members. We'll make the second pass through `folders` this time adding pointers to the swapped `Messages`:

Click here to view code image

```
void swap(Message &lhs, Message &rhs)
{
    using std::swap; // not strictly needed in this case, but good habit
    // remove pointers to each Message from their (original) respective Folders
    for (auto f: lhs.folders)
        f->remMsg(&lhs);
    for (auto f: rhs.folders)
        f->remMsg(&rhs);
    // swap the contents and Folder pointer sets
    swap(lhs.folders, rhs.folders);     // uses swap(set&, set&)
    swap(lhs.contents, rhs.contents);   // swap(string&, string&)
    // add pointers to each Message to their (new) respective Folders
    for (auto f: lhs.folders)
        f->addMsg(&lhs);
    for (auto f: rhs.folders)
        f->addMsg(&rhs);
}
```

Exercises Section 13.4

Exercise 13.33: Why is the parameter to the `save` and `remove` members of `Message` a `Folder&`? Why didn't we define that parameter as `Folder`? Or `const Folder&`?

Exercise 13.34: Write the `Message` class as described in this section.

Exercise 13.35: What would happen if `Message` used the synthesized versions of the copy-control members?

Exercise 13.36: Design and implement the corresponding `Folder` class. That class should hold a `set` that points to the `Messages` in that `Folder`.

Exercise 13.37: Add members to the `Message` class to insert or remove a given `Folder*` into `folders`. These members are analogous to `Folder`'s `addMsg` and `remMsg` operations.

Exercise 13.38: We did not use copy and swap to define the `Message` assignment operator. Why do you suppose this is so?

13.5. Classes That Manage Dynamic Memory

Some classes need to allocate a varying amount of storage at run time. Such classes often can (and if they can, generally should) use a library container to hold their data. For example, our `StrBlob` class uses a `vector` to manage the underlying storage for its elements.

However, this strategy does not work for every class; some classes need to do their own allocation. Such classes generally must define their own copy-control members to manage the memory they allocate.

As an example, we'll implement a simplification of the library `vector` class. Among the simplifications we'll make is that our class will not be a template. Instead, our class will hold `string`s. Thus, we'll call our class `StrVec`.

StrVec Class Design

Recall that the `vector` class stores its elements in contiguous storage. To obtain acceptable performance, `vector` preallocates enough storage to hold more elements than are needed (§ 9.4, p. 355). Each `vector` member that adds elements checks whether there is space available for another element. If so, the member constructs an object in the next available spot. If there isn't space left, then the `vector` is reallocated: The `vector` obtains new space, moves the existing elements into that space, frees the old space, and adds the new element.

We'll use a similar strategy in our `StrVec` class. We'll use an `allocator` to obtain raw memory (§ 12.2.2, p. 481). Because the memory an `allocator` allocates is unconstructed, we'll use the `allocator`'s `construct` member to create objects in that space when we need to add an element. Similarly, when we remove an element, we'll use the `destroy` member to destroy the element.

Each `StrVec` will have three pointers into the space it uses for its elements:

- `elements`, which points to the first element in the allocated memory
- `first_free`, which points just after the last actual element
- `cap`, which points just past the end of the allocated memory

Figure 13.2 illustrates the meaning of these pointers.

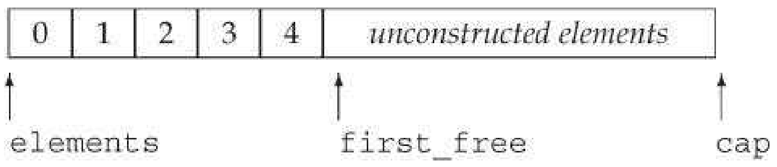


Figure 13.2. StrVec Memory Allocation Strategy

In addition to these pointers, `StrVec` will have a member named `alloc` that is an `allocator<string>`. The `alloc` member will allocate the memory used by a `StrVec`. Our class will also have four utility functions:

- `alloc_n_copy` will allocate space and copy a given range of elements.
- `free` will destroy the constructed elements and deallocate the space.
- `chk_n_alloc` will ensure that there is room to add at least one more element

to the `StrVec`. If there isn't room for another element, `chk_n_alloc` will call `reallocate` to get more space.

- `reallocate` will reallocate the `StrVec` when it runs out of space.

Although our focus is on the implementation, we'll also define a few members from `vector`'s interface.

StrVec Class Definition

Having sketched the implementation, we can now define our `StrVec` class:

Click here to view code image

```
// simplified implementation of the memory allocation strategy for a vector-like class
class StrVec {
public:
    StrVec(): // the allocator member is default initialized
       elements(nullptr), first_free(nullptr), cap(nullptr) {
}
    StrVec(const StrVec&);            // copy constructor
    StrVec &operator=(const StrVec&); // copy assignment
    ~StrVec();                        // destructor
    void push_back(const std::string&);  // copy the element
    size_t size() const { return first_free - elements; }
    size_t capacity() const { return cap - elements; }
    std::string *begin() const { return elements; }
    std::string *end() const { return first_free; }
    // ...
private:
    std::allocator<std::string> alloc; // allocates the elements
    // used by the functions that add elements to the StrVec
    void chk_n_alloc()
        { if (size() == capacity()) reallocate(); }
    // utilities used by the copy constructor, assignment operator, and destructor
    std::pair<std::string*, std::string*> alloc_n_copy
        (const std::string*, const std::string*);
    void free();             // destroy the elements and free the space
    void reallocate();        // get more space and copy the existing
elements
    std::string *elements;    // pointer to the first element in the array
    std::string *first_free; // pointer to the first free element in the array
    std::string *cap;         // pointer to one past the end of the array
};
```

The class body defines several of its members:

- The default constructor (implicitly) default initializes `alloc` and (explicitly) initializes the pointers to `nullptr`, indicating that there are no elements.

- The `size` member returns the number of elements actually in use, which is equal to `first_free - elements`.
- The `capacity` member returns the number of elements that the `StrVec` can hold, which is equal to `cap - elements`.
- The `chk_n_alloc` causes the `StrVec` to be reallocated when there is no room to add another element, which happens when `cap == first_free`.
- The `begin` and `end` members return pointers to the first (i.e., `elements`) and one past the last constructed element (i.e., `first_free`), respectively.

Using construct

The `push_back` function calls `chk_n_alloc` to ensure that there is room for an element. If necessary, `chk_n_alloc` will call `reallocate`. When `chk_n_alloc` returns, `push_back` knows that there is room for the new element. It asks its `allocator` member to `construct` a new last element:

Click here to view code image

```
void StrVec::push_back(const string& s)
{
    chk_n_alloc(); // ensure that there is room for another element
    // construct a copy of s in the element to which first_free points
    alloc.construct(first_free++, s);
}
```

When we use an `allocator` to allocate memory, we must remember that the memory is *unconstructed* (§ 12.2.2, p. 482). To use this raw memory we must call `construct`, which will construct an object in that memory. The first argument to `construct` must be a pointer to unconstructed space allocated by a call to `allocate`. The remaining arguments determine which constructor to use to construct the object that will go in that space. In this case, there is only one additional argument. That argument has type `string`, so this call uses the `string` copy constructor.

It is worth noting that the call to `construct` also increments `first_free` to indicate that a new element has been constructed. It uses the postfix increment (§ 4.5, p. 147), so this call constructs an object in the current value of `first_free` and increments `first_free` to point to the next, unconstructed element.

The alloc_n_copy Member

The `alloc_n_copy` member is called when we copy or assign a `StrVec`. Our `StrVec` class, like `vector`, will have valuelike behavior (§ 13.2.1, p. 511); when we copy or assign a `StrVec`, we have to allocate independent memory and copy the elements from the original to the new `StrVec`.

The alloc_n_copy member will allocate enough storage to hold its given range of elements, and will copy those elements into the newly allocated space. This function returns a pair (§ 11.2.3, p. 426) of pointers, pointing to the beginning of the new space and just past the last element it copied:

Click here to view code image

```
pair<string*, string*>
StrVec::alloc_n_copy(const string *b, const string *e)
{
    // allocate space to hold as many elements as are in the range
    auto data = alloc.allocate(e - b);
    // initialize and return a pair constructed from data and
    // the value returned by uninitialized_copy
    return {data, uninitialized_copy(b, e, data)};
}
```

alloc_n_copy calculates how much space to allocate by subtracting the pointer to the first element from the pointer one past the last. Having allocated memory, the function next has to construct copies of the given elements in that space.

It does the copy in the return statement, which list initializes the return value (§ 6.3.2, p. 226). The first member of the returned pair points to the start of the allocated memory; the second is the value returned from uninitialized_copy (§ 12.2.2, p. 483). That value will be pointer positioned one element past the last constructed element.

The free Member

The free member has two responsibilities: It must destroy the elements and then deallocate the space that this StrVec itself allocated. The for loop calls the allocator member destroy in reverse order, starting with the last constructed element and finishing with the first:

Click here to view code image

```
void StrVec::free()
{
    // may not pass deallocate a 0 pointer; if elements is 0, there's no work to do
    if (elements) {
        // destroy the old elements in reverse order
        for (auto p = first_free; p != elements; /* empty */)
            alloc.destroy(--p);
        alloc.deallocate(elements, cap - elements);
    }
}
```

The destroy function runs the string destructor. The string destructor frees whatever storage was allocated by the strings themselves.

Once the elements have been destroyed, we free the space that this `StrVec` allocated by calling `deallocate`. The pointer we pass to `deallocate` must be one that was previously generated by a call to `allocate`. Therefore, we first check that `elements` is not null before calling `deallocate`.

Copy-Control Members

Given our `alloc_n_copy` and `free` members, the copy-control members of our class are straightforward. The copy constructor calls `alloc_n_copy`:

Click here to view code image

```
StrVec::StrVec(const StrVec &s)
{
    // call alloc_n_copy to allocate exactly as many elements as in s
    auto newdata = alloc_n_copy(s.begin(), s.end());
    elements = newdata.first;
    first_free = cap = newdata.second;
}
```

and assigns the results from that call to the data members. The return value from `alloc_n_copy` is a `pair` of pointers. The `first` pointer points to the first constructed element and the `second` points just past the last constructed element. Because `alloc_n_copy` allocates space for exactly as many elements as it is given, `cap` also points just past the last constructed element.

The destructor calls `free`:

Click here to view code image

```
StrVec::~StrVec() { free(); }
```

The copy-assignment operator calls `alloc_n_copy` before freeing its existing elements. By doing so it protects against self-assignment:

Click here to view code image

```
StrVec &StrVec::operator=(const StrVec &rhs)
{
    // call alloc_n_copy to allocate exactly as many elements as in rhs
    auto data = alloc_n_copy(rhs.begin(), rhs.end());
    free();
    elements = data.first;
    first_free = cap = data.second;
    return *this;
}
```

Like the copy constructor, the copy-assignment operator uses the values returned from `alloc_n_copy` to initialize its pointers.

Moving, Not Copying, Elements during Reallocation

Before we write the `reallocate` member, we should think a bit about what it must do. This function will

- Allocate memory for a new, larger array of `strings`
- Construct the first part of that space to hold the existing elements
- Destroy the elements in the existing memory and deallocate that memory

Looking at this list of steps, we can see that reallocating a `StrVec` entails copying each `string` from the old `StrVec` memory to the new. Although we don't know the details of how `string` is implemented, we do know that `strings` have valuelike behavior. When we copy a `string`, the new `string` and the original `string` are independent from each other. Changes made to the original don't affect the copy, and vice versa.

Because `strings` act like values, we can conclude that each `string` must have its own copy of the characters that make up that `string`. Copying a `string` must allocate memory for those characters, and destroying a `string` must free the memory used by that `string`.

Copying a `string` copies the data because ordinarily after we copy a `string`, there are two users of that `string`. However, when `reallocate` copies the `strings` in a `StrVec`, there will be only one user of these `strings` after the copy. As soon as we copy the elements from the old space to the new, we will immediately destroy the original `strings`.

Copying the data in these `strings` is unnecessary. Our `StrVec`'s performance will be *much* better if we can avoid the overhead of allocating and deallocating the `strings` themselves each time we reallocate.

Move Constructors and **std::move**

We can avoid copying the `strings` by using two facilities introduced by the new library. First, several of the library classes, including `string`, define so-called "move constructors." The details of how the `string` move constructor works—like any other detail about the implementation—are not disclosed. However, we do know that move constructors typically operate by "moving" resources from the given object to the object being constructed. We also know that the library guarantees that the "moved-from" `string` remains in a valid, destructible state. For `string`, we can imagine that each `string` has a pointer to an array of `char`. Presumably the `string` move constructor copies the pointer rather than allocating space for and copying the characters themselves.

The second facility we'll use is a library function named **move**, which is defined in the `utility` header. For now, there are two important points to know about `move`.

First, for reasons we'll explain in § 13.6.1 (p. 532), when `reallocate` constructs the `strings` in the new memory it must call `move` to signal that it wants to use the `string` move constructor. If it omits the call to `move` the `string` the copy constructor will be used. Second, for reasons we'll cover in § 18.2.3 (p. 798), we usually do not provide a `using` declaration (§ 3.1, p. 82) for `move`. When we use `move`, we call `std::move`, not `move`.

The reallocate Member

Using this information, we can now write our `reallocate` member. We'll start by calling `allocate` to allocate new space. We'll double the capacity of the `StrVec` each time we reallocate. If the `StrVec` is empty, we allocate room for one element:

Click here to view code image

```
void StrVec::reallocate()
{
     // we'll allocate space for twice as many elements as the current size
     auto newcapacity = size() ? 2 * size() : 1;
     // allocate new memory
     auto newdata = alloc.allocate(newcapacity);
     // move the data from the old memory to the new
     auto dest = newdata;  // points to the next free position in the new array
     auto elem = elements; // points to the next element in the old array
     for (size_t i = 0; i != size(); ++i)
         alloc.construct(dest++, std::move(*elem++));
     free();  // free the old space once we've moved the elements
     // update our data structure to point to the new elements
     elements = newdata;
     first_free = dest;
     cap = elements + newcapacity;
}
```

The `for` loop iterates through the existing elements and `constructs` a corresponding element in the new space. We use `dest` to point to the memory in which to construct the new `string`, and use `elem` to point to an element in the original array. We use postfix increment to move the `dest` (and `elem`) pointers one element at a time through these two arrays.

The second argument in the call to `construct` (i.e., the one that determines which constructor to use (§ 12.2.2, p. 482)) is the value returned by `move`. Calling `move` returns a result that causes `construct` to use the `string` move constructor. Because we're using the move constructor, the memory managed by those `strings` will not be copied. Instead, each `string` we construct will take over ownership of the memory from the `string` to which `elem` points.

After moving the elements, we call `free` to destroy the old elements and free the memory that this `StrVec` was using before the call to `reallocate`. The `strings` themselves no longer manage the memory to which they had pointed; responsibility for their data has been moved to the elements in the new `StrVec` memory. We don't know what value the `strings` in the old `StrVec` memory have, but we are guaranteed that it is safe to run the `string` destructor on these objects.

What remains is to update the pointers to address the newly allocated and initialized array. The `first_free` and `cap` pointers are set to denote one past the last constructed element and one past the end of the allocated space, respectively.

Exercises Section 13.5

Exercise 13.39: Write your own version of `StrVec`, including versions of `reserve`, `capacity` (§ 9.4, p. 356), and `resize` (§ 9.3.5, p. 352).

Exercise 13.40: Add a constructor that takes an `initializer_list<string>` to your `StrVec` class.

Exercise 13.41: Why did we use postfix increment in the call to `construct` inside `push_back`? What would happen if it used the prefix increment?

Exercise 13.42: Test your `StrVec` class by using it in place of the `vector<string>` in your `TextQuery` and `QueryResult` classes (§ 12.3, p. 484).

Exercise 13.43: Rewrite the `free` member to use `for_each` and a lambda (§ 10.3.2, p. 388) in place of the `for` loop to `destroy` the elements. Which implementation do you prefer, and why?

Exercise 13.44: Write a class named `String` that is a simplified version of the library `string` class. Your class should have at least a default constructor and a constructor that takes a pointer to a C-style string. Use an `allocator` to allocate memory that your `String` class uses.

13.6. Moving Objects

One of the major features in the new standard is the ability to move rather than copy an object. As we saw in § 13.1.1 (p. 497), copies are made in many circumstances. In some of these circumstances, an object is immediately destroyed after it is copied. In those cases, moving, rather than copying, the object can provide a significant performance boost.

As we've just seen, our `StrVec` class is a good example of this kind of superfluous copy. During reallocation, there is no need to copy—rather than move—the elements from the old memory to the new. A second reason to move rather than copy occurs in classes such as the IO or `unique_ptr` classes. These classes have a resource (such as a pointer or an IO buffer) that may not be shared. Hence, objects of these types can't be copied but can be moved.

Under earlier versions of the language, there was no direct way to move an object. We had to make a copy even if there was no need for the copy. If the objects are large, or if the objects themselves require memory allocation (e.g., `string`s), making a needless copy can be expensive. Similarly, in previous versions of the library, classes stored in a container had to be copyable. Under the new standard, we can use containers on types that cannot be copied so long as they can be moved.

Note

The library containers, `string`, and `shared_ptr` classes support move as well as copy. The IO and `unique_ptr` classes can be moved but not copied.

13.6.1. Rvalue References

To support move operations, the new standard introduced a new kind of reference, an **rvalue reference**. An rvalue reference is a reference that must be bound to an rvalue. An rvalue reference is obtained by using `&&` rather than `&`. As we'll see, rvalue references have the important property that they may be bound only to an object that is about to be destroyed. As a result, we are free to "move" resources from an rvalue reference to another object.

Recall that lvalue and rvalue are properties of an expression (§ 4.1.1, p. 135). Some expressions yield or require lvalues; others yield or require rvalues. Generally speaking, an lvalue expression refers to an object's identity whereas an rvalue expression refers to an object's value.

Like any reference, an rvalue reference is just another name for an object. As we know, we cannot bind regular references—which we'll refer to as **lvalue references** when we need to distinguish them from rvalue references—to expressions that require a conversion, to literals, or to expressions that return an rvalue (§ 2.3.1, p. 51). Rvalue references have the opposite binding properties: We can bind an rvalue reference to these kinds of expressions, but we cannot directly bind an rvalue reference to an lvalue:

Click here to view code image

```
int i = 42;
int &r = i;             // ok: r refers to i
int &&rr = i;           // error: cannot bind an rvalue reference to an lvalue
int &r2 = i * 42;          // error: i * 42 is an rvalue
const int &r3 = i * 42; // ok: we can bind a reference to const to an rvalue
int &&rr2 = i * 42;        // ok: bind rr2 to the result of the multiplication
```

Functions that return lvalue references, along with the assignment, subscript, dereference, and prefix increment/decrement operators, are all examples of expressions that return lvalues. We can bind an lvalue reference to the result of any of these expressions.

Functions that return a nonreference type, along with the arithmetic, relational, bitwise, and postfix increment/decrement operators, all yield rvalues. We cannot bind an lvalue reference to these expressions, but we can bind either an lvalue reference to `const` or an rvalue reference to such expressions.

Lvalues Persist; Rvalues Are Ephemeral

Looking at the list of lvalue and rvalue expressions, it should be clear that lvalues and rvalues differ from each other in an important manner: Lvalues have persistent state, whereas rvalues are either literals or temporary objects created in the course of evaluating expressions.

Because rvalue references can only be bound to temporaries, we know that

- The referred-to object is about to be destroyed
- There can be no other users of that object

These facts together mean that code that uses an rvalue reference is free to take over resources from the object to which the reference refers.

Note

Rvalue references refer to objects that are about to be destroyed. Hence, we can “steal” state from an object bound to an rvalue reference.

Variables Are Lvalues

Although we rarely think about it this way, a variable is an expression with one operand and no operator. Like any other expression, a variable expression has the lvalue/rvalue property. Variable expressions are lvalues. It may be surprising, but as a consequence, we cannot bind an rvalue reference to a variable defined as an rvalue reference type:

Click here to view code image

```
int &&rr1 = 42;     // ok: literals are rvalues
int &&rr2 = rr1;    // error: the expression rr1 is an lvalue!
```

Given our previous observation that rvalues represent ephemeral objects, it should not be surprising that a variable is an lvalue. After all, a variable persists until it goes out of scope.

Note

A variable is an lvalue; we cannot directly bind an rvalue reference to a variable *even if that variable was defined as an rvalue reference type.*

The Library move Function

C++ 11

Although we cannot directly bind an rvalue reference to an lvalue, we can explicitly cast an lvalue to its corresponding rvalue reference type. We can also obtain an rvalue reference bound to an lvalue by calling a new library function named **move**, which is defined in the `utility` header. The `move` function uses facilities that we'll describe in § 16.2.6 (p. 690) to return an rvalue reference to its given object.

Click here to view code image

```
int &&rr3 = std::move(rr1);    // ok
```

Calling `move` tells the compiler that we have an lvalue that we want to treat as if it were an rvalue. It is essential to realize that the call to `move` promises that we do not intend to use `rr1` again except to assign to it or to destroy it. After a call to `move`, we cannot make any assumptions about the value of the moved-from object.

Note

We can destroy a moved-from object and can assign a new value to it, but we cannot use the value of a moved-from object.

As we've seen, differently from how we use most names from the library, we do not provide a `using` declaration (§ 3.1, p. 82) for move (§ 13.5, p. 530). We call `std::move` not `move`. We'll explain the reasons for this usage in § 18.2.3 (p. 798).

Warning

Code that uses `move` should use `std::move`, not `move`. Doing so avoids potential name collisions.

Exercises Section 13.6.1

Exercise 13.45: Distinguish between an rvalue reference and an lvalue reference.

Exercise 13.46: Which kind of reference can be bound to the following initializers?

```
int f();
vector<int> vi(100);
int? r1 = f();
int? r2 = vi[0];
int? r3 = r1;
int? r4 = vi[0] * f();
```

Exercise 13.47: Give the copy constructor and copy-assignment operator in your `String` class from exercise 13.44 in § 13.5 (p. 531) a statement that prints a message each time the function is executed.

Exercise 13.48: Define a `vector<String>` and call `push_back` several times on that `vector`. Run your program and see how often `String`s are copied.

13.6.2. Move Constructor and Move Assignment

Like the `string` class (and other library classes), our own classes can benefit from being able to be moved as well as copied. To enable move operations for our own types, we define a move constructor and a move-assignment operator. These members are similar to the corresponding copy operations, but they "steal" resources from their given object rather than copy them.

Like the copy constructor, the move constructor has an initial parameter that is a reference to the class type. Differently from the copy constructor, the reference parameter in the move constructor is an rvalue reference. As in the copy constructor, any additional parameters must all have default arguments.

In addition to moving resources, the move constructor must ensure that the moved-from object is left in a state such that destroying that object will be harmless. In particular, once its resources are moved, the original object must no longer point to those moved resources—responsibility for those resources has been assumed by the newly created object.

As an example, we'll define the `StrVec` move constructor to move rather than copy the elements from one `StrVec` to another:

Click here to view code image

```
StrVec::StrVec(StrVec &&s) noexcept   // move won't throw any exceptions
  // member initializers take over the resources in s
      : elements(s.elements), first_free(s.first_free), cap(s.cap)
{
    // leave s in a state in which it is safe to run the destructor
    s.elements = s.first_free = s.cap = nullptr;
}
```

We'll explain the use of `noexcept` (which signals that our constructor does not throw any exceptions) shortly, but let's first look at what this constructor does.

Unlike the copy constructor, the move constructor does not allocate any new memory; it takes over the memory in the given `StrVec`. Having taken over the memory from its argument, the constructor body sets the pointers in the given object to `nullptr`. After an object is moved from, that object continues to exist. Eventually, the moved-from object will be destroyed, meaning that the destructor will be run on that object. The `StrVec` destructor calls `deallocate` on `first_free`. If we neglected to change `s.first_free`, then destroying the moved-from object would delete the memory we just moved.

Move Operations, Library Containers, and Exceptions

Because a move operation executes by "stealing" resources, it ordinarily does not itself allocate any resources. As a result, move operations ordinarily will not throw any exceptions. When we write a move operation that cannot throw, we should inform the library of that fact. As we'll see, unless the library knows that our move constructor won't throw, it will do extra work to cater to the possibliity that moving an object of our class type might throw.

One way inform the library is to specify `noexcept` on our constructor. We'll cover `noexcept`, which was introduced by the new standard, in more detail in § 18.1.4 (p. 779). For now what's important to know is that `noexcept` is a way for us to promise that a function does not throw any exceptions. We specify `noexcept` on a function after its parameter list. In a constructor, `noexcept` appears between the parameter list and the `:` that begins the constructor initializer list:

Click here to view code image

```
class StrVec {
public:
    StrVec(StrVec&&) noexcept;      // move constructor
    // other members as before
};
StrVec::StrVec(StrVec &&s) noexcept : /* member initializers */
{ /* constructor body   */ }
```

We must specify `noexcept` on both the declaration in the class header and on the definition if that definition appears outside the class.

Note

Move constructors and move assignment operators that cannot throw exceptions should be marked as `noexcept`.

Understanding why `noexcept` is needed can help deepen our understanding of how the library interacts with objects of the types we write. We need to indicate that a move operation doesn't throw because of two interrelated facts: First, although move operations usually don't throw exceptions, they are permitted to do so. Second, the library containers provide guarantees as to what they do if an exception happens. As one example, `vector` guarantees that if an exception happens when we call `push_back`, the `vector` itself will be left unchanged.

Now let's think about what happens inside `push_back`. Like the corresponding `StrVec` operation (§ 13.5, p. 527), `push_back` on a `vector` might require that the `vector` be reallocated. When a `vector` is reallocated, it moves the elements from its old space to new memory, just as we did in `reallocate` (§ 13.5, p. 530).

As we've just seen, moving an object generally changes the value of the moved-from object. If reallocation uses a move constructor and that constructor throws an exception after moving some but not all of the elements, there would be a problem. The moved-from elements in the old space would have been changed, and the unconstructed elements in the new space would not yet exist. In this case, `vector` would be unable to meet its requirement that the `vector` is left unchanged.

On the other hand, if `vector` uses the copy constructor and an exception happens, it can easily meet this requirement. In this case, while the elements are being constructed in the new memory, the old elements remain unchanged. If an exception happens, `vector` can free the space it allocated (but could not successfully construct) and return. The original `vector` elements still exist.

To avoid this potential problem, `vector` must use a copy constructor instead of a move constructor during reallocation *unless it knows* that the element type's move constructor cannot throw an exception. If we want objects of our type to be moved rather than copied in circumstances such as `vector` reallocation, we must explicity tell the library that our move constructor is safe to use. We do so by marking the move constructor (and move-assignment operator) `noexcept`.

Move-Assignment Operator

The move-assignment operator does the same work as the destructor and the move constructor. As with the move constructor, if our move-assignment operator won't throw any exceptions, we should make it `noexcept`. Like a copy-assignment operator, a move-assignment operator must guard against self-assignment:

Click here to view code image

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    // direct test for self-assignment
    if (this != &rhs) {
        free();                  // free existing elements
        elements = rhs.elements; // take over resources from rhs
        first_free = rhs.first_free;
        cap = rhs.cap;
        // leave rhs in a destructible state
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
    return *this;
}
```

In this case we check directly whether the `this` pointer and the address of `rhs` are the same. If they are, the right- and left-hand operands refer to the same object and there is no work to do. Otherwise, we free the memory that the left-hand operand had used, and then take over the memory from the given object. As in the move constructor, we set the pointers in `rhs` to `nullptr`.

It may seem surprising that we bother to check for self-assignment. After all, move assignment requires an rvalue for the right-hand operand. We do the check because that rvalue could be the result of calling `move`. As in any other assignment operator, it is crucial that we not free the left-hand resources before using those (possibly same) resources from the right-hand operand.

A Moved-from Object Must Be Destructible

Moving from an object does not destroy that object: Sometime after the move operation completes, the moved-from object will be destroyed. Therefore, when we write a move operation, we must ensure that the moved-from object is in a state in which the destructor can be run. Our `StrVec` move operations meet this requirement by setting the pointer members of the moved-from object to `nullptr`.

In addition to leaving the moved-from object in a state that is safe to destroy, move operations must guarantee that the object remains valid. In general, a valid object is one that can safely be given a new value or used in other ways that do not depend on its current value. On the other hand, move operations have no requirements as to the value that remains in the moved-from object. As a result, our programs should never depend on the value of a moved-from object.

For example, when we move from a library `string` or container object, we know that the moved-from object remains valid. As a result, we can run operations such as as `empty` or `size` on moved-from objects. However, we don't know what result we'll get. We might expect a moved-from object to be empty, but that is not guaranteed.

Our `StrVec` move operations leave the moved-from object in the same state as a default-initialized object. Therefore, all the operations of `StrVec` will continue to run the same way as they do for any other default-initialized `StrVec`. Other classes, with more complicated internal structure, may behave differently.

Warning

After a move operation, the "moved-from" object must remain a valid, destructible object but users may make no assumptions about its value.

The Synthesized Move Operations

As it does for the copy constructor and copy-assignment operator, the compiler will synthesize the move constructor and move-assignment operator. However, the conditions under which it synthesizes a move operation are quite different from those in which it synthesizes a copy operation.

Recall that if we do not declare our own copy constructor or copy-assignment operator the compiler *always* synthesizes these operations (§ 13.1.1, p. 497 and § 13.1.2, p. 500). The copy operations are defined either to memberwise copy or assign the object or they are defined as deleted functions.

Differently from the copy operations, for some classes the compiler does not synthesize the move operations *at all.* In particular, if a class defines its own copy constructor, copy-assignment operator, or destructor, the move constructor and move-assignment operator are not synthesized. As a result, some classes do not have a move constructor or a move-assignment operator. As we'll see on page 540, when a class doesn't have a move operation, the corresponding copy operation is used in place of move through normal function matching.

The compiler will synthesize a move constructor or a move-assignment operator *only* if the class doesn't define any of its own copy-control members and if every `nonstatic` data member of the class can be moved. The compiler can move members of built-in type. It can also move members of a class type if the member's class has the corresponding move operation:

Click here to view code image

```
// the compiler will synthesize the move operations for X and hasX
struct X {
    int i;           // built-in types can be moved
    std::string s; // string defines its own move operations
};
struct hasX {
    X mem;   // X has synthesized move operations
};
X x, x2 = std::move(x);        // uses the synthesized move constructor
hasX hx, hx2 = std::move(hx); // uses the synthesized move constructor
```

Note

The compiler synthesizes the move constructor and move assignment only if a class does not define any of its own copy-control members and only if all the data members can be moved constructed and move assigned, respectively.

Unlike the copy operations, a move operation is never implicitly defined as a deleted function. However, if we explicitly ask the compiler to generate a move operation by using `= default` (§ 7.1.4, p. 264), and the compiler is unable to move all the members, then the move operation will be defined as deleted. With one important exception, the rules for when a synthesized move operation is defined as deleted are analogous to those for the copy operations (§ 13.1.6, p. 508):

- Unlike the copy constructor, the move constructor is defined as deleted if the class has a member that defines its own copy constructor but does not also define a move constructor, or if the class has a member that doesn't define its own copy operations and for which the compiler is unable to synthesize a move constructor. Similarly for move-assignment.
- The move constructor or move-assignment operator is defined as deleted if the class has a member whose own move constructor or move-assignment operator is deleted or inaccessible.
- Like the copy constructor, the move constructor is defined as deleted if the destructor is deleted or inaccessible.
- Like the copy-assignment operator, the move-assignment operator is defined as deleted if the class has a `const` or reference member.

For example, assuming `Y` is a class that defines its own copy constructor but does not also define its own move constructor:

Click here to view code image

```
// assume Y is a class that defines its own copy constructor but not a move constructor
```

```
struct hasY {
    hasY() = default;
    hasY(hasY&&) = default;
    Y mem; // hasY will have a deleted move constructor
};
hasY hy, hy2 = std::move(hy); // error: move constructor is deleted
```

The compiler can copy objects of type `Y` but cannot move them. Class `hasY` explicitly requested a move constructor, which the compiler is unable to generate. Hence, `hasY` will get a deleted move constructor. Had `hasY` omitted the declaration of its move constructor, then the compiler would not synthesize the `hasY` move constructor at all. The move operations are not synthesized if they would otherwise be defined as deleted.

There is one final interaction between move operations and the synthesized copy-control members: Whether a class defines its own move operations has an impact on how the copy operations are synthesized. If the class defines either a move constructor and/or a move-assignment operator, then the synthesized copy constructor and copy-assignment operator for that class will be defined as deleted.

Note

Classes that define a move constructor or move-assignment operator must also define their own copy operations. Otherwise, those members are deleted by default.

Rvalues Are Moved, Lvalues Are Copied ...

When a class has both a move constructor and a copy constructor, the compiler uses ordinary function matching to determine which constructor to use (§ 6.4, p. 233). Similarly for assignment. For example, in our `StrVec` class the copy versions take a reference to `const StrVec`. As a result, they can be used on any type that can be converted to `StrVec`. The move versions take a `StrVec&&` and can be used only when the argument is a (non`const`) rvalue:

Click here to view code image

```
StrVec v1, v2;
v1 = v2;                      // v2 is an lvalue; copy assignment
StrVec getVec(istream &);     // getVec returns an rvalue
v2 = getVec(cin);             // getVec(cin) is an rvalue; move assignment
```

In the first assignment, we pass `v2` to the assignment operator. The type of `v2` is `StrVec` and the expression, `v2`, is an lvalue. The move version of assignment is not viable (§ 6.6, p. 243), because we cannot implicitly bind an rvalue reference to an lvalue. Hence, this assignment uses the copy-assignment operator.

In the second assignment, we assign from the result of a call to `getVec`. That expression is an rvalue. In this case, both assignment operators are viable—we can bind the result of `getVec` to either operator's parameter. Calling the copy-assignment operator requires a conversion to `const`, whereas `StrVec&&` is an exact match. Hence, the second assignment uses the move-assignment operator.

...But Rvalues Are Copied If There Is No Move Constructor

What if a class has a copy constructor but does not define a move constructor? In this case, the compiler will not synthesize the move constructor, which means the class has a copy constructor but no move constructor. If a class has no move constructor, function matching ensures that objects of that type are copied, even if we attempt to move them by calling `move`:

Click here to view code image

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo&);   // copy constructor
    // other members, but Foo does not define a move constructor
};
Foo x;
Foo y(x);             // copy constructor; x is an lvalue
Foo z(std::move(x)); // copy constructor, because there is no move constructor
```

The call to `move(x)` in the initialization of `z` returns a `Foo&&` bound to `x`. The copy constructor for `Foo` is viable because we can convert a `Foo&&` to a `const Foo&`. Thus, the initialization of `z` uses the copy constructor for `Foo`.

It is worth noting that using the copy constructor in place of a move constructor is almost surely safe (and similarly for the assignment operators). Ordinarily, the copy constructor will meet the requirements of the corresponding move constructor: It will copy the given object and leave that original object in a valid state. Indeed, the copy constructor won't even change the value of the original object.

Note

If a class has a usable copy constructor and no move constructor, objects will be "moved" by the copy constructor. Similarly for the copy-assignment operator and move-assignment.

Copy-and-Swap Assignment Operators and Move

The version of our `HasPtr` class that defined a copy-and-swap assignment operator

(§ 13.3, p. 518) is a good illustration of the interaction between function matching and move operations. If we add a move constructor to this class, it will effectively get a move assignment operator as well:

Click here to view code image

```
class HasPtr {
public:
    // added move constructor
     HasPtr(HasPtr &&p) noexcept : ps(p.ps), i(p.i) {p.ps =
0;}
    // assignment operator is both the move- and copy-assignment operator
    HasPtr& operator=(HasPtr rhs)
                   { swap(*this, rhs); return *this; }
    // other members as in § 13.2.1 (p. 511)
};
```

In this version of the class, we've added a move constructor that takes over the values from its given argument. The constructor body sets the pointer member of the given `HasPtr` to zero to ensure that it is safe to destroy the moved-from object. Nothing this function does can throw an exception so we mark it as `noexcept` (§ 13.6.2, p. 535).

Now let's look at the assignment operator. That operator has a nonreference parameter, which means the parameter is copy initialized (§ 13.1.1, p. 497). Depending on the type of the argument, copy initialization uses either the copy constructor or the move constructor; lvalues are copied and rvalues are moved. As a result, this single assignment operator acts as both the copy-assignment and move-assignment operator.

For example, assuming both `hp` and `hp2` are `HasPtr` objects:

Click here to view code image

```
hp = hp2; //  hp2 is an lvalue; copy constructor used to copy hp2
hp = std::move(hp2); // move constructor moves hp2
```

In the first assignment, the right-hand operand is an lvalue, so the move constructor is not viable. The copy constructor will be used to initialize `rhs`. The copy constructor will allocate a new `string` and copy the `string` to which `hp2` points.

In the second assignment, we invoke `std::move` to bind an rvalue reference to `hp2`. In this case, both the copy constructor and the move constructor are viable. However, because the argument is an rvalue reference, it is an exact match for the move constructor. The move constructor copies the pointer from `hp2`. It does not allocate any memory.

Regardless of whether the copy or move constructor was used, the body of the assignment operator `swap`s the state of the two operands. Swapping a `HasPtr` exchanges the pointer (and `int`) members of the two objects. After the `swap`, `rhs` will hold a pointer to the `string` that had been owned by the left-hand side. That `string` will be destroyed when `rhs` goes out of scope.

Advice: Updating the Rule of Three

All five copy-control members should be thought of as a unit: Ordinarily, if a class defines any of these operations, it usually should define them all. As we've seen, some classes *must* define the copy constructor, copy-assignment operator, and destructor to work correctly (§ 13.1.4, p. 504). Such classes typically have a resource that the copy members must copy. Ordinarily, copying a resource entails some amount of overhead. Classes that define the move constructor and move-assignment operator can avoid this overhead in those circumstances where a copy isn't necessary.

Move Operations for the `Message` Class

Classes that define their own copy constructor and copy-assignment operator generally also benefit by defining the move operations. For example, our `Message` and `Folder` classes (§ 13.4, p. 519) should define move operations. By defining move operations, the `Message` class can use the `string` and `set` move operations to avoid the overhead of copying the `contents` and `folders` members.

However, in addition to moving the `folders` member, we must also update each `Folder` that points to the original `Message`. We must remove pointers to the old `Message` and add a pointer to the new one.

Both the move constructor and move-assignment operator need to update the `Folder` pointers, so we'll start by defining an operation to do this common work:

Click here to view code image

```
// move the Folder pointers from m to this Message
void Message::move_Folders(Message *m)
{
    folders = std::move(m->folders); // uses set move assignment
    for (auto f : folders) {  // for each Folder
        f->remMsg(m);     // remove the old Message from the Folder
        f->addMsg(this); // add this Message to that Folder
    }
    m->folders.clear();   // ensure that destroying m is harmless
}
```

This function begins by moving the `folders` set. By calling `move`, we use the `set` move assignment rather than its copy assignment. Had we omitted the call to `move`, the code would still work, but the copy is unnecessary. The function then iterates through those `Folders`, removing the pointer to the original `Message` and adding a pointer to the new `Message`.

It is worth noting that inserting an element to a `set` might throw an exception—adding an element to a container requires memory to be allocated, which means that a `bad_alloc` exception might be thrown (§ 12.1.2, p. 460). As a result, unlike our `HasPtr` and `StrVec` move operations, the `Message` move constructor and move-assignment operators might throw exceptions. We will not mark them as `noexcept` (§ 13.6.2, p. 535).

The function ends by calling `clear` on `m.folders`. After the `move`, we know that `m.folders` is valid but have no idea what its contents are. Because the `Message` destructor iterates through `folders`, we want to be certain that the `set` is empty.

The `Message` move constructor calls `move` to move the `contents` and default initializes its `folders` member:

Click here to view code image

```
Message::Message(Message &&m):
contents(std::move(m.contents))
{
    move_Folders(&m); // moves folders and updates the Folder pointers
}
```

In the body of the constructor, we call `move_Folders` to remove the pointers to `m` and insert pointers to this `Message`.

The move-assignment operator does a direct check for self-assignment:

Click here to view code image

```
Message& Message::operator=(Message &&rhs)
{
    if (this != &rhs) {        // direct check for self-assignment
        remove_from_Folders();
        contents = std::move(rhs.contents); // move assignment
        move_Folders(&rhs); // reset the Folders to point to this Message
    }
    return *this;
}
```

As with any assignment operator, the move-assignment operator must destroy the old state of the left-hand operand. In this case, destroying the left-hand operand requires that we remove pointers to this `Message` from the existing `folders`, which we do in the call to `remove_from_Folders`. Having removed itself from its `Folders`, we call `move` to move the `contents` from `rhs` to `this` object. What remains is to call `move_Messages` to update the `Folder` pointers.

Move Iterators

The `reallocate` member of `StrVec` (§ 13.5, p. 530) used a `for` loop to call `construct` to copy the elements from the old memory to the new. As an alternative to writing that loop, it would be easier if we could call `uninitialized_copy` to construct the newly allocated space. However, `uninitialized_copy` does what it says: It copies the elements. There is no analogous library function to "move" objects into unconstructed memory.

Instead, the new library defines a **move iterator** adaptor (§ 10.4, p. 401). A move iterator adapts its given iterator by changing the behavior of the iterator's dereference operator. Ordinarily, an iterator dereference operator returns an lvalue reference to the element. Unlike other iterators, the dereference operator of a move iterator yields an rvalue reference.

C++ 11

We transform an ordinary iterator to a move iterator by calling the library `make_move_iterator` function. This function takes an iterator and returns a move iterator.

All of the original iterator's other operations work as usual. Because these iterators support normal iterator operations, we can pass a pair of move iterators to an algorithm. In particular, we can pass move iterators to `uninitialized_copy`:

Click here to view code image

```
void StrVec::reallocate()
{
    // allocate space for twice as many elements as the current size
    auto newcapacity = size() ? 2 * size() : 1;
    auto first = alloc.allocate(newcapacity);
    // move the elements
    auto last = uninitialized_copy(make_move_iterator(begin()),
                                   make_move_iterator(end()),
                                   first);
    free();             // free the old space
    elements = first;   // update the pointers
    first_free = last;
    cap = elements + newcapacity;
}
```

`uninitialized_copy` calls `construct` on each element in the input sequence to "copy" that element into the destination. That algorithm uses the iterator dereference operator to fetch elements from the input sequence. Because we passed move iterators, the dereference operator yields an rvalue reference, which means `construct` will use the move constructor to construct the elements.

It is worth noting that standard library makes no guarantees about which algorithms can be used with move iterators and which cannot. Because moving an object can obliterate the source, you should pass move iterators to algorithms only when you are *confident* that the algorithm does not access an element after it has assigned to that element or passed that element to a user-defined function.

Advice: Don’t Be Too Quick to Move

Because a moved-from object has indeterminate state, calling `std::move` on an object is a dangerous operation. When we call `move`, we must be absolutely certain that there can be no other users of the moved-from object.

Judiciously used inside class code, `move` can offer significant performance benefits. Casually used in ordinary user code (as opposed to class implementation code), moving an object is more likely to lead to mysterious and hard-to-find bugs than to any improvement in the performance of the application.

Best Practices

Outside of class implementation code such as move constructors or move-assignment operators, use `std::move` only when you *are certain* that you need to do a move and that the move is guaranteed to be safe.

Exercises Section 13.6.2

Exercise 13.49: Add a move constructor and move-assignment operator to your `StrVec`, `String`, and `Message` classes.

Exercise 13.50: Put print statements in the move operations in your `String` class and rerun the program from exercise 13.48 in § 13.6.1 (p. 534) that used a `vector<String>` to see when the copies are avoided.

Exercise 13.51: Although `unique_ptrs` cannot be copied, in § 12.1.5 (p. 471) we wrote a `clone` function that returned a `unique_ptr` by value. Explain why that function is legal and how it works.

Exercise 13.52: Explain in detail what happens in the assignments of the `HasPtr` objects on page 541. In particular, describe step by step what happens to values of `hp`, `hp2`, and of the `rhs` parameter in the `HasPtr` assignment operator.

Exercise 13.53: As a matter of low-level efficiency, the `HasPtr` assignment operator is not ideal. Explain why. Implement a copy-assignment and move-assignment operator for `HasPtr` and compare the operations executed in your new move-assignment operator versus the copy-and-swap version.

Exercise 13.54: What would happen if we defined a `HasPtr` move-assignment operator but did not change the copy-and-swap operator? Write code to test your answer.

13.6.3. Rvalue References and Member Functions

Member functions other than constructors and assignment can benefit from providing both copy and move versions. Such move-enabled members typically use the same parameter pattern as the copy/move constructor and the assignment operators—one version takes an lvalue reference to `const`, and the second takes an rvalue reference to non`const`.

For example, the library containers that define `push_back` provide two versions: one that has an rvalue reference parameter and the other a `const` lvalue reference. Assuming `X` is the element type, these containers define:

Click here to view code image

```
void push_back(const X&); // copy: binds to any kind of X
void push_back(X&&);       // move: binds only to modifiable rvalues of type
X
```

We can pass any object that can be converted to type `X` to the first version of `push_back`. This version copies data from its parameter. We can pass only an rvalue that is not `const` to the second version. This version is an exact match (and a better match) for non`const` rvalues and will be run when we pass a modifiable rvalue (§ 13.6.2, p. 539). This version is free to steal resources from its parameter.

Ordinarily, there is no need to define versions of the operation that take a `const X&&` or a (plain) `X&`. Usually, we pass an rvalue reference when we want to "steal" from the argument. In order to do so, the argument must not be `const`. Similarly, copying from an object should not change the object being copied. As a result, there is usually no need to define a version that take a (plain) `X&` parameter.

Note

Overloaded functions that distinguish between moving and copying a parameter typically have one version that takes a `const T&` and one that takes a `T&&`.

As a more concrete example, we'll give our `StrVec` class a second version of `push_back`:

Click here to view code image

```
class StrVec {
public:
    void push_back(const std::string&);   // copy the element
    void push_back(std::string&&);        // move the element
```

```
        // other members as before
    };
    // unchanged from the original version in § 13.5 (p. 527)
    void StrVec::push_back(const string& s)
    {
        chk_n_alloc(); // ensure that there is room for another element
        // construct a copy of s in the element to which first_free points
        alloc.construct(first_free++, s);
    }
    void StrVec::push_back(string &&s)
    {
        chk_n_alloc(); // reallocates the StrVec if necessary
        alloc.construct(first_free++, std::move(s));
    }
```

These members are nearly identical. The difference is that the rvalue reference version of `push_back` calls `move` to pass its parameter to `construct`. As we've seen, the `construct` function uses the type of its second and subsequent arguments to determine which constructor to use. Because `move` returns an rvalue reference, the type of the argument to `construct` is `string&&`. Therefore, the `string` move constructor will be used to construct a new last element.

When we call `push_back` the type of the argument determines whether the new element is copied or moved into the container:

Click here to view code image

```
    StrVec vec;  // empty StrVec
    string s = "some string or another";
    vec.push_back(s);      // calls push_back(const string&)
    vec.push_back("done"); // calls push_back(string&&)
```

These calls differ as to whether the argument is an lvalue (`s`) or an rvalue (the temporary `string` created from `"done"`). The calls are resolved accordingly.

Rvalue and Lvalue Reference Member Functions

Ordinarily, we can call a member function on an object, regardless of whether that object is an lvalue or an rvalue. For example:

Click here to view code image

```
    string s1 = "a value", s2 = "another";
    auto n = (s1 + s2).find('a');
```

Here, we called the `find` member (§ 9.5.3, p. 364) on the `string` rvalue that results from adding two `strings`. Sometimes such usage can be surprising:

```
    s1 + s2 = "wow!";
```

Here we assign to the rvalue result of concatentating these `strings`.

Prior to the new standard, there was no way to prevent such usage. In order to maintain backward compatability, the library classes continue to allow assignment to rvalues, However, we might want to prevent such usage in our own classes. In this case, we'd like to force the left-hand operand (i.e., the object to which `this` points) to be an lvalue.

C++ 11

We indicate the lvalue/rvalue property of `this` in the same way that we define `const` member functions (§ 7.1.2, p. 258); we place a **reference qualifier** after the parameter list:

Click here to view code image

```
class Foo {
public:
    Foo &operator=(const Foo&) &; // may assign only to modifiable lvalues
    // other members of Foo
};
Foo &Foo::operator=(const Foo &rhs) &
{
    // do whatever is needed to assign rhs to this object
    return *this;
}
```

The reference qualifier can be either `&` or `&&`, indicating that `this` may point to an rvalue or lvalue, respectively. Like the `const` qualifier, a reference qualifier may appear only on a (`nonstatic`) member function and must appear in both the declaration and definition of the function.

We may run a function qualified by `&` only on an lvalue and may run a function qualified by `&&` only on an rvalue:

Click here to view code image

```
Foo &retFoo();   // returns a reference; a call to retFoo is an lvalue
Foo retVal();    // returns by value; a call to retVal is an rvalue
Foo i, j;        // i and j are lvalues
i = j;           // ok: i is an lvalue
retFoo() = j;    // ok: retFoo() returns an lvalue
retVal() = j;    // error: retVal() returns an rvalue
i = retVal();    // ok: we can pass an rvalue as the right-hand operand to assignment
```

A function can be both `const` and reference qualified. In such cases, the reference qualifier must follow the `const` qualifier:

Click here to view code image

```
class Foo {
```

```
public:
    Foo someMem() & const;     // error: const qualifier must come first
    Foo anotherMem() const &; // ok: const qualifier comes first
};
```

Overloading and Reference Functions

Just as we can overload a member function based on whether it is `const` (§ 7.3.2, p. 276), we can also overload a function based on its reference qualifier. Moreover, we may overload a function by its reference qualifier and by whether it is a `const` member. As an example, we'll give `Foo` a `vector` member and a function named `sorted` that returns a copy of the `Foo` object in which the `vector` is sorted:

Click here to view code image

```
class Foo {
public:
    Foo sorted() &&;           // may run on modifiable rvalues
    Foo sorted() const &;      // may run on any kind of Foo
    // other members of Foo
private:
    vector<int> data;
};
// this object is an rvalue, so we can sort in place
Foo Foo::sorted() &&
{
    sort(data.begin(), data.end());
    return *this;
}
// this object is either const or it is an lvalue; either way we can't sort in place
Foo Foo::sorted() const & {
    Foo ret(*this);                             // make a copy
    sort(ret.data.begin(), ret.data.end()); // sort the copy
    return ret;                                 // return the copy
}
```

When we run `sorted` on an rvalue, it is safe to sort the `data` member directly. The object is an rvalue, which means it has no other users, so we can change the object itself. When we run `sorted` on a `const` rvalue or on an lvalue, we can't change this object, so we copy `data` before sorting it.

Overload resolution uses the lvalue/rvalue property of the object that calls `sorted` to determine which version is used:

Click here to view code image

```
retVal().sorted(); // retVal() is an rvalue, calls Foo::sorted() &&
retFoo().sorted(); // retFoo() is an lvalue, calls Foo::sorted() const &
```

When we define `const` memeber functions, we can define two versions that differ only in that one is `const` qualified and the other is not. There is no similar default for reference qualified functions. When we define two or more members that have the same name and the same parameter list, we must provide a reference qualifier on all or none of those functions:

Click here to view code image

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const; // error: must have reference qualifier
    // Comp is type alias for the function type (see § 6.7 (p. 249))
    // that can be used to compare int values
    using Comp = bool(const int&, const int&);
    Foo sorted(Comp*);         // ok: different parameter list
    Foo sorted(Comp*) const;   // ok: neither version is reference qualified
};
```

Here the declaration of the `const` version of `sorted` that has no parameters is an error. There is a second version of `sorted` that has no parameters and that function has a reference qualifier, so the `const` version of that function must have a reference qualifier as well. On the other hand, the versions of `sorted` that take a pointer to a comparison operation are fine, because neither function has a qualifier.

Note

If a member function has a reference qualifier, all the versions of that member with the same parameter list must have reference qualifiers.

Exercises Section 13.6.3

Exercise 13.55: Add an rvalue reference version of `push_back` to your `StrBlob`.

Exercise 13.56: What would happen if we defined `sorted` as:

Click here to view code image

```
Foo Foo::sorted() const & {
    Foo ret(*this);
    return ret.sorted();
}
```

Exercise 13.57: What if we defined `sorted` as:

Click here to view code image

```
Foo Foo::sorted() const & { return Foo(*this).sorted(); }
```

Exercise 13.58: Write versions of class `Foo` with print statements in their `sorted` functions to test your answers to the previous two exercises.

Chapter Summary

Each class controls what happens when we copy, move, assign, or destroy objects of its type. Special member functions—the copy constructor, move constructor, copy-assignment operator, move-assignment operator, and destructor—define these operations. The move constructor and move-assignment operator take a (usually non`const`) rvalue reference; the copy versions take a (usually `const`) ordinary lvalue reference.

If a class declares none of these operations, the compiler will define them automatically. If not defined as deleted, these operations memberwise initialize, move, assign, or destroy the object: Taking each non`static` data member in turn, the synthesized operation does whatever is appropriate to the member's type to move, copy, assign, or destroy that member.

Classes that allocate memory or other resources almost always require that the class define the copy-control members to manage the allocated resource. If a class needs a destructor, then it almost surely needs to define the move and copy constructors and the move- and copy-assignment operators as well.

Defined Terms

copy and swap Technique for writing assignment operators by copying the right-hand operand followed by a call to `swap` to exchange the copy with the left-hand operand.

copy-assignment operator Version of the assignment operator that takes an object of the same type as its type. Ordinarily, the copy-assignment operator has a parameter that is a reference to `const` and returns a reference to its object. The compiler synthesizes the copy-assignment operator if the class does not explicitly provide one.

copy constructor Constructor that initializes a new object as a copy of another object of the same type. The copy constructor is applied implicitly to pass objects to or from a function by value. If we do not provide the copy constructor, the compiler synthesizes one for us.

copy control Special members that control what happens when objects of class type are copied, moved, assigned, and destroyed. The compiler synthesizes appropriate definitions for these operations if the class does not otherwise declare them.

copy initialization Form of initialization used when we use = to supply an initializer for a newly created object. Also used when we pass or return an object by value and when we initialize an array or an aggregate class. Copy initialization uses the copy constructor or the move constructor, depending on whether the initializer is an lvalue or an rvalue.

deleted function Function that may not be used. We delete a function by specifying `= delete` on its declaration. A common use of deleted functions is to tell the compiler not to synthesize the copy and/or move operations for a class.

destructor Special member function that cleans up an object when the object goes out of scope or is deleted. The compiler automatically destroys each data member. Members of class type are destroyed by invoking their destructor; no work is done when destroying members of built-in or compound type. In particular, the object pointed to by a pointer member is not deleted by the destructor.

lvalue reference Reference that can bind to an lvalue.

memberwise copy/assign How the synthesized copy and move constructors and the copy- and move-assignment operators work. Taking each non`static` data member in turn, the synthesized copy or move constructor initializes each member by copying or moving the corresponding member from the given object; the copy- or move-assignment operators copy-assign or move-assign each member from the right-hand object to the left. Members of built-in or compound type are initialized or assigned directly. Members of class type are initialized or assigned by using the member's corresponding copy/move constructor or copy-/move-assignment operator.

move Library function used to bind an rvalue reference to an lvalue. Calling `move` implicitly promises that we will not use the moved-from object except to destroy it or assign a new value to it.

move-assignment operator Version of the assignment operator that takes an rvalue reference to its type. Typically, a move-assignment operator moves data from the right-hand operand to the left. After the assignment, it must be safe to run the destructor on the right-hand operand.

move constructor Constructor that takes an rvalue reference to its type. Typically, a move constructor moves data from its parameter into the newly created object. After the move, it must be safe to run the destructor on the given argument.

move iterator Iterator adaptor that generates an iterator that, when dereferenced, yields an rvalue reference.

overloaded operator Function that redefines the meaning of an operator when applied to operand(s) of class type. This chapter showed how to define the assignment operator; Chapter 14 covers overloaded operators in more detail.

reference count Programming technique often used in copy-control members. A reference count keeps track of how many objects share state. Constructors (other than copy/move constructors) set the reference count to 1. Each time a new copy is made the count is incremented. When an object is destroyed, the count is decremented. The assignment operator and the destructor check whether the decremented reference count has gone to zero and, if so, they destroy the object.

reference qualifier Symbol used to indicate that a non`static` member function can be called on an lvalue or an rvalue. The qualifier, `&` or `&&`, follows the parameter list or the `const` qualifier if there is one. A function qualified by `&` may be called only on lvalues; a function qualified by `&&` may be called only on rvalues.

rvalue reference Reference to an object that is about to be destroyed.

synthesized assignment operator A version of the copy- or move-assignment operator created (synthesized) by the compiler for classes that do not explicitly define assignment operators. Unless it is defined as deleted, a synthesized assignment operator memberwise assigns (moves) the right-hand operand to the left.

synthesized copy/move constructor A version of the copy or move constructor that is generated by the compiler for classes that do not explicitly define the corresponding constructor. Unless it is defined as deleted, a synthesized copy or move constructor memberwise initializes the new object by copying or moving members from the given object, respectively.

synthesized destructor Version of the destructor created (synthesized) by the compiler for classes that do not explicitly define one. The synthesized destructor has an empty function body.

Chapter 14. Overloaded Operations and Conversions

Contents

In Chapter 4, we saw that C++ defines a large number of operators and automatic conversions among the built-in types. These facilities allow programmers to write a rich set of mixed-type expressions.

C++ lets us define what the operators mean when applied to objects of class type. It also lets us define conversions for class types. Class-type conversions are used like the built-in conversions to implicitly convert an object of one type to another type when needed.

Operator overloading lets us define the meaning of an operator when applied to operand(s) of a class type. Judicious use of operator overloading can make our programs easier to write and easier to read. As an example, because our original `Sales_item` class type (§ 1.5.1, p. 20) defined the input, output, and addition operators, we can print the sum of two `Sales_items` as

Click here to view code image

```
cout << item1 + item2;   // print the sum of two Sales_items
```

In contrast, because our `Sales_data` class (§ 7.1, p. 254) does not yet have overloaded operators, code to print their sum is more verbose and, hence, less clear:

Click here to view code image

```
print(cout, add(data1, data2));   // print the sum of two Sales_datas
```

14.1. Basic Concepts

Overloaded operators are functions with special names: the keyword `operator` followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type, a parameter list, and a body.

An overloaded operator function has the same number of parameters as the operator has operands. A unary operator has one parameter; a binary operator has two. In a binary operator, the left-hand operand is passed to the first parameter and the right-hand operand to the second. Except for the overloaded function-call operator, `operator()`, an overloaded operator may not have default arguments (§ 6.5.1, p. 236).

If an operator function is a member function, the first (left-hand) operand is bound to the implicit `this` pointer (§ 7.1.2, p. 257). Because the first operand is implicitly bound to `this`, a member operator function has one less (explicit) parameter than the operator has operands.

Note

When an overloaded operator is a member function, `this` is bound to the left-hand operand. Member operator functions have one less (explicit) parameter than the number of operands.

An operator function must either be a member of a class or have at least one parameter of class type:

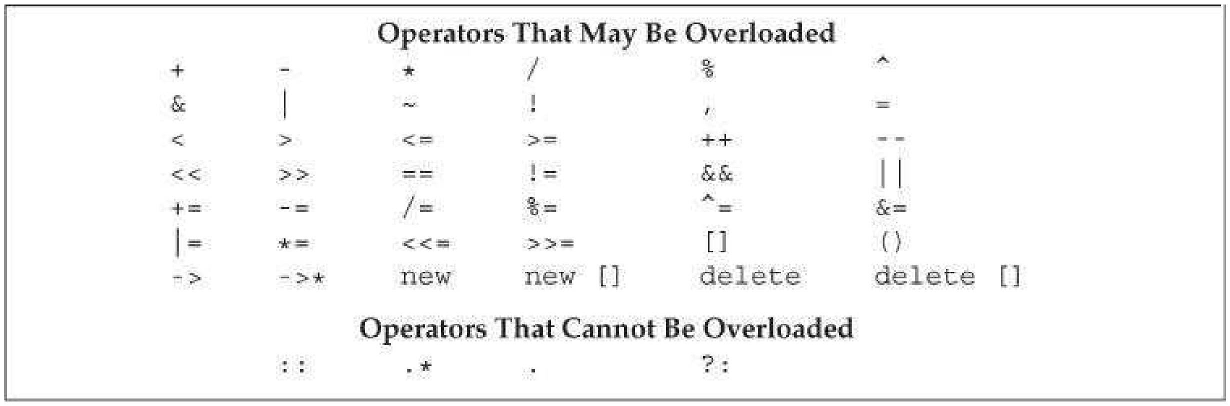
Click here to view code image

```
// error: cannot redefine the built-in operator for ints
int operator+(int, int);
```

This restriction means that we cannot change the meaning of an operator when applied to operands of built-in type.

We can overload most, but not all, of the operators. Table 14.1 shows whether or not an operator may be overloaded. We'll cover overloading `new` and `delete` in § 19.1.1 (p. 820).

Table 14.1. Operators



Operators That May Be Overloaded					
+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []
Operators That Cannot Be Overloaded					
	::	.*	.	?:	

We can overload only existing operators and cannot invent new operator symbols. For example, we cannot define `operator**` to provide exponentiation.

Four symbols (`+`, `-`, `*`, and `&`) serve as both unary and binary operators. Either or both of these operators can be overloaded. The number of parameters determines which operator is being defined.

An overloaded operator has the same precedence and associativity (§ 4.1.2, p. 136) as the corresponding built-in operator. Regardless of the operand types

```
x == y + z;
```

is always equivalent to `x == (y + z)`.

Calling an Overloaded Operator Function Directly

Ordinarily, we "call" an overloaded operator function indirectly by using the operator on arguments of the appropriate type. However, we can also call an overloaded operator function directly in the same way that we call an ordinary function. We name the function and pass an appropriate number of arguments of the appropriate type:

Click here to view code image

```
// equivalent calls to a nonmember operator function
data1 + data2;             // normal expression
operator+(data1, data2);   // equivalent function call
```

These calls are equivalent: Both call the nonmember function `operator+`, passing `data1` as the first argument and `data2` as the second.

We call a member operator function explicitly in the same way that we call any other member function. We name an object (or pointer) on which to run the function and use the dot (or arrow) operator to fetch the function we wish to call:

Click here to view code image

```
data1 += data2;             // expression-based "call"
data1.operator+=(data2);    // equivalent call to a member operator
function
```

Each of these statements calls the member function `operator+=`, binding `this` to the address of `data1` and passing `data2` as an argument.

Some Operators Shouldn't Be Overloaded

Recall that a few operators guarantee the order in which operands are evaluated. Because using an overloaded operator is really a function call, these guarantees do not apply to overloaded operators. In particular, the operand-evaluation guarantees of the logical AND, logical OR (§ 4.3, p. 141), and comma (§ 4.10, p. 157) operators are not preserved. Moreover, overloaded versions of && or `||` operators do not preserve short-circuit evaluation properties of the built-in operators. Both operands are always evaluated.

Because the overloaded versions of these operators do not preserve order of evaluation and/or short-circuit evaluation, it is usually a bad idea to overload them. Users are likely to be surprised when the evaluation guarantees they are accustomed to are not honored for code that happens to use an overloaded version of one of these operators.

Another reason not to overload comma, which also applies to the address-of operator, is that unlike most operators, the language defines what the comma and address-of operators mean when applied to objects of class type. Because these operators have built-in meaning, they ordinarily should not be overloaded. Users of the class will be surprised if these operators behave differently from their normal meanings.

Best Practices

Ordinarily, the comma, address-of, logical AND, and logical OR operators should *not* be overloaded.

Use Definitions That Are Consistent with the Built-in Meaning

When you design a class, you should always think first about what operations the class will provide. Only after you know what operations are needed should you think about whether to define each operation as an ordinary function or as an overloaded operator. Those operations with a logical mapping to an operator are good candidates for defining as overloaded operators:

- If the class does IO, define the shift operators to be consistent with how IO is done for the built-in types.
- If the class has an operation to test for equality, define `operator==`. If the class has `operator==`, it should usually have `operator!=` as well.
- If the class has a single, natural ordering operation, define `operator<`. If the class has `operator<`, it should probably have all of the relational operators.
- The return type of an overloaded operator usually should be compatible with the return from the built-in version of the operator: The logical and relational operators should return `bool`, the arithmetic operators should return a value of the class type, and assignment and compound assignment should return a reference to the left-hand operand.

Assignment and Compound Assignment Operators

Assignment operators should behave analogously to the synthesized operators: After an assignment, the values in the left-hand and right-hand operands should have the same value, and the operator should return a reference to its left-hand operand. Overloaded assignment should generalize the built-in meaning of assignment, not circumvent it.

Caution: Use Operator Overloading Judiciously

Each operator has an associated meaning from its use on the built-in types. Binary +, for example, is strongly identified with addition. Mapping binary + to an analogous operation for a class type can provide a convenient notational shorthand. For example, the library `string` type, following a convention common to many programming languages, uses + to represent concatenation—“adding” one `string` to the other.

Operator overloading is most useful when there is a logical mapping of a built-in operator to an operation on our type. Using overloaded operators rather than inventing named operations can make our programs more natural and intuitive. Overuse or outright abuse of operator overloading can make our classes incomprehensible.

Obvious abuses of operator overloading rarely happen in practice. As an example, no responsible programmer would define `operator+` to perform subtraction. More common, but still inadvisable, are uses that contort an operator’s “normal” meaning to force a fit to a given type. Operators should be used only for operations that are likely to be unambiguous to users. An operator has an ambiguous meaning if it plausibly has more than one interpretation.

If a class has an arithmetic (§ 4.2, p. 139) or bitwise (§ 4.8, p. 152) operator, then it is usually a good idea to provide the corresponding compound-assignment operator as well. Needless to say, the += operator should be defined to behave the same way the built-in operators do: it should behave as + followed by =.

Choosing Member or Nonmember Implementation

When we define an overloaded operator, we must decide whether to make the operator a class member or an ordinary nonmember function. In some cases, there is no choice—some operators are required to be members; in other cases, we may not be able to define the operator appropriately if it is a member.

The following guidelines can be of help in deciding whether to make an operator a member or an ordinary nonmember function:

- The assignment (`=`), subscript (`[]`), call (`()`), and member access arrow (`->`) operators *must* be defined as members.
- The compound-assignment operators ordinarily *ought* to be members. However, unlike assignment, they are not required to be members.
- Operators that change the state of their object or that are closely tied to their

given type—such as increment, decrement, and dereference—usually should be members.

- Symmetric operators—those that might convert either operand, such as the arithmetic, equality, relational, and bitwise operators—usually should be defined as ordinary nonmember functions.

Programmers expect to be able to use symmetric operators in expressions with mixed types. For example, we can add an `int` and a `double`. The addition is symmetric because we can use either type as the left-hand or the right-hand operand. If we want to provide similar mixed-type expressions involving class objects, then the operator must be defined as a nonmember function.

When we define an operator as a member function, then the left-hand operand must be an object of the class of which that operator is a member. For example:

Click here to view code image

```
string s = "world";
string t = s + "!";   // ok: we can add a const char* to a string
string u = "hi" + s;  // would be an error if + were a member of string
```

If `operator+` were a member of the `string` class, the first addition would be equivalent to `s.operator+("!")`. Likewise, `"hi" + s` would be equivalent to `"hi".operator+(s)`. However, the type of `"hi"` is `const char*`, and that is a built-in type; it does not even have member functions.

Because `string` defines + as an ordinary nonmember function, `"hi" + s` is equivalent to `operator+("hi", s)`. As with any function call, either of the arguments can be converted to the type of the parameter. The only requirements are that at least one of the operands has a class type, and that both operands can be converted (unambiguously) to `string`.

Exercises Section 14.1

Exercise 14.1: In what ways does an overloaded operator differ from a built-in operator? In what ways are overloaded operators the same as the built-in operators?

Exercise 14.2: Write declarations for the overloaded input, output, addition, and compound-assignment operators for `Sales_data`.

Exercise 14.3: Both `string` and `vector` define an overloaded == that can be used to compare objects of those types. Assuming `svec1` and `svec2` are `vector`s that hold `string`s, identify which version of == is applied in each of the following expressions:

(a) `"cobble" == "stone"`

(b) `svec1[0] == svec2[0]`

(c) `svec1 == svec2`

(d) `"svec1[0] == "stone"`

Exercise 14.4: Explain how to decide whether the following should be class members:

(a) `%`

(b) `%=`

(c) `++`

(d) `->`

(e) `<<`

(f) `&&`

(g) `==`

(h) `()`

Exercise 14.5: In exercise 7.40 from § 7.5.1 (p. 291) you wrote a sketch of one of the following classes. Decide what, if any, overloaded operators your class should provide.

(a) `Book`

(b) `Date`

(c) `Employee`

(d) `Vehicle`

(e) `Object`

(f) `Tree`

14.2. Input and Output Operators

As we've seen, the IO library uses >> and << for input and output, respectively. The IO library itself defines versions of these operators to read and write the built-in types. Classes that support IO ordinarily define versions of these operators for objects of the class type.

14.2.1. Overloading the Output Operator <<

Ordinarily, the first parameter of an output operator is a reference to a nonconst `ostream` object. The `ostream` is nonconst because writing to the stream changes its state. The parameter is a reference because we cannot copy an `ostream` object.

The second parameter ordinarily should be a reference to `const` of the class type we want to print. The parameter is a reference to avoid copying the argument. It can be `const` because (ordinarily) printing an object does not change that object.

To be consistent with other output operators, `operator<<` normally returns its `ostream` parameter.

The Sales_data Output Operator

As an example, we'll write the `Sales_data` output operator:

Click here to view code image

```
ostream &operator<<(ostream &os, const Sales_data &item)
{
    os << item.isbn() << " " << item.units_sold << " "
       << item.revenue << " " << item.avg_price();
    return os;
}
```

Except for its name, this function is identical to our earlier `print` function (§ 7.1.3, p. 261). Printing a `Sales_data` entails printing its three data elements and the computed average sales price. Each element is separated by a space. After printing the values, the operator returns a reference to the `ostream` it just wrote.

Output Operators Usually Do Minimal Formatting

The output operators for the built-in types do little if any formatting. In particular, they do not print newlines. Users expect class output operators to behave similarly. If the operator does print a newline, then users would be unable to print descriptive text along with the object on the same line. An output operator that does minimal formatting lets users control the details of their output.

★ Best Practices

Generally, output operators should print the contents of the object, with minimal formatting. They should not print a newline.

IO Operators Must Be Nonmember Functions

Input and output operators that conform to the conventions of the `iostream` library must be ordinary nonmember functions. These operators cannot be members of our own class. If they were, then the left-hand operand would have to be an object of our class type:

Click here to view code image

```
Sales_data data;
data << cout; // if operator<< is a member of Sales_data
```

If these operators are members of any class, they would have to be members of `istream` or `ostream`. However, those classes are part of the standard library, and we cannot add members to a class in the library.

Thus, if we want to define the IO operators for our types, we must define them as nonmember functions. Of course, IO operators usually need to read or write the non`public` data members. As a consequence, IO operators usually must be declared as friends (§ 7.2.1, p. 269).

Exercises Section 14.2.1

Exercise 14.6: Define an output operator for your `Sales_data` class.

Exercise 14.7: Define an output operator for you `String` class you wrote for the exercises in § 13.5 (p. 531).

Exercise 14.8: Define an output operator for the class you chose in exercise 7.40 from § 7.5.1 (p. 291).

14.2.2. Overloading the Input Operator >>

Ordinarily the first parameter of an input operator is a reference to the stream from which it is to read, and the second parameter is a reference to the (non`const`) object into which to read. The operator usually returns a reference to its given stream. The second parameter must be non`const` because the purpose of an input operator is to read data into this object.

The Sales_data Input Operator

As an example, we'll write the `Sales_data` input operator:

Click here to view code image

```
istream &operator>>(istream &is, Sales_data &item)
{
    double price;   // no need to initialize; we'll read into price before we use
it
    is >> item.bookNo >> item.units_sold >> price;
    if (is)         // check that the inputs succeeded
        item.revenue = item.units_sold * price;
    else
```

```
            item = Sales_data(); // input failed: give the object the default
state
        return is;
    }
```

Except for the `if` statement, this definition is similar to our earlier `read` function (§ 7.1.3, p. 261). The `if` checks whether the reads were successful. If an IO error occurs, the operator resets its given object to the empty `Sales_data`. That way, the object is guaranteed to be in a consistent state.

Note

Input operators must deal with the possibility that the input might fail; output operators generally don't bother.

Errors during Input

The kinds of errors that might happen in an input operator include the following:

- A read operation might fail because the stream contains data of an incorrect type. For example, after reading `bookNo`, the input operator assumes that the next two items will be numeric data. If nonnumeric data is input, that read and any subsequent use of the stream will fail.
- Any of the reads could hit end-of-file or some other error on the input stream.

Rather than checking each read, we check once after reading all the data and before using those data:

Click here to view code image

```
if (is)          // check that the inputs succeeded
    item.revenue = item.units_sold * price;
else
    item = Sales_data(); // input failed: give the object the default state
```

If any of the read operations fails, `price` will have an undefined value. Therefore, before using `price`, we check that the input stream is still valid. If it is, we do the calculation and store the result in `revenue`. If there was an error, we do not worry about which input failed. Instead, we reset the entire object to the empty `Sales_data` by assigning a new, default-initialized `Sales_data` object to `item`. After this assignment, `item` will have an empty `string` for its `bookNo` member, and its `revenue` and `units_sold` members will be zero.

Putting the object into a valid state is especially important if the object might have been partially changed before the error occurred. For example, in this input operator, we might encounter an error after successfully reading a new `bookNo`. An error after reading `bookNo` would mean that the `units_sold` and `revenue` members of the old object were unchanged. The effect would be to associate a different bookNo with those data.

By leaving the object in a valid state, we (somewhat) protect a user that ignores the possibility of an input error. The object will be in a usable state—its members are all defined. Similarly, the object won't generate misleading results—its data are internally consistent.

Best Practices

Input operators should decide what, if anything, to do about error recovery.

Indicating Errors

Some input operators need to do additional data verification. For example, our input operator might check that the bookNo we read is in an appropriate format. In such cases, the input operator might need to set the stream's condition state to indicate failure (§ 8.1.2, p. 312), even though technically speaking the actual IO was successful. Usually an input operator should set only the failbit. Setting eofbit would imply that the file was exhausted, and setting badbit would indicate that the stream was corrupted. These errors are best left to the IO library itself to indicate.

Exercises Section 14.2.2

Exercise 14.9: Define an input operator for your Sales_data class.

Exercise 14.10: Describe the behavior of the Sales_data input operator if given the following input:

(a) 0-201-99999-9 10 24.95

(b) 10 24.95 0-210-99999-9

Exercise 14.11: What, if anything, is wrong with the following Sales_data input operator? What would happen if we gave this operator the data in the previous exercise?

Click here to view code image

```
istream& operator>>(istream& in, Sales_data& s)
{
    double price;
    in >> s.bookNo >> s.units_sold >> price;
    s.revenue = s.units_sold * price;
    return in;
}
```

Exercise 14.12: Define an input operator for the class you used in exercise 7.40 from § 7.5.1 (p. 291). Be sure the operator handles input errors.

14.3. Arithmetic and Relational Operators

Ordinarily, we define the arithmetic and relational operators as nonmember functions in order to allow conversions for either the left- or right-hand operand (§ 14.1, p. 555). These operators shouldn't need to change the state of either operand, so the parameters are ordinarily references to `const`.

An arithmetic operator usually generates a new value that is the result of a computation on its two operands. That value is distinct from either operand and is calculated in a local variable. The operation returns a copy of this local as its result. Classes that define an arithmetic operator generally define the corresponding compound assignment operator as well. When a class has both operators, it is usually more efficient to define the arithmetic operator to use compound assignment:

Click here to view code image

```
// assumes that both objects refer to the same book
Sales_data
operator+(const Sales_data &lhs, const Sales_data &rhs)
{
    Sales_data sum = lhs;  // copy data members from lhs into sum
    sum += rhs;             // add rhs into sum
    return sum;
}
```

This definition is essentially identical to our original `add` function (§ 7.1.3, p. 261). We copy `lhs` into the local variable `sum`. We then use the `Sales_data` compound-assignment operator (which we'll define on page 564) to add the values from `rhs` into `sum`. We end the function by returning a copy of `sum`.

Tip

Classes that define both an arithmetic operator and the related compound assignment ordinarily ought to implement the arithmetic operator by using the compound assignment.

Exercises Section 14.3

Exercise 14.13: Which other arithmetic operators (Table 4.1 (p. 139)), if any, do you think `Sales_data` ought to support? Define any you think the class should include.

Exercise 14.14: Why do you think it is more efficient to define `operator+`

to call `operator+=` rather than the other way around?

Exercise 14.15: Should the class you chose for exercise 7.40 from § 7.5.1 (p. 291) define any of the arithmetic operators? If so, implement them. If not, explain why not.

14.3.1. Equality Operators

Ordinarily, classes in C++ define the equality operator to test whether two objects are equivalent. That is, they usually compare every data member and treat two objects as equal if and only if all the corresponding members are equal. In line with this design philosophy, our `Sales_data` equality operator should compare the `bookNo` as well as the sales figures:

Click here to view code image

```
bool operator==(const Sales_data &lhs, const Sales_data &rhs)
{
    return lhs.isbn() == rhs.isbn() &&
           lhs.units_sold == rhs.units_sold &&
           lhs.revenue == rhs.revenue;
}
bool operator!=(const Sales_data &lhs, const Sales_data &rhs)
{
    return !(lhs == rhs);
}
```

The definition of these functions is trivial. More important are the design principles that these functions embody:

- If a class has an operation to determine whether two objects are equal, it should define that function as `operator==` rather than as a named function: Users will expect to be able to compare objects using `==`; providing `==` means they won't need to learn and remember a new name for the operation; and it is easier to use the library containers and algorithms with classes that define the `==` operator.
- If a class defines `operator==`, that operator ordinarily should determine whether the given objects contain equivalent data.
- Ordinarily, the equality operator should be transitive, meaning that if `a == b` and `b == c` are both true, then `a == c` should also be true.
- If a class defines `operator==`, it should also define `operator!=`. Users will expect that if they can use `==` then they can also use `!=`, and vice versa.
- One of the equality or inequality operators should delegate the work to the other. That is, one of these operators should do the real work to compare objects. The other should call the one that does the real work.

Best Practices

Classes for which there is a logical meaning for equality normally should define `operator==`. Classes that define `==` make it easier for users to use the class with the library algorithms.

Exercises Section 14.3.1

Exercise 14.16: Define equality and inequality operators for your `StrBlob` (§ 12.1.1, p. 456), `StrBlobPtr` (§ 12.1.6, p. 474), `StrVec` (§ 13.5, p. 526), and `String` (§ 13.5, p. 531) classes.

Exercise 14.17: Should the class you chose for exercise 7.40 from § 7.5.1 (p. 291) define the equality operators? If so, implement them. If not, explain why not.

14.3.2. Relational Operators

Classes for which the equality operator is defined also often (but not always) have relational operators. In particular, because the associative containers and some of the algorithms use the less-than operator, it can be useful to define an `operator<`.

Ordinarily the relational operators should

1. Define an ordering relation that is consistent with the requirements for use as a key to an associative container (§ 11.2.2, p. 424); and
2. Define a relation that is consistent with `==` if the class has both operators. In particular, if two objects are `!=`, then one object should be `<` the other.

Although we might think our `Sales_data` class should support the relational operators, it turns out that it probably should not do so. The reasons are subtle and are worth understanding.

We might think that we'd define `<` similarly to `compareIsbn` (§ 11.2.2, p. 425). That function compared `Sales_data` objects by comparing their ISBNS. Although `compareIsbn` provides an ordering relation that meets requirment 1, that function yields results that are inconsistent with our definition of `==`. As a result, it does not meet requirement 2.

The `Sales_data ==` operator treats two transactions with the same ISBN as unequal if they have different `revenue` or `units_sold` members. If we defined the

< operator to compare only the ISBN member, then two objects with the same ISBN but different `units_sold` or `revenue` would compare as unequal, but neither object would be less than the other. Ordinarily, if we have two objects, neither of which is less than the other, then we expect that those objects are equal.

We might think that we should, therefore, define `operator<` to compare each data element in turn. We could define `operator<` to compare objects with equal `isbns` by looking next at the `units_sold` and then at the `revenue` members.

However, there is nothing essential about this ordering. Depending on how we plan to use the class, we might want to define the order based first on either `revenue` or `units_sold`. We might want those objects with fewer `units_sold` to be "less than" those with more. Or we might want to consider those with smaller `revenue` "less than" those with more.

For `Sales_data`, there is no single logical definition of <. Thus, it is better for this class not to define < at all.

Best Practices

If a single logical definition for < exists, classes usually should define the < operator. However, if the class also has ==, define < only if the definitions of < and == yield consistent results.

Exercises Section 14.3.2

Exercise 14.18: Define relational operators for your `StrBlob`, `StrBlobPtr`, `StrVec`, and `String` classes.

Exercise 14.19: Should the class you chose for exercise 7.40 from § 7.5.1 (p. 291) define the relational operators? If so, implement them. If not, explain why not.

14.4. Assignment Operators

In addition to the copy- and move-assignment operators that assign one object of the class type to another object of the same type (§ 13.1.2, p. 500, and § 13.6.2, p. 536), a class can define additional assignment operators that allow other types as the right-hand operand.

As one example, in addition to the copy- and move-assignment operators, the library `vector` class defines a third assignment operator that takes a braced list of elements (§ 9.2.5, p. 337). We can use this operator as follows:

Click here to view code image

```
vector<string> v;
v = {"a", "an", "the"};
```

We can add this operator to our `StrVec` class (§ 13.5, p. 526) as well:

Click here to view code image

```
class StrVec {
public:
    StrVec &operator=(std::initializer_list<std::string>);
    // other members as in § 13.5 (p. 526)
};
```

To be consistent with assignment for the built-in types (and with the copy- and move-assignment operators we already defined), our new assignment operator will return a reference to its left-hand operand:

Click here to view code image

```
StrVec &StrVec::operator=(initializer_list<string> il)
{
    // alloc_n_copy allocates space and copies elements from the given range
    auto data = alloc_n_copy(il.begin(), il.end());
    free();    // destroy the elements in this object and free the space
     elements = data.first; // update data members to point to the new
space
    first_free = cap = data.second;
    return *this;
}
```

As with the copy- and move-assignment operators, other overloaded assignment operators have to free the existing elements and create new ones. Unlike the copy- and move-assignment operators, this operator does not need to check for self-assignment. The parameter is an `initializer_list<string>` (§ 6.2.6, p. 220), which means that `il` cannot be the same object as the one denoted by `this`.

Note

Assignment operators can be overloaded. Assignment operators, regardless of parameter type, must be defined as member functions.

Compound-Assignment Operators

Compound assignment operators are not required to be members. However, we prefer to define all assignments, including compound assignments, in the class. For consistency with the built-in compound assignment, these operators should return a reference to their left-hand operand. For example, here is the definition of the

`Sales_data` compound-assignment operator:

Click here to view code image

```
// member binary operator: left-hand operand is bound to the implicit this pointer
// assumes that both objects refer to the same book
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

★ **Best Practices**

Assignment operators must, and ordinarily compound-assignment operators should, be defined as members. These operators should return a reference to the left-hand operand.

14.5. Subscript Operator

Classes that represent containers from which elements can be retrieved by position often define the subscript operator, `operator[]`.

Exercises Section 14.4

Exercise 14.20: Define the addition and compound-assignment operators for your `Sales_data` class.

Exercise 14.21: Write the `Sales_data` operators so that + does the actual addition and += calls +. Discuss the disadvantages of this approach compared to the way these operators were defined in § 14.3 (p. 560) and § 14.4 (p. 564).

Exercise 14.22: Define a version of the assignment operator that can assign a `string` representing an ISBN to a `Sales_data`.

Exercise 14.23: Define an `initializer_list` assignment operator for your version of the `StrVec` class.

Exercise 14.24: Decide whether the class you used in exercise 7.40 from § 7.5.1 (p. 291) needs a copy- and move-assignment operator. If so, define those operators.

Exercise 14.25: Implement any other assignment operators your class should define. Explain which types should be used as operands and why.

Note

The subscript operator must be a member function.

To be compatible with the ordinary meaning of subscript, the subscript operator usually returns a reference to the element that is fetched. By returning a reference, subscript can be used on either side of an assignment. Consequently, it is also usually a good idea to define both `const` and nonconst versions of this operator. When applied to a `const` object, subscript should return a reference to `const` so that it is not possible to assign to the returned object.

Best Practices

If a class has a subscript operator, it usually should define two versions: one that returns a plain reference and the other that is a `const` member and returns a reference to `const`.

As an example, we'll define subscript for `StrVec` (§ 13.5, p. 526):

Click here to view code image

```
class StrVec {
public:
    std::string& operator[](std::size_t n)
        { return elements[n]; }
    const std::string& operator[](std::size_t n) const
        { return elements[n]; }
    // other members as in § 13.5 (p. 526)
private:
    std::string *elements;    // pointer to the first element in the array
};
```

We can use these operators similarly to how we subscript a `vector` or array. Because subscript returns a reference to an element, if the `StrVec` is nonconst, we can assign to that element; if we subscript a `const` object, we can't:

Click here to view code image

```
// assume svec is a StrVec
const StrVec cvec = svec; // copy elements from svec into cvec
// if svec has any elements, run the string empty function on the first one
if (svec.size() && svec[0].empty()) {
    svec[0] = "zero"; // ok: subscript returns a reference to a string
    cvec[0] = "Zip";  // error: subscripting cvec returns a reference to const
}
```

Exercises Section 14.5

Exercise 14.26: Define subscript operators for your `StrVec`, `String`, `StrBlob`, and `StrBlobPtr` classes.

14.6. Increment and Decrement Operators

The increment (`++`) and decrement (`--`) operators are most often implemented for iterator classes. These operators let the class move between the elements of a sequence. There is no language requirement that these operators be members of the class. However, because these operators change the state of the object on which they operate, our preference is to make them members.

For the built-in types, there are both prefix and postfix versions of the increment and decrement operators. Not surprisingly, we can define both the prefix and postfix instances of these operators for our own classes as well. We'll look at the prefix versions first and then implement the postfix ones.

Best Practices

Classes that define increment or decrement operators should define both the prefix and postfix versions. These operators usually should be defined as members.

Defining Prefix Increment/Decrement Operators

To illustrate the increment and decrement operators, we'll define these operators for our `StrBlobPtr` class (§ 12.1.6, p. 474):

Click here to view code image

```
class StrBlobPtr {
public:
    // increment and decrement
    StrBlobPtr& operator++();       // prefix operators
    StrBlobPtr& operator--();
    // other members as before
};
```

Best Practices

To be consistent with the built-in operators, the prefix operators should return a reference to the incremented or decremented object.

The increment and decrement operators work similarly to each other—they call `check` to verify that the `StrBlobPtr` is still valid. If so, `check` also verifies that its given index is valid. If `check` doesn't throw an exception, these operators return a reference to this object.

In the case of increment, we pass the current value of `curr` to `check`. So long as that value is less than the size of the underlying `vector`, `check` will return. If `curr` is already at the end of the `vector`, `check` will throw:

Click here to view code image

```
// prefix: return a reference to the incremented/decremented object
StrBlobPtr& StrBlobPtr::operator++()
{
    // if curr already points past the end of the container, can't increment it
    check(curr, "increment past end of StrBlobPtr");
    ++curr;       // advance the current state
    return *this;
}

StrBlobPtr& StrBlobPtr::operator--()
{
    // if curr is zero, decrementing it will yield an invalid subscript
    --curr;       // move the current state back one element
    check(-1, "decrement past begin of StrBlobPtr");
    return *this;
}
```

The decrement operator decrements `curr` before calling `check`. That way, if `curr` (which is an `unsigned` number) is already zero, the value that we pass to `check` will be a large positive value representing an invalid subscript (§ 2.1.2, p. 36).

Differentiating Prefix and Postfix Operators

There is one problem with defining both the prefix and postfix operators: Normal overloading cannot distinguish between these operators. The prefix and postfix versions use the same symbol, meaning that the overloaded versions of these operators have the same name. They also have the same number and type of operands.

To solve this problem, the postfix versions take an extra (unused) parameter of type `int`. When we use a postfix operator, the compiler supplies 0 as the argument for this parameter. Although the postfix function can use this extra parameter, it usually should not. That parameter is not needed for the work normally performed by a postfix operator. Its sole purpose is to distinguish a postfix function from the prefix version.

We can now add the postfix operators to `StrBlobPtr`:

Click here to view code image

```
class StrBlobPtr {
public:
    // increment and decrement
    StrBlobPtr operator++(int);    // postfix operators
    StrBlobPtr operator--(int);
    // other members as before
};
```

Best Practices

To be consistent with the built-in operators, the postfix operators should return the old (unincremented or undecremented) value. That value is returned as a value, not a reference.

The postfix versions have to remember the current state of the object before incrementing the object:

Click here to view code image

```
// postfix: increment/decrement the object but return the unchanged value
StrBlobPtr StrBlobPtr::operator++(int)
{
    // no check needed here; the call to prefix increment will do the check
    StrBlobPtr ret = *this;    // save the current value
    ++*this;      // advance one element; prefix ++ checks the increment
    return ret;   // return the saved state
}
StrBlobPtr StrBlobPtr::operator--(int)
{
    // no check needed here; the call to prefix decrement will do the check
    StrBlobPtr ret = *this;   // save the current value
     --*this;      // move backward one element; prefix -- checks the
decrement
    return ret;   // return the saved state
}
```

Each of our operators calls its own prefix version to do the actual work. For example, the postfix increment operator executes

```
++*this
```

This expression calls the prefix increment operator. That operator checks that the increment is safe and either throws an exception or increments `curr`. Assuming `check` doesn't throw an exception, the postfix functions return the stored copy in `ret`. Thus, after the return, the object itself has been advanced, but the value returned reflects the original, unincremented value.

Note

The `int` parameter is not used, so we do not give it a name.

Calling the Postfix Operators Explicitly

As we saw on page 553, we can explicitly call an overloaded operator as an alternative to using it as an operator in an expression. If we want to call the postfix version using a function call, then we must pass a value for the integer argument:

Click here to view code image

```
StrBlobPtr p(a1); // p points to the vector inside a1
p.operator++(0);  // call postfix operator++
p.operator++();   // call prefix  operator++
```

The value passed usually is ignored but is necessary in order to tell the compiler to use the postfix version.

Exercises Section 14.6

Exercise 14.27: Add increment and decrement operators to your `StrBlobPtr` class.

Exercise 14.28: Define addition and subtraction for `StrBlobPtr` so that these operators implement pointer arithmetic (§ 3.5.3, p. 119).

Exercise 14.29: We did not define a `const` version of the increment and decrement operators. Why not?

14.7. Member Access Operators

The dereference (`*`) and arrow (`->`) operators are often used in classes that represent iterators and in smart pointer classes (§ 12.1, p. 450). We can logically add these operators to our `StrBlobPtr` class as well:

Click here to view code image

```
class StrBlobPtr {
public:
    std::string& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr];  // (*p) is the vector to which this object points
    }
    std::string* operator->() const
    { // delegate the real work to the dereference operator
     return & this->operator*();
    }
    // other members as before
};
```

The dereference operator checks that `curr` is still in range and, if so, returns a reference to the element denoted by `curr`. The arrow operator avoids doing any work of its own by calling the dereference operator and returning the address of the element returned by that operator.

Note

Operator arrow must be a member. The dereference operator is not required to be a member but usually should be a member as well.

It is worth noting that we've defined these operators as `const` members. Unlike the increment and decrment operators, fetching an element doesn't change the state of a `StrBlobPtr`. Also note that these operators return a reference or pointer to nonconst `string`. They do so because we know that a `StrBlobPtr` can only be bound to a nonconst `StrBlob` (§ 12.1.6, p. 474).

We can use these operators the same way that we've used the corresponding operations on pointers or `vector` iterators:

Click here to view code image

```
StrBlob a1 = {"hi", "bye", "now"};
StrBlobPtr p(a1);            // p points to the vector inside a1
*p = "okay";                 // assigns to the first element in a1
cout << p->size() << endl;   // prints 4, the size of the first element in a1
cout << (*p).size() << endl; // equivalent to p->size()
```

Constraints on the Return from Operator Arrow

As with most of the other operators (although it would be a bad idea to do so), we can define `operator*` to do whatever processing we like. That is, we can define `operator*` to return a fixed value, say, 42, or print the contents of the object to which it is applied, or whatever. The same is not true for overloaded arrow. The arrow operator never loses its fundamental meaning of member access. When we overload arrow, we change the object from which arrow fetches the specified member. We cannot change the fact that arrow fetches a member.

When we write `point->mem`, `point` must be a pointer to a class object or it must be an object of a class with an overloaded `operator->`. Depending on the type of `point`, writing `point->mem` is equivalent to

Click here to view code image

```
(*point).mem;           // point is a built-in pointer type
point.operator()->mem; // point is an object of class type
```

Otherwise the code is in error. That is, `point->mem` executes as follows:

1. If `point` is a pointer, then the built-in arrow operator is applied, which means this expression is a synonym for `(*point).mem`. The pointer is dereferenced and the indicated member is fetched from the resulting object. If the type pointed to by `point` does not have a member named `mem`, then the code is in error.
2. If `point` is an object of a class that defines `operator->`, then the result of `point.operator->()` is used to fetch `mem`. If that result is a pointer, then step 1 is executed on that pointer. If the result is an object that itself has an overloaded `operator->()`, then this step is repeated on that object. This process continues until either a pointer to an object with the indicated member is returned or some other value is returned, in which case the code is in error.

Note

The overloaded arrow operator *must* return either a pointer to a class type or an object of a class type that defines its own operator arrow.

Exercises Section 14.7

Exercise 14.30: Add dereference and arrow operators to your `StrBlobPtr` class and to the `ConstStrBlobPtr` class that you defined in exercise 12.22 from § 12.1.6 (p. 476). Note that the operators in `constStrBlobPtr` must return `const` references because the `data` member in `constStrBlobPtr` points to a `const vector`.

Exercise 14.31: Our `StrBlobPtr` class does not define the copy constructor, assignment operator, or a destructor. Why is that okay?

Exercise 14.32: Define a class that holds a pointer to a `StrBlobPtr`. Define the overloaded arrow operator for that class.

14.8. Function-Call Operator

Classes that overload the call operator allow objects of its type to be used as if they were a function. Because such classes can also store state, they can be more flexible than ordinary functions.

As a simple example, the following `struct`, named `absInt`, has a call operator that returns the absolute value of its argument:

Click here to view code image

```
struct absInt {
    int operator()(int val) const {
        return val < 0 ? -val : val;
    }
};
```

This class defines a single operation: the function-call operator. That operator takes an argument of type `int` and returns the argument's absolute value.

We use the call operator by applying an argument list to an `absInt` object in a way that looks like a function call:

Click here to view code image

```
int i = -42;
absInt absObj;       // object that has a function-call operator
int ui = absObj(i);  // passes i to absObj.operator()
```

Even though `absObj` is an object, not a function, we can "call" this object. Calling an object runs its overloaded call operator. In this case, that operator takes an `int` value and returns its absolute value.

Note

The function-call operator must be a member function. A class may define multiple versions of the call operator, each of which must differ as to the number or types of their parameters.

Objects of classes that define the call operator are referred to as **function objects**. Such objects "act like functions" because we can call them.

Function-Object Classes with State

Like any other class, a function-object class can have additional members aside from `operator()`. Function-object classes often contain data members that are used to customize the operations in the call operator.

As an example, we'll define a class that prints a `string` argument. By default, our class will write to `cout` and will print a space following each `string`. We'll also let users of our class provide a different stream on which to write and provide a different separator. We can define this class as follows:

Click here to view code image

```
class PrintString {
public:
    PrintString(ostream &o = cout, char c = ' '):
        os(o), sep(c) { }
    void operator()(const string &s) const { os << s << sep;
}
private:
    ostream &os;    // stream on which to write
    char sep;       // character to print after each output
};
```

Our class has a constructor that takes a reference to an output stream and a character to use as the separator. It uses `cout` and a space as default arguments (§ 6.5.1, p. 236) for these parameters. The body of the function-call operator uses these members when it prints the given `string`.

When we define `PrintString` objects, we can use the defaults or supply our own values for the separator or output stream:

Click here to view code image

```
PrintString printer;    // uses the defaults; prints to cout
printer(s);             // prints s followed by a space on cout
PrintString errors(cerr, '\n');
errors(s);              // prints s followed by a newline on cerr
```

Function objects are most often used as arguments to the generic algorithms. For example, we can use the library `for_each` algorithm (§ 10.3.2, p. 391) and our `PrintString` class to print the contents of a container:

Click here to view code image

```
for_each(vs.begin(), vs.end(), PrintString(cerr, '\n'));
```

The third argument to `for_each` is a temporary object of type `PrintString` that we initialize from `cerr` and a newline character. The call to `for_each` will print each element in `vs` to `cerr` followed by a newline.

Exercises Section 14.8

Exercise 14.33: How many operands may an overloaded function-call operator take?

Exercise 14.34: Define a function-object class to perform an if-then-else operation: The call operator for this class should take three parameters. It should test its first parameter and if that test succeeds, it should return its second parameter; otherwise, it should return its third parameter.

Exercise 14.35: Write a class like `PrintString` that reads a line of input from an `istream` and returns a `string` representing what was read. If the read fails, return the empty `string`.

Exercise 14.36: Use the class from the previous exercise to read the standard input, storing each line as an element in a `vector`.

Exercise 14.37: Write a class that tests whether two values are equal. Use that object and the library algorithms to write a program to replace all instances of a given value in a sequence.

14.8.1. Lambdas Are Function Objects

In the previous section, we used a `PrintString` object as an argument in a call to `for_each`. This usage is similar to the programs we wrote in § 10.3.2 (p. 388) that used lambda expressions. When we write a lambda, the compiler translates that expression into an unnamed object of an unnamed class (§ 10.3.3, p. 392). The classes generated from a lambda contain an overloaded function-call operator. For example, the lambda that we passed as the last argument to `stable_sort`:

Click here to view code image

```
// sort words by size, but maintain alphabetical order for words of the same size
stable_sort(words.begin(), words.end(),
            [](const string &a, const string &b)
              { return a.size() < b.size();});
```

acts like an unnamed object of a class that would look something like

Click here to view code image

```
class ShorterString {
public:
       bool operator()(const string &s1, const string &s2)
const
    { return s1.size() < s2.size(); }
};
```

The generated class has a single member, which is a function-call operator that takes two `string`s and compares their lengths. The parameter list and function body are the same as the lambda. As we saw in § 10.3.3 (p. 395), by default, lambdas may not change their captured variables. As a result, by default, the function-call operator in a class generated from a lambda is a `const` member function. If the lambda is declared as `mutable`, then the call operator is not `const`.

We can rewrite the call to `stable_sort` to use this class instead of the lambda expression:

Click here to view code image

```
stable_sort(words.begin(), words.end(), ShorterString());
```

The third argument is a newly constructed `ShorterString` object. The code in `stable_sort` will "call" this object each time it compares two `string`s. When the object is called, it will execute the body of its call operator, returning `true` if the first `string`'s size is less than the second's.

Classes Representing Lambdas with Captures

As we've seen, when a lambda captures a variable by reference, it is up to the program to ensure that the variable to which the reference refers exists when the lambda is executed (§ 10.3.3, p. 393). Therefore, the compiler is permitted to use the reference directly without storing that reference as a data member in the generated class.

In contrast, variables that are captured by value are copied into the lambda (§ 10.3.3, p. 392). As a result, classes generated from lambdas that capture variables by value have data members corresponding to each such variable. These classes also have a constructor to initialize these data members from the value of the captured variables. As an example, in § 10.3.2 (p. 390), the lambda that we used to find the first `string` whose length was greater than or equal to a given bound:

Click here to view code image

```
// get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(),
            [sz](const string &a)
```

would generate a class that looks something like

Click here to view code image

```
class SizeComp {
     SizeComp(size_t n): sz(n) { } // parameter for each captured
variable
    // call operator with the same return type, parameters, and body as the lambda
    bool operator()(const string &s) const
        { return s.size() >= sz; }
private:
    size_t sz; // a data member for each variable captured by value
};
```

Unlike our `ShorterString` class, this class has a data member and a constructor to initialize that member. This synthesized class does not have a default constructor; to use this class, we must pass an argument:

Click here to view code image

```
// get an iterator to the first element whose size() is >= sz
auto wc = find_if(words.begin(), words.end(), SizeComp(sz));
```

Classes generated from a lambda expression have a deleted default constructor, deleted assignment operators, and a default destructor. Whether the class has a defaulted or deleted copy/move constructor depends in the usual ways on the types of the captured data members (§ 13.1.6, p. 508, and § 13.6.2, p. 537).

Exercises Section 14.8.1

Exercise 14.38: Write a class that tests whether the length of a given `string` matches a given bound. Use that object to write a program to report how many words in an input file are of sizes 1 through 10 inclusive.

Exercise 14.39: Revise the previous program to report the count of words that are sizes 1 through 9 and 10 or more.

Exercise 14.40: Rewrite the `biggies` function from § 10.3.2 (p. 391) to use function-object classes in place of lambdas.

Exercise 14.41: Why do you suppose the new standard added lambdas? Explain when you would use a lambda and when you would write a class instead.

14.8.2. Library-Defined Function Objects

The standard library defines a set of classes that represent the arithmetic, relational, and logical operators. Each class defines a call operator that applies the named operation. For example, the `plus` class has a function-call operator that applies + to a pair of operands; the `modulus` class defines a call operator that applies the binary `%` operator; the `equal_to` class applies `==`; and so on.

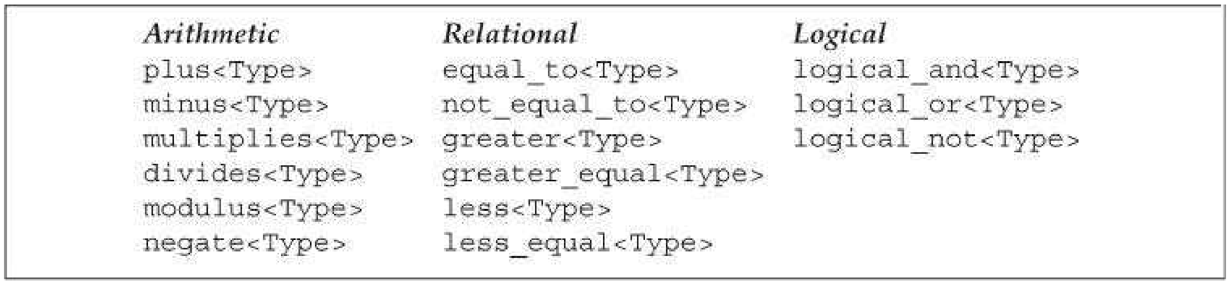
These classes are templates to which we supply a single type. That type specifies the parameter type for the call operator. For example, `plus<string>` applies the `string` addition operator to `string` objects; for `plus<int>` the operands are `int`s; `plus<Sales_data>` applies + to `Sales_data`s; and so on:

Click here to view code image

```
plus<int> intAdd;        // function object that can add two int values
negate<int> intNegate;   // function object that can negate an int value
// uses intAdd::operator(int, int) to add 10 and 20
int sum = intAdd(10, 20);          // equivalent to sum = 30
sum = intNegate(intAdd(10, 20));   // equivalent to sum = 30
// uses intNegate::operator(int) to generate -10 as the second parameter
// to intAdd::operator(int, int)
sum = intAdd(10, intNegate(10));  // sum = 0
```

These types, listed in Table 14.2, are defined in the `functional` header.

Table 14.2. Library Function Objects



Arithmetic	*Relational*	*Logical*
`plus<Type>`	`equal_to<Type>`	`logical_and<Type>`
`minus<Type>`	`not_equal_to<Type>`	`logical_or<Type>`
`multiplies<Type>`	`greater<Type>`	`logical_not<Type>`
`divides<Type>`	`greater_equal<Type>`	
`modulus<Type>`	`less<Type>`	
`negate<Type>`	`less_equal<Type>`	

Using a Library Function Object with the Algorithms

The function-object classes that represent operators are often used to override the default operator used by an algorithm. As we’ve seen, by default, the sorting algorithms use `operator<`, which ordinarily sorts the sequence into ascending order. To sort into descending order, we can pass an object of type `greater`. That class generates a call operator that invokes the greater-than operator of the underlying element type. For example, if `svec` is a `vector<string>`,

Click here to view code image

```
// passes a temporary function object that applies the < operator to two strings
sort(svec.begin(), svec.end(), greater<string>());
```

sorts the `vector` in descending order. The third argument is an unnamed object of type `greater<string>`. When `sort` compares elements, rather than applying the < operator for the element type, it will call the given `greater` function object. That object applies > to the `string` elements.

One important aspect of these library function objects is that the library guarantees that they will work for pointers. Recall that comparing two unrelated pointers is undefined (§ 3.5.3, p. 120). However, we might want to `sort` a `vector` of pointers based on their addresses in memory. Although it would be undefined for us to do so directly, we can do so through one of the library function objects:

Click here to view code image

```
vector<string *> nameTable;  // vector of pointers
// error: the pointers in nameTable are unrelated, so < is undefined
sort(nameTable.begin(), nameTable.end(),
     [](string *a, string *b) { return a < b; });
// ok: library guarantees that less on pointer types is well defined
sort(nameTable.begin(), nameTable.end(), less<string*>());
```

It is also worth noting that the associative containers use `less<key_type>` to order their elements. As a result, we can define a `set` of pointers or use a pointer as the key in a `map` without specifying `less` directly.

Exercises Section 14.8.2

Exercise 14.42: Using library function objects and adaptors, define an expression to

(a) Count the number of values that are greater than 1024

(b) Find the first string that is not equal to `pooh`

(c) Multiply all values by 2

Exercise 14.43: Using library function objects, determine whether a given `int` value is divisible by any element in a container of `int`s.

14.8.3. Callable Objects and `function`

C++ has several kinds of callable objects: functions and pointers to functions, lambdas (§ 10.3.2, p. 388), objects created by `bind` (§ 10.3.4, p. 397), and classes that overload the function-call operator.

Like any other object, a callable object has a type. For example, each lambda has its own unique (unnamed) class type. Function and function-pointer types vary by their return type and argument types, and so on.

However, two callable objects with different types may share the same **call signature**. The call signature specifies the type returned by a call to the object and the argument type(s) that must be passed in the call. A call signature corresponds to a function type. For example:

```
int(int, int)
```

is a function type that takes two `int`s and returns an `int`.

Different Types Can Have the Same Call Signature

Sometimes we want to treat several callable objects that share a call signature as if they had the same type. For example, consider the following different types of callable objects:

Click here to view code image

```
// ordinary function
int add(int i, int j) { return i + j; }
// lambda, which generates an unnamed function-object class
```

```
auto mod = [](int i, int j) { return i % j; };
// function-object class
struct div {
    int operator()(int denominator, int divisor) {
        return denominator / divisor;
    }
};
```

Each of these callables applies an arithmetic operation to its parameters. Even though each has a distinct type, they all share the same call signature:

```
int(int, int)
```

We might want to use these callables to build a simple desk calculator. To do so, we'd want to define a **function table** to store "pointers" to these callables. When the program needs to execute a particular operation, it will look in the table to find which function to call.

In C++, function tables are easy to implement using a `map`. In this case, we'll use a `string` corresponding to an operator symbol as the key; the value will be the function that implements that operator. When we want to evaluate a given operator, we'll index the `map` with that operator and call the resulting element.

If all our functions were freestanding functions, and assuming we were handling only binary operators for type `int`, we could define the `map` as

Click here to view code image

```
// maps an operator to a pointer to a function taking two ints and returning an int
map<string, int(*)(int,int)> binops;
```

We could put a pointer to `add` into `binops` as follows:

Click here to view code image

```
// ok: add is a pointer to function of the appropriate type
binops.insert({"+", add}); // {"+", add} is a pair § 11.2.3 (p. 426)
```

However, we can't store `mod` or `div` in `binops`:

Click here to view code image

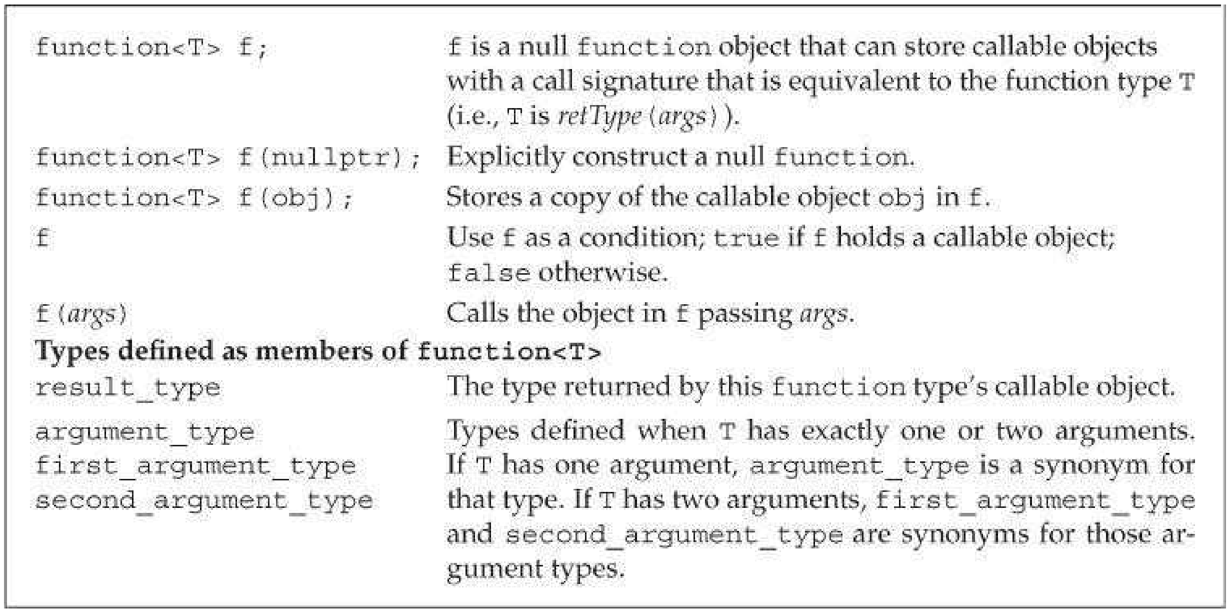
```
binops.insert({"%", mod}); // error: mod is not a pointer to function
```

The problem is that `mod` is a lambda, and each lambda has its own class type. That type does not match the type of the values stored in `binops`.

The Library function Type

We can solve this problem using a new library type named **function** that is defined in the `functional` header; Table 14.3 (p. 579) lists the operations defined by `function`.

Table 14.3. Operations on function



`function<T> f;`	`f` is a null `function` object that can store callable objects with a call signature that is equivalent to the function type `T` (i.e., `T` is *retType* (*args*)).
`function<T> f(nullptr);`	Explicitly construct a null `function`.
`function<T> f(obj);`	Stores a copy of the callable object `obj` in `f`.
`f`	Use `f` as a condition; `true` if `f` holds a callable object; `false` otherwise.
`f(`*args*`)`	Calls the object in `f` passing *args*.
Types defined as members of `function<T>`	
`result_type`	The type returned by this `function` type's callable object.
`argument_type` `first_argument_type` `second_argument_type`	Types defined when `T` has exactly one or two arguments. If `T` has one argument, `argument_type` is a synonym for that type. If `T` has two arguments, `first_argument_type` and `second_argument_type` are synonyms for those argument types.

`function` is a template. As with other templates we've used, we must specify additional information when we create a `function` type. In this case, that information is the call signature of the objects that this particular `function` type can represent. As with other templates, we specify the type inside angle brackets:

```
function<int(int, int)>
```

Here we've declared a `function` type that can represent callable objects that return an `int` result and have two `int` parameters. We can use that type to represent any of our desk calculator types:

Click here to view code image

```
function<int(int, int)> f1 = add;     // function pointer
function<int(int, int)> f2 = div();   // object of a function-object
class
function<int(int, int)> f3 = [](int  i, int j) // lambda
                             { return i * j; };
cout << f1(4,2) << endl; // prints 6
cout << f2(4,2) << endl; // prints 2
cout << f3(4,2) << endl; // prints 8
```

We can now redefine our `map` using this `function` type:

Click here to view code image

```
// table of callable objects corresponding to each binary operator
// all the callables must take two ints and return an int
// an element can be a function pointer, function object, or lambda
map<string, function<int(int, int)>> binops;
```

We can add each of our callable objects, be they function pointers, lambdas, or function objects, to this `map`:

Click here to view code image

```
map<string, function<int(int, int)>> binops = {
    {"+", add},                           // function pointer
    {"-", std::minus<int>()},             // library function object
    {"/",  div()},                        // user-defined function object
     {"*", [](int i, int j) { return i * j; }}, // unnamed
lambda
    {"%", mod} };                         // named lambda object
```

Our `map` has five elements. Although the underlying callable objects all have different types from one another, we can store each of these distinct types in the common `function<int(int, int)>` type.

As usual, when we index a `map`, we get a reference to the associated value. When we index `binops`, we get a reference to an object of type `function`. The `function` type overloads the call operator. That call operator takes its own arguments and passes them along to its stored callable object:

Click here to view code image

```
binops["+"](10, 5); // calls add(10, 5)
binops["-"](10, 5); // uses the call operator of the minus<int> object
binops["/"](10, 5); // uses the call operator of the div object
binops["*"](10, 5); // calls the lambda function object
binops["%"](10, 5); // calls the lambda function object
```

Here we call each of the operations stored in `binops`. In the first call, the element we get back holds a function pointer that points to our `add` function. Calling `binops["+"](10, 5)` uses that pointer to call `add`, passing it the values `10` and `5`. In the next call, `binops["-"]`, returns a `function` that stores an object of type `std::minus<int>`. We call that object's call operator, and so on.

Overloaded Functions and `function`

We cannot (directly) store the name of an overloaded function in an object of type `function`:

Click here to view code image

```
int add(int i, int j) { return i + j; }
Sales_data add(const Sales_data&, const Sales_data&);
map<string, function<int(int, int)>> binops;
```

```
binops.insert( {"+", add} ); // error: which add?
```

One way to resolve the ambiguity is to store a function pointer (§ 6.7, p. 247) instead of the name of the function:

Click here to view code image

```
int (*fp)(int,int) = add; // pointer to the version of add that takes two ints
binops.insert( {"+", fp} ); // ok: fp points to the right version of add
```

Alternatively, we can use a lambda to disambiguate:

Click here to view code image

```
// ok: use a lambda to disambiguate which version of add we want to use
binops.insert( {"+", [](int a, int b) {return add(a, b);} } );
```

The call inside the lambda body passes two `int`s. That call can match only the version of `add` that takes two `int`s, and so that is the function that is called when the lambda is executed.

Note

The `function` class in the new library is not related to classes named `unary_function` and `binary_function` that were part of earlier versions of the library. These classes have been deprecated by the more general `bind` function (§ 10.3.4, p. 401).

Exercises Section 14.8.3

Exercise 14.44: Write your own version of a simple desk calculator that can handle binary operations.

14.9. Overloading, Conversions, and Operators

In § 7.5.4 (p. 294) we saw that a `nonexplicit` constructor that can be called with one argument defines an implicit conversion. Such constructors convert an object from the argument's type *to* the class type. We can also define conversions *from* the class type. We define a conversion from a class type by defining a conversion operator. Converting constructors and conversion operators define **class-type conversions**. Such conversions are also referred to as **user-defined conversions**.

14.9.1. Conversion Operators

A **conversion operator** is a special kind of member function that converts a value of a class type to a value of some other type. A conversion function typically has the general form

```
operator type() const;
```

where *type* represents a type. Conversion operators can be defined for any type (other than `void`) that can be a function return type (§ 6.1, p. 204). Conversions to an array or a function type are not permitted. Conversions to pointer types—both data and function pointers—and to reference types are allowed.

Conversion operators have no explicitly stated return type and no parameters, and they must be defined as member functions. Conversion operations ordinarily should not change the object they are converting. As a result, conversion operators usually should be defined as `const` members.

Note

A conversion function must be a member function, may not specify a return type, and must have an empty parameter list. The function usually should be `const`.

Defining a Class with a Conversion Operator

As an example, we'll define a small class that represents an integer in the range of 0 to 255:

Click here to view code image

```
class SmallInt {
public:
    SmallInt(int i = 0): val(i)
    {
        if (i < 0 || i > 255)
            throw std::out_of_range("Bad SmallInt value");
    }
    operator int() const { return val; }
private:
    std::size_t val;
};
```

Our `SmallInt` class defines conversions *to* and *from* its type. The constructor converts values of arithmetic type to a `SmallInt`. The conversion operator converts `SmallInt` objects to `int`:

Click here to view code image

```
SmallInt si;
si = 4; // implicitly converts 4 to SmallInt then calls SmallInt::operator=
si + 3; // implicitly converts si to int followed by integer addition
```

Although the compiler will apply only one user-defined conversion at a time (§ 4.11.2, p. 162), an implicit user-defined conversion can be preceded or followed by a standard (built-in) conversion (§ 4.11.1, p. 159). As a result, we can pass any arithmetic type to the `SmallInt` constructor. Similarly, we can use the converion operator to convert a `SmallInt` to an `int` and then convert the resulting `int` value to another arithmetic type:

Click here to view code image

```
// the double argument is converted to int using the built-in conversion
SmallInt si = 3.14; // calls the SmallInt(int) constructor
// the SmallInt conversion operator converts si to int;
si + 3.14; // that int is converted to double using the built-in conversion
```

Because conversion operators are implicitly applied, there is no way to pass arguments to these functions. Hence, conversion operators may not be defined to take parameters. Although a conversion function does not specify a return type, each conversion function must return a value of its corresponding type:

Click here to view code image

```
class SmallInt;
operator int(SmallInt&);                    // error: nonmember
class SmallInt {
public:
    int operator int() const;               // error: return type
    operator int(int = 0) const;            // error: parameter list
     operator int*() const { return 42; } // error: 42 is not a
pointer
};
```

Caution: Avoid Overuse of Conversion Functions

As with using overloaded operators, judicious use of conversion operators can greatly simplify the job of a class designer and make using a class easier. However, some conversions can be misleading. Conversion operators are misleading when there is no obvious single mapping between the class type and the conversion type.

For example, consider a class that represents a `Date`. We might think it would be a good idea to provide a conversion from `Date` to `int`. However, what value should the conversion function return? The function might return a decimal representation of the year, month, and day. For example, July 30, 1989 might be represented as the `int` value 19800730. Alternatively, the conversion operator might return an `int` representing the number of days that have elapsed since some epoch point, such as January 1, 1970. Both these conversions have the desirable property that later dates correspond to larger integers, and so either might be useful.

The problem is that there is no single one-to-one mapping between an object of type `Date` and a value of type `int`. In such cases, it is better not to define the conversion operator. Instead, the class ought to define one or more ordinary members to extract the information in these various forms.

Conversion Operators Can Yield Suprising Results

In practice, classes rarely provide conversion operators. Too often users are more likely to be surprised if a conversion happens automatically than to be helped by the existence of the conversion. However, there is one important exception to this rule of thumb: It is not uncommon for classes to define conversions to `bool`.

Under earlier versions of the standard, classes that wanted to define a conversion to `bool` faced a problem: Because `bool` is an arithmetic type, a class-type object that is converted to `bool` can be used in any context where an arithmetic type is expected. Such conversions can happen in surprising ways. In particular, if `istream` had a conversion to `bool`, the following code would compile:

Click here to view code image

```
int i = 42;
cin << i; // this code would be legal if the conversion to bool were not explicit!
```

This program attempts to use the output operator on an input stream. There is no << defined for `istream`, so the code is almost surely in error. However, this code could use the `bool` conversion operator to convert `cin` to `bool`. The resulting `bool` value would then be promoted to `int` and used as the left-hand operand to the built-in version of the left-shift operator. The promoted `bool` value (either 1 or 0) would be shifted left 42 positions.

explicit Conversion Operators

C++ 11

To prevent such problems, the new standard introduced **explicit conversion operators**:

Click here to view code image

```
class SmallInt {
public:
    // the compiler won't automatically apply this conversion
    explicit operator int() const { return val; }
    // other members as before
};
```

As with an `explicit` constructor (§ 7.5.4, p. 296), the compiler won't (generally) use an `explicit` conversion operator for implicit conversions:

Click here to view code image

```
    SmallInt si = 3;  // ok: the SmallInt constructor is not explicit
    si + 3; // error: implicit is conversion required, but operator int is explicit
    static_cast<int>(si) + 3; // ok: explicitly request the conversion
```

If the conversion operator is `explicit`, we can still do the conversion. However, with one exception, we must do so explicitly through a cast.

The exception is that the compiler will apply an `explicit` conversion to an expression used as a condition. That is, an `explicit` conversion will be used implicitly to convert an expression used as

- The condition of an `if, while`, or do statement
- The condition expression in a `for` statement header
- An operand to the logical NOT (`!`), OR (`||`), or AND (`&&`) operators
- The condition expression in a conditional (`?:`) operator

Conversion to bool

In earlier versions of the library, the IO types defined a conversion to `void*`. They did so to avoid the kinds of problems illustrated above. Under the new standard, the IO library instead defines an `explicit` conversion to `bool`.

Whenever we use a stream object in a condition, we use the `operator bool` that is defined for the IO types. For example,

```
    while (std::cin >> value)
```

The condition in the `while` executes the input operator, which reads into `value` and returns `cin`. To evaluate the condition, `cin` is implicitly converted by the `istream operator bool` conversion function. That function returns `true` if the condition state of `cin` is `good` (§ 8.1.2, p. 312), and `false` otherwise.

Best Practices

Conversion to `bool` is usually intended for use in conditions. As a result, `operator bool` ordinarily should be defined as `explicit`.

Exercises Section 14.9.1

Exercise 14.45: Write conversion operators to convert a `Sales_data` to `string` and to `double`. What values do you think these operators should return?

Exercise 14.46: Explain whether defining these `Sales_data` conversion operators is a good idea and whether they should be `explicit`.

Exercise 14.47: Explain the difference between these two conversion operators:

```
struct Integral {
    operator const int();
    operator int() const;
};
```

Exercise 14.48: Determine whether the class you used in exercise 7.40 from § 7.5.1 (p. 291) should have a conversion to `bool`. If so, explain why, and explain whether the operator should be `explicit`. If not, explain why not.

Exercise 14.49: Regardless of whether it is a good idea to do so, define a conversion to `bool` for the class from the previous exercise.

14.9.2. Avoiding Ambiguous Conversions

If a class has one or more conversions, it is important to ensure that there is only one way to convert from the class type to the target type. If there is more than one way to perform a conversion, it will be hard to write unambiguous code.

There are two ways that multiple conversion paths can occur. The first happens when two classes provide mutual conversions. For example, mutual conversions exist when a class `A` defines a converting constructor that takes an object of class `B` and `B` itself defines a conversion operator to type `A`.

The second way to generate multiple conversion paths is to define multiple conversions from or to types that are themselves related by conversions. The most obvious instance is the built-in arithmetic types. A given class ordinarily ought to define at most one conversion to or from an arithmetic type.

Warning

Ordinarily, it is a bad idea to define classes with mutual conversions or to define conversions to or from two arithmetic types.

Argument Matching and Mutual Conversions

In the following example, we've defined two ways to obtain an A from a B: either by using B's conversion operator or by using the A constructor that takes a B:

Click here to view code image

```
// usually a bad idea to have mutual conversions between two class types
struct B;
struct A {
    A() = default;
    A(const B&);        // converts a B to an A
    // other members
};
struct B {
    operator A() const; // also converts a B to an A
    // other members
};
A f(const A&);
B b;
A a = f(b); // error ambiguous: f(B::operator A())
            //          or f(A::A(const B&))
```

Because there are two ways to obtain an A from a B, the compiler doesn't know which conversion to run; the call to f is ambiguous. This call can use the A constructor that takes a B, or it can use the B conversion operator that converts a B to an A. Because these two functions are equally good, the call is in error.

If we want to make this call, we have to explicitly call the conversion operator or the constructor:

Click here to view code image

```
A a1 = f(b.operator A()); // ok: use B's conversion operator
A a2 = f(A(b));           // ok: use A's constructor
```

Note that we can't resolve the ambiguity by using a cast—the cast itself would have the same ambiguity.

Ambiguities and Multiple Conversions to Built-in Types

Ambiguities also occur when a class defines multiple conversions to (or from) types that are themselves related by conversions. The easiest case to illustrate—and one that is particularly problematic—is when a class defines constructors from or conversions to more than one arithmetic type.

For example, the following class has converting constructors from two different arithmetic types, and conversion operators to two different arithmetic types:

Click here to view code image

```
struct A {
    A(int = 0);    // usually a bad idea to have two
    A(double);     // conversions from arithmetic types
    operator int() const;     // usually a bad idea to have two
    operator double() const; // conversions to arithmetic types
    //     other members

};
void f2(long double);
A a;
f2(a); // error ambiguous:  f(A::operator int())
       //                or  f(A::operator double())
long lg;
A a2(lg); // error ambiguous:  A::A(int)  or  A::A(double)
```

In the call to `f2`, neither conversion is an exact match to `long double`. However, either conversion can be used, followed by a standard conversion to get to `long double`. Hence, neither conversion is better than the other; the call is ambiguous.

We encounter the same problem when we try to initialize `a2` from a `long`. Neither constructor is an exact match for `long`. Each would require that the argument be converted before using the constructor:

- Standard `long` to `double` conversion followed by `A(double)`
- Standard `long` to `int` conversion followed by `A(int)`

These conversion sequences are indistinguishable, so the call is ambiguous.

The call to `f2`, and the initialization of `a2`, are ambiguous because the standard conversions that were needed had the same rank (§ 6.6.1, p. 245). When a user-defined conversion is used, the rank of the standard conversion, if any, is used to select the best match:

Click here to view code image

```
short s = 42;
// promoting  short  to  int  is better than converting  short  to  double
A a3(s);  // uses  A::A(int)
```

In this case, promoting a `short` to an `int` is preferred to converting the `short` to a `double`. Hence `a3` is constructed using the `A::A(int)` constructor, which is run on the (promoted) value of `s`.

Note

When two user-defined conversions are used, the rank of the standard conversion, if any, *preceding* or *following* the conversion function is used to select the best match.

Overloaded Functions and Converting Constructors

Choosing among multiple conversions is further complicated when we call an overloaded function. If two or more conversions provide a viable match, then the conversions are considered equally good.

As one example, ambiguity problems can arise when overloaded functions take parameters that differ by class types that define the same converting constructors:

Caution: Conversions and Operators

Correctly designing the overloaded operators, conversion constructors, and conversion functions for a class requires some care. In particular, ambiguities are easy to generate if a class defines both conversion operators and overloaded operators. A few rules of thumb can be helpful:

• Don't define mutually converting classes—if class `Foo` has a constructor that takes an object of class `Bar`, do not give `Bar` a conversion operator to type `Foo`.

• Avoid conversions to the built-in arithmetic types. In particular, if you do define a conversion to an arithmetic type, then

– Do not define overloaded versions of the operators that take arithmetic types. If users need to use these operators, the conversion operation will convert objects of your type, and then the built-in operators can be used.

– Do not define a conversion to more than one arithmetic type. Let the standard conversions provide conversions to the other arithmetic types.

The easiest rule of all: With the exception of an `explicit` conversion to `bool`, avoid defining conversion functions and limit non`explicit` constructors to those that are "obviously right."

Click here to view code image

```
struct C {
    C(int);
    //  other members
};
struct D {
    D(int);
    //  other members
};
void manip(const C&);
void manip(const D&);
manip(10); //  error ambiguous:  manip(C(10))  or  manip(D(10))
```

Here both `C` and `D` have constructors that take an `int`. Either constructor can be used to match a version of `manip`. Hence, the call is ambiguous: It could mean convert the `int` to `C` and call the first version of `manip`, or it could mean convert the `int` to `D` and call the second version.

The caller can disambiguate by explicitly constructing the correct type:

Click here to view code image

```
manip(C(10)); // ok: calls manip(const C&)
```

Warning

Needing to use a constructor or a cast to convert an argument in a call to an overloaded function frequently is a sign of bad design.

Overloaded Functions and User-Defined Conversion

In a call to an overloaded function, if two (or more) user-defined conversions provide a viable match, the conversions are considered equally good. The rank of any standard conversions that might or might not be required is not considered. Whether a built-in conversion is also needed is considered only if the overload set can be matched *using the same conversion function.*

For example, our call to `manip` would be ambiguous even if one of the classes defined a constructor that required a standard conversion for the argument:

Click here to view code image

```
struct E {
    E(double);
    // other members
};
void manip2(const C&);
void manip2(const E&);
// error ambiguous: two different user-defined conversions could be used
manip2(10); // manip2(C(10) or manip2(E(double(10)))
```

In this case, `C` has a conversion from `int` and `E` has a conversion from `double`. For the call `manip2(10)`, both `manip2` functions are viable:

• `manip2(const C&)` is viable because `C` has a converting constructor that takes an `int`. That constructor is an exact match for the argument.

• `manip2(const E&)` is viable because `E` has a converting constructor that takes a `double` and we can use a standard conversion to convert the `int` argument in order to use that converting constructor.

Because calls to the overloaded functions require *different* user-defined conversions from one another, this call is ambiguous. In particular, even though one of the calls requires a standard conversion and the other is an exact match, the compiler will still flag this call as an error.

Note

In a call to an overloaded function, the rank of an additional standard conversion (if any) matters only if the viable functions require the same user-defined conversion. If different user-defined conversions are needed, then the call is ambiguous.

14.9.3. Function Matching and Overloaded Operators

Overloaded operators are overloaded functions. Normal function matching (§ 6.4, p. 233) is used to determine which operator—built-in or overloaded—to apply to a given expression. However, when an operator function is used in an expression, the set of candidate functions is broader than when we call a function using the call operator. If a has a class type, the expression a *sym* b might be

Click here to view code image

```
a.operatorsym (b); // a has operatorsym as a member function
operatorsym(a, b); // operatorsym is an ordinary function
```

Unlike ordinary function calls, we cannot use the form of the call to distinquish whether we're calling a nonmember or a member function.

Exercises Section 14.9.2

Exercise 14.50: Show the possible class-type conversion sequences for the initializations of ex1 and ex2. Explain whether the initializations are legal or not.

Click here to view code image

```
struct LongDouble {
    LongDouble(double = 0.0);
    operator double();
    operator float();
};
LongDouble ldObj;
int ex1 = ldObj;
float ex2 = ldObj;
```

Exercise 14.51: Show the conversion sequences (if any) needed to call each version of `calc` and explain why the best viable function is selected.

Click here to view code image

```
void calc(int);
void calc(LongDouble);
double dval;
calc(dval); // which calc?
```

When we use an overloaded operator with an operand of class type, the candidate functions include ordinary nonmember versions of that operator, as well as the built-in versions of the operator. Moreover, if the left-hand operand has class type, the overloaded versions of the operator, if any, defined by that class are also included.

When we call a named function, member and nonmember functions with the same name do *not* overload one another. There is no overloading because the syntax we use to call a named function distinguishes between member and nonmember functions. When a call is through an object of a class type (or through a reference or pointer to such an object), then only the member functions of that class are considered. When we use an overloaded operator in an expression, there is nothing to indicate whether we're using a member or nonmember function. Therefore, both member and nonmember versions must be considered.

Note

The set of candidate functions for an operator used in an expression can contain both nonmember and member functions.

As an example, we'll define an addition operator for our `SmallInt` class:

Click here to view code image

```
class SmallInt {
    friend
    SmallInt operator+(const SmallInt&, const SmallInt&);
public:
    SmallInt(int = 0);                   // conversion from int
    operator int() const { return val; } // conversion to int
private:
    std::size_t val;
};
```

We can use this class to add two `SmallInt`s, but we will run into ambiguity problems if we attempt to perform mixed-mode arithmetic:

Click here to view code image

```
SmallInt s1, s2;
SmallInt s3 = s1 + s2;  // uses overloaded operator+
int i = s3 + 0;         // error: ambiguous
```

The first addition uses the overloaded version of `+` that takes two `SmallInt` values. The second addition is ambiguous, because we can convert `0` to a `SmallInt` and use the `SmallInt` version of `+`, or convert `s3` to `int` and use the built-in addition operator on `int`s.

Warning

Providing both conversion functions to an arithmetic type and overloaded operators for the same class type may lead to ambiguities between the overloaded operators and the built-in operators.

Exercises Section 14.9.3

Exercise 14.52: Which `operator+`, if any, is selected for each of the addition expressions? List the candidate functions, the viable functions, and the type conversions on the arguments for each viable function:

Click here to view code image

```
struct LongDouble {
    // member operator+ for illustration purposes; + is usually a nonmember
    LongDouble operator+(const SmallInt&);
    // other members as in § 14.9.2 (p. 587)
};
LongDouble operator+(LongDouble&, double);
SmallInt si;
LongDouble ld;
ld = si + ld;
ld = ld + si;
```

Exercise 14.53: Given the definition of `SmallInt` on page 588, determine whether the following addition expression is legal. If so, what addition operator is used? If not, how might you change the code to make it legal?

```
SmallInt s1;
double d = s1 + 3.14;
```

Chapter Summary

An overloaded operator must either be a member of a class or have at least one operand of class type. Overloaded operators have the same number of operands, associativity, and precedence as the corresponding operator when applied to the built-in types. When an operator is defined as a member, its implicit `this` pointer is bound to the first operand. The assignment, subscript, function-call, and arrow operators must be class members.

Objects of classes that overload the function-call operator, `operator()`, are known as “function objects.” Such objects are often used in combination with the standard algorithms. Lambda expressions are succinct ways to define simple function-object classes.

A class can define conversions to or from its type that are used automatically. Non`explicit` constructors that can be called with a single argument define conversions from the parameter type to the class type; non`explicit` conversion operators define conversions from the class type to other types.

Defined Terms

call signature Represents the interface of a callable object. A call signature includes the return type and a comma-separated list of argument types enclosed in parentheses.

class-type conversion Conversions to or from class types are defined by constructors and conversion operators, respectively. Non`explicit` constructors that take a single argument define a conversion from the argument type to the class type. Conversion operators define conversions from the class type to the specified type.

conversion operator A member function that defines a conversions from the class type to another type. A conversion operator must be a member of the class from which it converts and is usually a `const` member. These operators have no return type and take no parameters. They return a value convertible to the type of the conversion operator. That is, `operator int` returns an `int`, `operator string` returns a `string`, and so on.

explicit conversion operator Conversion operator preceeded by the `explicit` keyword. Such operators are used for implicit conversions only in conditions.

function object Object of a class that defines an overloaded call operator. Function objects can be used where functions are normally expected.

function table Container, often a `map` or a `vector`, that holds values that can be called.

function template Library template that can represent any callable type.

overloaded operator Function that redefines the meaning of one of the built-in operators. Overloaded operator functions have the name `operator` followed by the symbol being defined. Overloaded operators must have at least one operand of class type. Overloaded operators have the same precedence, associativity and number of operands as their built-in counterparts.

user-defined conversion A synonym for class-type conversion.

Chapter 15. Object-Oriented Programming

Contents

Object-oriented programming is based on three fundamental concepts: data abstraction, which we covered in Chapter 7, and inheritance and dynamic binding, which we'll cover in this chapter.

Inheritance and dynamic binding affect how we write our programs in two ways: They make it easier to define new classes that are similar, but not identical, to other classes, and they make it easier for us to write programs that can ignore the details of how those similar types differ.

Many applications include concepts that are related to but slightly different from one another. For example, our bookstore might offer different pricing strategies for different books. Some books might be sold only at a given price. Others might be sold subject to a discount. We might give a discount to purchasers who buy a specified number of copies of the book. Or we might give a discount for only the first few copies purchased but charge full price for any bought beyond a given limit, and so on. Object-oriented programming (OOP) is a good match to this kind of application.

15.1. OOP: An Overview

The key ideas in **object-oriented programming** are data abstraction, inheritance, and dynamic binding. Using data abstraction, we can define classes that separate interface from implementation (Chapter 7). Through inheritance, we can define classes that model the relationships among similar types. Through dynamic binding, we can use objects of these types while ignoring the details of how they differ.

Inheritance

Classes related by **inheritance** form a hierarchy. Typically there is a **base class** at the root of the hierarchy, from which the other classes inherit, directly or indirectly. These inheriting classes are known as **derived classes**. The base class defines those members that are common to the types in the hierarchy. Each derived class defines those members that are specific to the derived class itself.

To model our different kinds of pricing strategies, we'll define a class named `Quote`, which will be the base class of our hierarchy. A `Quote` object will represent undiscounted books. From `Quote` we will inherit a second class, named `Bulk_quote`, to represent books that can be sold with a quantity discount.

These classes will have the following two member functions:

- `isbn()`, which will return the ISBN. This operation does not depend on the specifics of the inherited class(es); it will be defined only in class `Quote`.
- `net_price(size_t)`, which will return the price for purchasing a specified number of copies of a book. This operation is type specific; both `Quote` and `Bulk_quote` will define their own version of this function.

In C++, a base class distinguishes functions that are type dependent from those that it expects its derived classes to inherit without change. The base class defines as **virtual** those functions it expects its derived classes to define for themselves. Using this knowledge, we can start to write our `Quote` class:

Click here to view code image

```
class Quote {
public:
    std::string isbn() const;
    virtual double net_price(std::size_t n) const;
};
```

A derived class must specify the class(es) from which it intends to inherit. It does so in a **class derivation list**, which is a colon followed by a comma-separated list of base classes each of which may have an optional access specifier:

Click here to view code image

```
class Bulk_quote : public Quote { // Bulk_quote inherits from Quote
public:
    double net_price(std::size_t) const override;
};
```

Because `Bulk_quote` uses `public` in its derivation list, we can use objects of type `Bulk_quote` *as if* they were `Quote` objects.

A derived class must include in its own class body a declaration of all the virtual functions it intends to define for itself. A derived class may include the `virtual` keyword on these functions but is not required to do so. For reasons we'll explain in § 15.3 (p. 606), the new standard lets a derived class explicitly note that it intends a member function to **override** a virtual that it inherits. It does so by specifying `override` after its parameter list.

Dynamic Binding

Through **dynamic binding**, we can use the same code to process objects of either type `Quote` or `Bulk_quote` interchangeably. For example, the following function prints the total price for purchasing the given number of copies of a given book:

Click here to view code image

```
// calculate and print the price for the given number of copies, applying any discounts
double print_total(ostream &os,
                   const Quote &item, size_t n)
{
    // depending on the type of the object bound to the item parameter
    // calls either Quote::net_price or Bulk_quote::net_price
    double ret = item.net_price(n);
    os << "ISBN: " << item.isbn() // calls Quote::isbn
       << " # sold: " << n << " total due: " << ret << endl;
     return ret;
}
```

This function is pretty simple—it prints the results of calling `isbn` and `net_price` on its parameter and returns the value calculated by the call to `net_price`.

Nevertheless, there are two interesting things about this function: For reasons we'll explain in § 15.2.3 (p. 601), because the `item` parameter is a reference to `Quote`, we can call this function on either a `Quote` object or a `Bulk_quote` object. And, for reasons we'll explain in § 15.2.1 (p. 594), because `net_price` is a virtual function, and because `print_total` calls `net_price` through a reference, the version of `net_price` that is run will depend on the type of the object that we pass to `print_total`:

Click here to view code image

```
// basic has type Quote; bulk has type Bulk_quote
print_total(cout, basic, 20); // calls Quote version of net_price
print_total(cout, bulk, 20);  // calls Bulk_quote version of net_price
```

The first call passes a `Quote` object to `print_total`. When `print_total` calls `net_price`, the `Quote` version will be run. In the next call, the argument is a

Bulk_quote, so the Bulk_quote version of net_price (which applies a discount) will be run. Because the decision as to which version to run depends on the type of the argument, that decision can't be made until run time. Therefore, dynamic binding is sometimes known as **run-time binding**.

Note

In C++, dynamic binding happens when a virtual function is called through a reference (or a pointer) to a base class.

15.2. Defining Base and Derived Classes

In many, but not all, ways base and derived classes are defined like other classes we have already seen. In this section, we'll cover the basic features used to define classes related by inheritance.

15.2.1. Defining a Base Class

We'll start by completing the definition of our Quote class:

Click here to view code image

```
class Quote {
public:
    Quote() = default;   //  = default  see § 7.1.4 (p. 264)
    Quote(const std::string &book, double sales_price):
                     bookNo(book), price(sales_price) { }
    std::string isbn() const { return bookNo; }
    //  returns the total sales price for the specified number of items
    //  derived classes will override and apply different discount algorithms
    virtual double net_price(std::size_t n) const
                { return n * price; }
    virtual ~Quote() = default; //  dynamic binding for the destructor
private:
    std::string bookNo; // ISBN  number of this item
protected:
    double price = 0.0; //  normal, undiscounted price
};
```

The new parts in this class are the use of virtual on the net_price function and the destructor, and the protected access specifier. We'll explain virtual destructors in §15.7.1 (p. 622), but for now it is worth noting that classes used as the root of an inheritance hierarchy almost always define a virtual destructor.

Note

Base classes ordinarily should define a virtual destructor. Virtual destructors are needed even if they do no work.

Member Functions and Inheritance

Derived classes inherit the members of their base class. However, a derived class needs to be able to provide its own definition for operations, such as `net_price`, that are type dependent. In such cases, the derived class needs to **override** the definition it inherits from the base class, by providing its own definition.

In C++, a base class must distinguish the functions it expects its derived classes to override from those that it expects its derived classes to inherit without change. The base class defines as **virtual** those functions it expects its derived classes to override. When we call a virtual function *through a pointer or reference*, the call will be dynamically bound. Depending on the type of the object to which the reference or pointer is bound, the version in the base class or in one of its derived classes will be executed.

A base class specifies that a member function should be dynamically bound by preceding its declaration with the keyword `virtual`. Any non`static` member function (§7.6, p. 300), other than a constructor, may be virtual. The `virtual` keyword appears only on the declaration inside the class and may not be used on a function definition that appears outside the class body. A function that is declared as `virtual` in the base class is implicitly `virtual` in the derived classes as well. We'll have more to say about virtual functions in §15.3 (p. 603).

Member functions that are not declared as `virtual` are resolved at compile time, not run time. For the `isbn` member, this is exactly the behavior we want. The `isbn` function does not depend on the details of a derived type. It behaves identically when run on a `Quote` or `Bulk_quote` object. There will be only one version of the `isbn` function in our inheritance hierarchy. Thus, there is no question as to which function to run when we call `isbn()`.

Access Control and Inheritance

A derived class inherits the members defined in its base class. However, the member functions in a derived class may not necessarily access the members that are inherited from the base class. Like any other code that uses the base class, a derived class may access the `public` members of its base class but may not access the `private` members. However, sometimes a base class has members that it wants to let its derived classes use while still prohibiting access to those same members by other users. We specify such members after a **protected** access specifier.

Our `Quote` class expects its derived classes to define their own `net_price` function. To do so, those classes need access to the `price` member. As a result, `Quote` defines that member as `protected`. Derived classes are expected to access `bookNo` in the same way as ordinary users—by calling the `isbn` function. Hence, the `bookNo` member is `private` and is inaccessible to classes that inherit from `Quote`. We'll have more to say about `protected` members in §15.5 (p. 611).

Exercises Section 15.2.1

Exercise 15.1: What is a virtual member?

Exercise 15.2: How does the `protected` access specifier differ from `private`?

Exercise 15.3: Define your own versions of the `Quote` class and the `print_total` function.

15.2.2. Defining a Derived Class

A derived class must specify from which class(es) it inherits. It does so in its **class derivation list**, which is a colon followed by a comma-separated list of names of previously defined classes. Each base class name may be preceded by an optional access specifier, which is one of `public`, `protected`, or `private`.

A derived class must declare each inherited member function it intends to override. Therefore, our `Bulk_quote` class must include a `net_price` member:

Click here to view code image

```
class Bulk_quote : public Quote { // Bulk_quote inherits from Quote
    Bulk_quote() = default;
    Bulk_quote(const std::string&, double, std::size_t, double);
    // overrides the base version in order to implement the bulk purchase discount policy
    double net_price(std::size_t) const override;
private:
    std::size_t min_qty = 0; // minimum purchase for the discount to apply
    double discount = 0.0;    // fractional discount to apply
};
```

Our `Bulk_quote` class inherits the `isbn` function and the `bookNo` and `price` data members of its `Quote` base class. It defines its own version of `net_price` and has two additional data members, min_qty and discount. These members specify the minimum quantity and the discount to apply once that number of copies are purchased.

We'll have more to say about the access specifier used in a derivation list in §15.5 (p. 612). For now, what's useful to know is that the access specifier determines whether users of a derived class are allowed to know that the derived class inherits from its base class.

When the derivation is public, the public members of the base class become part of the interface of the derived class as well. In addition, we can bind an object of a publicly derived type to a pointer or reference to the base type. Because we used public in the derivation list, the interface to Bulk_quote implicitly contains the isbn function, and we may use a Bulk_quote object where a pointer or reference to Quote is expected.

Most classes inherit directly from only one base class. This form of inheritance, known as "single inheritance," forms the topic of this chapter. §18.3 (p. 802) will cover classes that have derivation lists with more than one base class.

Virtual Functions in the Derived Class

Derived classes frequently, but not always, override the virtual functions that they inherit. If a derived class does not override a virtual from its base, then, like any other member, the derived class inherits the version defined in its base class.

C++ 11

A derived class may include the virtual keyword on the functions it overrides, but it is not required to do so. For reasons we'll explain in §15.3 (p. 606), the new standard lets a derived class explicitly note that it intends a member function to override a virtual that it inherits. It does so by specifying override after the parameter list, or after the const or reference qualifier(s) if the member is a const (§7.1.2, p. 258) or reference (§13.6.3, p. 546) function.

Derived-Class Objects and the Derived-to-Base Conversion

A derived object contains multiple parts: a subobject containing the (nonstatic) members defined in the derived class itself, plus subobjects corresponding to each base class from which the derived class inherits. Thus, a Bulk_quote object will contain four data elements: the bookNo and price data members that it inherits from Quote, and the min_qty and discount members, which are defined by Bulk_quote.

Although the standard does not specify how derived objects are laid out in memory, we can think of a Bulk_quote object as consisting of two parts as represented in Figure 15.1.

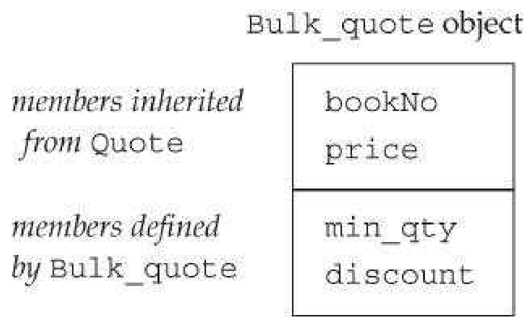


Figure 15.1. Conceptual Structure of a Bulk_quote Object

The base and derived parts of an object are not guaranteed to be stored contiguously. Figure 15.1 is a conceptual, not physical, representation of how classes work.

Because a derived object contains subparts corresponding to its base class(es), we can use an object of a derived type *as if* it were an object of its base type(s). In particular, we can bind a base-class reference or pointer to the base-class part of a derived object.

Click here to view code image

```
Quote item;          //  object of base type
Bulk_quote bulk;     //  object of derived type
Quote *p = &item;    //  p points to a Quote object
p = &bulk;           //  p points to the Quote part of bulk
Quote &r = bulk;     //  r bound to the Quote part of bulk
```

This conversion is often referred to as the **derived-to-base** conversion. As with any other conversion, the compiler will apply the derived-to-base conversion implicitly (§4.11, p. 159).

The fact that the derived-to-base conversion is implicit means that we can use an object of derived type or a reference to a derived type when a reference to the base type is required. Similarly, we can use a pointer to a derived type where a pointer to the base type is required.

Note

The fact that a derived object contains subobjects for its base classes is key to how inheritance works.

Derived-Class Constructors

Although a derived object contains members that it inherits from its base, it cannot directly initialize those members. Like any other code that creates an object of the base-class type, a derived class must use a base-class constructor to initialize its base-class part.

Note

Each class controls how its members are initialized.

The base-class part of an object is initialized, along with the data members of the derived class, during the initialization phase of the constructor (§7.5.1, p. 288). Analogously to how we initialize a member, a derived-class constructor uses its constructor initializer list to pass arguments to a base-class constructor. For example, the `Bulk_quote` constructor with four parameters:

Click here to view code image

```
Bulk_quote(const std::string& book, double p,
           std::size_t qty, double disc) :
           Quote(book, p), min_qty(qty), discount(disc) { }
    // as before
};
```

passes its first two parameters (representing the ISBN and price) to the `Quote` constructor. That `Quote` constructor initializes the `Bulk_quote`'s base-class part (i.e., the `bookNo` and `price` members). When the (empty) `Quote` constructor body completes, the base-class part of the object being constructed will have been initialized. Next the direct members, `min_qty` and `discount`, are initialized. Finally, the (empty) function body of the `Bulk_quote` constructor is run.

As with a data member, unless we say otherwise, the base part of a derived object is default initialized. To use a different base-class constructor, we provide a constructor initializer using the name of the base class, followed (as usual) by a parenthesized list of arguments. Those arguments are used to select which base-class constructor to use to initialize the base-class part of the derived object.

Note

The base class is initialized first, and then the members of the derived class are initialized in the order in which they are declared in the class.

Using Members of the Base Class from the Derived Class

A derived class may access the `public` and `protected` members of its base class:

Click here to view code image

```
// if the specified number of items are purchased, use the discounted price
double Bulk_quote::net_price(size_t cnt) const
{
    if (cnt >= min_qty)
        return cnt * (1 - discount) * price;
    else
        return cnt * price;
}
```

This function generates a discounted price: If the given quantity is more than `min_qty`, we apply the `discount` (which was stored as a fraction) to the `price`.

We'll have more to say about scope in §15.6 (p. 617), but for now it's worth knowing that the scope of a derived class is nested inside the scope of its base class. As a result, there is no distinction between how a member of the derived class uses members defined in its own class (e.g., `min_qty` and `discount`) and how it uses members defined in its base (e.g., `price`).

Key Concept: Respecting the Base-Class Interface

It is essential to understand that each class defines its own interface. Interactions with an object of a class-type should use the interface of that class, even if that object is the base-class part of a derived object.

As a result, derived-class constructors may not directly initialize the members of its base class. The constructor body of a derived constructor can assign values to its `public` or `protected` base-class members. Although it *can* assign to those members, it generally *should not* do so. Like any other user of the base class, a derived class should respect the interface of its base class by using a constructor to initialize its inherited members.

Inheritance and `static` Members

If a base class defines a `static` member (§7.6, p. 300), there is only one such member defined for the entire hierarchy. Regardless of the number of classes derived from a base class, there exists a single instance of each `static` member.

Click here to view code image

```
class Base {
public:
    static void statmem();
};
class Derived : public Base {
    void f(const Derived&);
};
```

`static` members obey normal access control. If the member is `private` in the base class, then derived classes have no access to it. Assuming the member is accessible, we can use a `static` member through either the base or derived:

Click here to view code image

```
void Derived::f(const Derived &derived_obj)
{
    Base::statmem();    // ok: Base defines statmem
    Derived::statmem(); // ok: Derived inherits statmem
    // ok: derived objects can be used to access static from base
    derived_obj.statmem(); // accessed through a Derived object
    statmem();             // accessed through this object
}
```

Declarations of Derived Classes

A derived class is declared like any other class (§7.3.3, p. 278). The declaration contains the class name but does not include its derivation list:

Click here to view code image

```
class Bulk_quote : public Quote; // error: derivation list can't appear here
class Bulk_quote;                // ok: right way to declare a derived class
```

The purpose of a declaration is to make known that a name exists and what kind of entity it denotes, for example, a class, function, or variable. The derivation list, and all other details of the definition, must appear together in the class body.

Classes Used as a Base Class

A class must be defined, not just declared, before we can use it as a base class:

Click here to view code image

```
class Quote;   // declared but not defined
// error: Quote must be defined
class Bulk_quote : public Quote { ... };
```

The reason for this restriction should be easy to see: Each derived class contains, and may use, the members it inherits from its base class. To use those members, the derived class must know what they are. One implication of this rule is that it is impossible to derive a class from itself.

A base class can itself be a derived class:

Click here to view code image

```
class Base { /* … */ } ;
class D1: public Base { /* … */ };
class D2: public D1 { /* … */ };
```

In this hierarchy, `Base` is a **direct base** to `D1` and an **indirect base** to `D2`. A direct base class is named in the derivation list. An indirect base is one that a derived class inherits through its direct base class.

Each class inherits all the members of its direct base class. The most derived class inherits the members of its direct base. The members in the direct base include those it inherits from its base class, and so on up the inheritance chain. Effectively, the most derived object contains a subobject for its direct base and for each of its indirect bases.

Preventing Inheritance

C++ 11

Sometimes we define a class that we don't want others to inherit from. Or we might define a class for which we don't want to think about whether it is appropriate as a base class. Under the new standard, we can prevent a class from being used as a base by following the class name with `final`:

Click here to view code image

```
class NoDerived final { /*   */ }; // NoDerived  can't be a base class
class Base { /*   */ };
// Last  is  final;  we cannot inherit from  Last
class Last final : Base { /*   */ }; // Last  can't be a base class
class Bad : NoDerived { /*   */ };    // error:  NoDerived is final
class Bad2 : Last { /*   */ };        // error:  Last  is  final
```

Exercises Section 15.2.2

Exercise 15.4: Which of the following declarations, if any, are incorrect? Explain why.

`class Base { ... };`

(a) `class Derived : public Derived { ... };`

(b) `class Derived : private Base { ... };`

(c) `class Derived : public Base;`

Exercise 15.5: Define your own version of the `Bulk_quote` class.

Exercise 15.6: Test your `print_total` function from the exercises in § 15.2.1 (p. 595) by passing both `Quote` and `Bulk_quote` objects o that function.

Exercise 15.7: Define a class that implements a limited discount strategy, which applies a discount to books purchased up to a given limit. If the number of copies exceeds that limit, the normal price applies to those purchased beyond the limit.

15.2.3. Conversions and Inheritance

Warning

Understanding conversions between base and derived classes is essential to understanding how object-oriented programming works in C++.

Ordinarily, we can bind a reference or a pointer only to an object that has the same type as the corresponding reference or pointer (§2.3.1, p. 51, and §2.3.2, p. 52) or to a type that involves an acceptable `const` conversion (§4.11.2, p. 162). Classes related by inheritance are an important exception: We can bind a pointer or reference to a base-class type to an object of a type derived from that base class. For example, we can use a `Quote&` to refer to a `Bulk_quote` object, and we can assign the address of a `Bulk_quote` object to a `Quote*`.

The fact that we can bind a reference (or pointer) to a base-class type to a derived object has a crucially important implication: When we use a reference (or pointer) to a base-class type, we don't know the actual type of the object to which the pointer or reference is bound. That object can be an object of the base class or it can be an object of a derived class.

Note

Like built-in pointers, the smart pointer classes (§12.1, p. 450) support the derived-to-base conversion—we can store a pointer to a derived object in a smart pointer to the base type.

Static Type and Dynamic Type

When we use types related by inheritance, we often need to distinguish between the **static type** of a variable or other expression and the **dynamic type** of the object that expression represents. The static type of an expression is always known at compile time—it is the type with which a variable is declared or that an expression yields. The dynamic type is the type of the object in memory that the variable or expression represents. The dynamic type may not be known until run time.

For example, when `print_total` calls `net_price` (§15.1, p. 593):

```
double ret = item.net_price(n);
```

we know that the static type of `item` is `Quote&`. The dynamic type depends on the type of the argument to which `item` is bound. That type cannot be known until a call is executed at run time. If we pass a `Bulk_quote` object to `print_total`, then the static type of `item` will differ from its dynamic type. As we've seen, the static type of `item` is `Quote&`, but in this case the dynamic type is `Bulk_quote`.

The dynamic type of an expression that is neither a reference nor a pointer is always the same as that expression's static type. For example, a variable of type `Quote` is always a `Quote` object; there is nothing we can do that will change the type of the object to which that variable corresponds.

Note

It is crucial to understand that the static type of a pointer or reference to a base class may differ from its dynamic type.

There Is No Implicit Conversion from Base to Derived ...

The conversion from derived to base exists because every derived object contains a base-class part to which a pointer or reference of the base-class type can be bound. There is no similar guarantee for base-class objects. A base-class object can exist either as an independent object or as part of a derived object. A base object that is not part of a derived object has only the members defined by the base class; it doesn't have the members defined by the derived class.

Because a base object might or might not be part of a derived object, there is no automatic conversion from the base class to its derived class(s):

Click here to view code image

```
Quote base;
Bulk_quote* bulkP = &base;   // error: can't convert base to derived
Bulk_quote& bulkRef = base; // error: can't convert base to derived
```

If these assignments were legal, we might attempt to use `bulkP` or `bulkRef` to use members that do not exist in `base`.

What is sometimes a bit surprising is that we cannot convert from base to derived even when a base pointer or reference is bound to a derived object:

Click here to view code image

```
Bulk_quote bulk;
Quote *itemP = &bulk;          // ok: dynamic type is Bulk_quote
Bulk_quote *bulkP = itemP;    // error: can't convert base to derived
```

The compiler has no way to know (at compile time) that a specific conversion will be safe at run time. The compiler looks only at the static types of the pointer or reference to determine whether a conversion is legal. If the base class has one or more virtual functions, we can use a `dynamic_cast` (which we'll cover in §19.2.1 (p. 825)) to request a conversion that is checked at run time. Alternatively, in those cases when we *know* that the conversion from base to derived is safe, we can use a `static_cast` (§4.11.3, p. 162) to override the compiler.

...and No Conversion between Objects

The automatic derived-to-base conversion applies only for conversions to a reference or pointer type. There is no such conversion from a derived-class type to the base-class type. Nevertheless, it is often possible to convert an object of a derived class to its base-class type. However, such conversions may not behave as we might want.

Remember that when we initialize or assign an object of a class type, we are actually calling a function. When we initialize, we're calling a constructor (§13.1.1, p. 496, and §13.6.2, p. 534); when we assign, we're calling an assignment operator (§13.1.2, p. 500, and §13.6.2, p. 536). These members normally have a parameter that is a reference to the `const` version of the class type.

Because these members take references, the derived-to-base conversion lets us pass a derived object to a base-class copy/move operation. These operations are not virtual. When we pass a derived object to a base-class constructor, the constructor that is run is defined in the base class. That constructor knows *only* about the members of the base class itself. Similarly, if we assign a derived object to a base object, the assignment operator that is run is the one defined in the base class. That operator also knows *only* about the members of the base class itself.

For example, our bookstore classes use the synthesized versions of copy and assignment (§13.1.1, p. 497, and §13.1.2, p. 500). We'll have more to say about copy control and inheritance in §15.7.2 (p. 623), but for now what's useful to know is that the synthesized versions memberwise copy or assign the data members of the class the same way as for any other class:

Click here to view code image

```
Bulk_quote bulk;    // object of derived type
Quote item(bulk);   // uses the Quote::Quote(const Quote&) constructor
item = bulk;        // calls Quote::operator=(const Quote&)
```

When `item` is constructed, the `Quote` copy constructor is run. That constructor knows only about the `bookNo` and `price` members. It copies those members from the `Quote` part of `bulk` and *ignores* the members that are part of the `Bulk_quote` portion of `bulk`. Similarly for the assignment of `bulk` to `item`; only the `Quote` part of `bulk` is assigned to `item`.

Because the `Bulk_quote` part is ignored, we say that the `Bulk_quote` portion of `bulk` is **sliced down**.

Warning

When we initialize or assign an object of a base type from an object of a derived type, only the base-class part of the derived object is copied, moved, or assigned. The derived part of the object is ignored.

15.3. Virtual Functions

As we've seen, in C++ dynamic binding happens when a virtual member function is called through a reference or a pointer to a base-class type (§15.1, p. 593). Because we don't know which version of a function is called until run time, virtual functions must *always* be defined. Ordinarily, if we do not use a function, we don't need to supply a definition for that function (§6.1.2, p. 206). However, we must define every virtual function, regardless of whether it is used, because the compiler has no way to determine whether a virtual function is used.

Exercises Section 15.2.3

Exercise 15.8: Define static type and dynamic type.

Exercise 15.9: When is it possible for an expression's static type to differ from its dynamic type? Give three examples in which the static and dynamic type differ.

Exercise 15.10: Recalling the discussion from §8.1 (p. 311), explain how the program on page 317 that passed an `ifstream` to the `Sales_data` `read` function works.

Key Concept: Conversions among Types Related by Inheritance

There are three things that are important to understand about conversions among classes related by inheritance:

• The conversion from derived to base applies only to pointer or reference types.

- There is no implicit conversion from the base-class type to the derived type.
- Like any member, the derived-to-base conversion may be inaccessible due to access controls. We'll cover accessibility in §15.5 (p. 613).

Although the automatic conversion applies only to pointers and references, most classes in an inheritance hierarchy (implicitly or explicitly) define the copy-control members (Chapter 13). As a result, we can often copy, move, or assign an object of derived type to a base-type object. However, copying, moving, or assigning a derived-type object to a base-type object copies, moves, or assigns *only* the members in the base-class part of the object.

Calls to Virtual Functions *May Be* Resolved at Run Time

When a virtual function is called through a reference or pointer, the compiler generates code to *decide at run time* which function to call. The function that is called is the one that corresponds to the dynamic type of the object bound to that pointer or reference.

As an example, consider our `print_total` function from §15.1 (p. 593). That function calls `net_price` on its parameter named `item`, which has type `Quote&`. Because `item` is a reference, and because `net_price` is virtual, the version of `net_price` that is called depends at run time on the actual (dynamic) type of the argument bound to `item`:

Click here to view code image

```
Quote base("0-201-82470-1", 50);
print_total(cout, base, 10);    // calls Quote::net_price
Bulk_quote derived("0-201-82470-1", 50, 5, .19);
print_total(cout, derived, 10); // calls Bulk_quote::net_price
```

In the first call, `item` is bound to an object of type `Quote`. As a result, when `print_total` calls `net_price`, the version defined by `Quote` is run. In the second call, `item` is bound to a `Bulk_quote` object. In this call, `print_total` calls the `Bulk_quote` version of `net_price`.

It is crucial to understand that dynamic binding happens only when a virtual function is called through a pointer or a reference.

Click here to view code image

```
base = derived;         // copies the Quote part of derived into base
base.net_price(20);     // calls Quote::net_price
```

When we call a virtual function on an expression that has a plain—nonreference and nonpointer—type, that call is bound at compile time. For example, when we call `net_price` on `base`, there is no question as to which version of `net_price` to run. We can change the value (i.e., the contents) of the object that `base` represents, but there is no way to change the type of that object. Hence, this call is resolved, at compile time, to the `Quote` version of `net_price`.

Key Concept: Polymorphism in C++

The key idea behind OOP is polymorphism. Polymorphism is derived from a Greek word meaning "many forms." We speak of types related by inheritance as polymorphic types, because we can use the "many forms" of these types while ignoring the differences among them. The fact that the static and dynamic types of references and pointers can differ is the cornerstone of how C++ supports polymorphism.

When we call a function defined in a base class through a reference or pointer to the base class, we do not know the type of the object on which that member is executed. The object can be a base-class object or an object of a derived class. If the function is virtual, then the decision as to which function to run is delayed until run time. The version of the virtual function that is run is the one defined by the type of the object to which the reference is bound or to which the pointer points.

On the other hand, calls to nonvirtual functions are bound at compile time. Similarly, calls to any function (virtual or not) on an object are also bound at compile time. The type of an object is fixed and unvarying—there is nothing we can do to make the dynamic type of an object differ from its static type. Therefore, calls made on an object are bound at compile time to the version defined by the type of the object.

Note

Virtuals are resolved at run time *only* if the call is made through a reference or pointer. Only in these cases is it possible for an object's dynamic type to differ from its static type.

Virtual Functions in a Derived Class

When a derived class overrides a virtual function, it may, but is not required to, repeat the `virtual` keyword. Once a function is declared as `virtual`, it remains `virtual` in all the derived classes.

A derived-class function that overrides an inherited virtual function must have exactly the same parameter type(s) as the base-class function that it overrides.

With one exception, the return type of a virtual in the derived class also must match the return type of the function from the base class. The exception applies to virtuals that return a reference (or pointer) to types that are themselves related by inheritance. That is, if `D` is derived from `B`, then a base class virtual can return a `B*` and the version in the derived can return a `D*`. However, such return types require that the derived-to-base conversion from `D` to `B` is accessible. §15.5 (p. 613) covers how to determine whether a base class is accessible. We'll see an example of this kind of virtual function in §15.8.1 (p. 633).

Note

A function that is `virtual` in a base class is implicitly `virtual` in its derived classes. When a derived class overrides a virtual, the parameters in the base and derived classes must match exactly.

The final and override Specifiers

As we'll see in §15.6 (p. 620), it is legal for a derived class to define a function with the same name as a virtual in its base class but with a different parameter list. The compiler considers such a function to be independent from the base-class function. In such cases, the derived version does not override the version in the base class. In practice, such declarations often are a mistake—the class author intended to override a virtual from the base class but made a mistake in specifying the parameter list.

Finding such bugs can be surprisingly hard. Under the new standard we can specify `override` on a virtual function in a derived class. Doing so makes our intention clear and (more importantly) enlists the compiler in finding such problems for us. The compiler will reject a program if a function marked `override` does not override an existing virtual function:

Click here to view code image

```
struct B {
    virtual void f1(int) const;
    virtual void f2();
    void f3();
};
struct D1 : B {
    void f1(int) const override; // ok: f1 matches f1 in the base
    void f2(int) override; // error: B has no f2(int) function
    void f3() override;    // error: f3 not virtual
    void f4() override;    // error: B doesn't have a function named f4
};
```

In `D1`, the `override` specifier on `f1` is fine; both the base and derived versions of `f1` are `const` members that take an `int` and return `void`. The version of `f1` in `D1` properly overrides the virtual that it inherits from `B`.

The declaration of `f2` in `D1` does not match the declaration of `f2` in `B`—the version defined in `B` takes no arguments and the one defined in `D1` takes an `int`. Because the declarations don't match, `f2` in `D1` doesn't override `f2` from `B`; it is a new function that happens to have the same name. Because we said we intended this declaration to be an `override` and it isn't, the compiler will generate an error.

Because only a virtual function can be overridden, the compiler will also reject `f3` in `D1`. That function is not virtual in `B`, so there is no function to override. Similarly `f4` is in error because `B` doesn't even have a function named `f4`.

We can also designate a function as `final`. Any attempt to override a function that has been defined as `final` will be flagged as an error:

Click here to view code image

```
struct D2 : B {
    // inherits f2() and f3() from B and overrides f1(int)
     void f1(int) const final; // subsequent classes can't override f1
(int)
};
struct D3 : D2 {
    void f2();          // ok: overrides f2 inherited from the indirect base,
B
    void f1(int) const; // error: D2 declared f2 as final
};
```

`final` and `override` specifiers appear after the parameter list (including any `const` or reference qualifiers) and after a trailing return (§ 6.3.3, p. 229).

Virtual Functions and Default Arguments

Like any other function, a virtual function can have default arguments (§ 6.5.1, p. 236). If a call uses a default argument, the value that is used is the one defined by the static type through which the function is called.

That is, when a call is made through a reference or pointer to base, the default argument(s) will be those defined in the base class. The base-class arguments will be used even when the derived version of the function is run. In this case, the derived function will be passed the default arguments defined for the base-class version of the function. If the derived function relies on being passed different arguments, the program will not execute as expected.

Best Practices

Virtual functions that have default arguments should use the same argument values in the base and derived classes.

Circumventing the Virtual Mechanism

In some cases, we want to prevent dynamic binding of a call to a virtual function; we want to force the call to use a particular version of that virtual. We can use the scope operator to do so. For example, this code:

Click here to view code image

```
//  calls the version from the base class regardless of the dynamic type of baseP
double undiscounted = baseP->Quote::net_price(42);
```

calls the `Quote` version of `net_price` regardless of the type of the object to which `baseP` actually points. This call will be resolved at compile time.

Note

Ordinarily, only code inside member functions (or friends) should need to use the scope operator to circumvent the virtual mechanism.

Why might we wish to circumvent the virtual mechanism? The most common reason is when a derived-class virtual function calls the version from the base class. In such cases, the base-class version might do work common to all types in the hierarchy. The versions defined in the derived classes would do whatever additional work is particular to their own type.

Warning

If a derived virtual function that intended to call its base-class version omits the scope operator, the call will be resolved at run time as a call to the derived version itself, resulting in an infinite recursion.

Exercises Section 15.3

Exercise 15.11: Add a virtual `debug` function to your `Quote` class hierarchy that displays the data members of the respective classes.

Exercise 15.12: Is it ever useful to declare a member function as both `override` and `final`? Why or why not?

Exercise 15.13: Given the following classes, explain each `print` function:

Click here to view code image

```
class base {
public:
   string name() { return basename; }
   virtual void print(ostream &os) { os << basename; }
private:
   string basename;
};
class derived : public base {
public:
   void print(ostream &os) { print(os); os << " " << i; }
private:
   int i;
};
```

If there is a problem in this code, how would you fix it?

Exercise 15.14: Given the classes from the previous exercise and the following objects, determine which function is called at run time:

Click here to view code image

```
base bobj;     base *bp1 = &bobj;   base &br1 = bobj;
derived dobj;  base *bp2 = &dobj;   base &br2 = dobj;
```

(a) `bobj.print();`

(b) `dobj.print();`

(c) `bp1->name();`

(d) `bp2->name();`

(e) `br1.print();`

(f) `br2.print();`

15.4. Abstract Base Classes

Imagine that we want to extend our bookstore classes to support several discount strategies. In addition to a bulk discount, we might offer a discount for purchases up to a certain quantity and then charge the full price thereafter. Or we might offer a discount for purchases above a certain limit but not for purchases up to that limit.

Each of these discount strategies is the same in that it requires a quantity and a discount amount. We might support these differing strategies by defining a new class named `Disc_quote` to store the quantity and the discount amount. Classes, such as `Bulk_item`, that represent a specific discount strategy will inherit from `Disc_quote`. Each of the derived classes will implement its discount strategy by defining its own version of `net_price`.

Before we can define our `Disc_Quote` class, we have to decide what to do about `net_price`. Our `Disc_quote` class doesn't correspond to any particular discount strategy; there is no meaning to ascribe to `net_price` for this class.

We could define `Disc_quote` without its own version of `net_price`. In this case, `Disc_quote` would inherit `net_price` from `Quote`.

However, this design would make it possible for our users to write nonsensical code. A user could create an object of type `Disc_quote` by supplying a quantity and a discount rate. Passing that `Disc_quote` object to a function such as `print_total` would use the `Quote` version of `net_price`. The calculated price would not include the discount that was supplied when the object was created. That state of affairs makes no sense.

Pure Virtual Functions

Thinking about the question in this detail reveals that our problem is not just that we don't know how to define `net_price`. In practice, we'd like to prevent users from creating `Disc_quote` objects at all. This class represents the general concept of a discounted book, not a concrete discount strategy.

We can enforce this design intent—and make it clear that there is no meaning for `net_price`—by defining `net_price` as a **pure virtual** function. Unlike ordinary virtuals, a pure virtual function does not have to be defined. We specify that a virtual function is a pure virtual by writing `= 0` in place of a function body (i.e., just before the semicolon that ends the declaration). The `= 0` may appear only on the declaration of a virtual function in the class body:

Click here to view code image

```
// class to hold the discount rate and quantity
// derived classes will implement pricing strategies using these data
class Disc_quote : public Quote {
public:
    Disc_quote() = default;
    Disc_quote(const std::string& book, double price,
              std::size_t qty, double disc):
                 Quote(book, price),
                 quantity(qty), discount(disc) { }
    double net_price(std::size_t) const = 0;
protected:
    std::size_t quantity = 0; // purchase size for the discount to apply
    double discount = 0.0;    // fractional discount to apply
};
```

Like our earlier `Bulk_item` class, `Disc_quote` defines a default constructor and a constructor that takes four parameters. Although we cannot define objects of this type directly, constructors in classes derived from `Disc_quote` will use the `Disc_quote` constructors to construct the `Disc_quote` part of their objects. The constructor that has four parameters passes its first two to the `Quote` constructor and directly initializes its own members, `discount` and `quantity`. The default constructor default initializes those members.

It is worth noting that we can provide a definition for a pure virtual. However, the function body must be defined outside the class. That is, we cannot provide a function body inside the class for a function that is `= 0`.

Classes with Pure Virtuals Are Abstract Base Classes

A class containing (or inheriting without overridding) a pure virtual function is an **abstract base class**. An abstract base class defines an interface for subsequent classes to override. We cannot (directly) create objects of a type that is an abstract base class. Because `Disc_quote` defines `net_price` as a pure virtual, we cannot define objects of type `Disc_quote`. We can define objects of classes that inherit from `Disc_quote`, so long as those classes override `net_price`:

Click here to view code image

```
// Disc_quote declares pure virtual functions, which Bulk_quote will override
Disc_quote discounted; // error: can't define a Disc_quote object
Bulk_quote bulk;       // ok: Bulk_quote has no pure virtual functions
```

Classes that inherit from `Disc_quote` must define `net_price` or those classes will be abstract as well.

Note

We may not create objects of a type that is an abstract base class.

A Derived Class Constructor Initializes Its Direct Base Class Only

Now we can reimplement `Bulk_quote` to inherit from `Disc_quote` rather than inheriting directly from `Quote`:

Click here to view code image

```
// the discount kicks in when a specified number of copies of the same book are sold
// the discount is expressed as a fraction to use to reduce the normal price
class Bulk_quote : public Disc_quote {
public:
    Bulk_quote() = default;
    Bulk_quote(const std::string& book, double price,
              std::size_t qty, double disc):
          Disc_quote(book, price, qty, disc) { }
```

```
        // overrides the base version to implement the bulk purchase discount policy
        double net_price(std::size_t) const override;
    };
```

This version of `Bulk_quote` has a direct base class, `Disc_quote`, and an indirect base class, `Quote`. Each `Bulk_quote` object has three subobjects: an (empty) `Bulk_quote` part, a `Disc_quote` subobject, and a `Quote` subobject.

As we've seen, each class controls the initialization of objects of its type. Therefore, even though `Bulk_quote` has no data members of its own, it provides the same four-argument constructor as in our original class. Our new constructor passes its arguments to the `Disc_quote` constructor. That constructor in turn runs the `Quote` constructor. The `Quote` constructor initializes the `bookNo` and `price` members of `bulk`. When the `Quote` constructor ends, the `Disc_quote` constructor runs and initializes the `quantity` and `discount` members. At this point, the `Bulk_quote` constructor resumes. That constructor has no further initializations or any other work to do.

Key Concept: Refactoring

Adding `Disc_quote` to the `Quote` hierarchy is an example of *refactoring.* Refactoring involves redesigning a class hierarchy to move operations and/or data from one class to another. Refactoring is common in object-oriented applications.

It is noteworthy that even though we changed the inheritance hierarchy, code that uses `Bulk_quote` or `Quote` would not need to change. However, when classes are refactored (or changed in any other way) we must recompile any code that uses those classes.

Exercises Section 15.4

Exercise 15.15: Define your own versions of `Disc_quote` and `Bulk_quote`.

Exercise 15.16: Rewrite the class representing a limited discount strategy, which you wrote for the exercises in § 15.2.2 (p. 601), to inherit from `Disc_quote`.

Exercise 15.17: Try to define an object of type `Disc_quote` and see what errors you get from the compiler.

15.5. Access Control and Inheritance

Just as each class controls the initialization of its own members (§ 15.2.2, p. 598), each class also controls whether its members are **accessible** to a derived class.

protected Members

As we've seen, a class uses `protected` for those members that it is willing to share with its derived classes but wants to protect from general access. The `protected` specifier can be thought of as a blend of `private` and `public`:

- Like `private`, `protected` members are inaccessible to users of the class.
- Like `public`, `protected` members are accessible to members and friends of classes derived from this class.

In addition, `protected` has another important property:

- A derived class member or friend may access the `protected` members of the base class *only* through a derived object. The derived class has no special access to the `protected` members of base-class objects.

To understand this last rule, consider the following example:

Click here to view code image

```
class Base {
protected:
    int prot_mem;      // protected member
};
class Sneaky : public Base  {
    friend void clobber(Sneaky&);  // can access Sneaky::prot_mem
    friend void clobber(Base&);    // can't access Base::prot_mem
    int j;                          // j is private by default
};
// ok: clobber can access the private and protected members in Sneaky objects
void clobber(Sneaky &s) { s.j = s.prot_mem = 0; }
// error: clobber can't access the protected members in Base
void clobber(Base &b) { b.prot_mem = 0; }
```

If derived classes (and friends) could access protected members in a base-class object, then our second version of `clobber` (that takes a `Base&`) would be legal. That function is not a friend of `Base`, yet it would be allowed to change an object of type `Base`; we could circumvent the protection provided by `protected` for any class simply by defining a new class along the lines of `Sneaky`.

To prevent such usage, members and friends of a derived class can access the `protected` members *only* in base-class objects that are embedded inside a derived type object; they have no special access to ordinary objects of the base type.

public, private, and protected Inheritance

Access to a member that a class inherits is controlled by a combination of the access specifier for that member in the base class, and the access specifier in the derivation list of the derived class. As an example, consider the following hierarchy:

Click here to view code image

```
class Base {
public:
    void pub_mem();    // public member
protected:
    int prot_mem;      // protected member
private:
    char priv_mem;     // private member
};
struct Pub_Derv : public Base {
    // ok: derived classes can access protected members
    int f() { return prot_mem; }
    // error: private members are inaccessible to derived classes
    char g() { return priv_mem; }
};
struct Priv_Derv : private Base {
    // private derivation doesn't affect access in the derived class
    int f1() const { return prot_mem; }
};
```

The derivation access specifier has no effect on whether members (and friends) of a derived class may access the members of its own direct base class. Access to the members of a base class is controlled by the access specifiers in the base class itself. Both `Pub_Derv` and `Priv_Derv` may access the `protected` member `prot_mem`. Neither may access the `private` member `priv_mem`.

The purpose of the derivation access specifier is to control the access that *users* of the derived class—including other classes derived from the derived class—have to the members inherited from `Base`:

Click here to view code image

```
Pub_Derv d1;    //  members inherited from Base are public
Priv_Derv d2;   //  members inherited from Base are private
d1.pub_mem();   //  ok: pub_mem is public in the derived class
d2.pub_mem();   //  error: pub_mem is private in the derived class
```

Both `Pub_Derv` and `Priv_Derv` inherit the `pub_mem` function. When the inheritance is `public`, members retain their access specification. Thus, `d1` can call `pub_mem`. In `Priv_Derv`, the members of `Base` are `private`; users of that class may not call `pub_mem`.

The derivation access specifier used by a derived class also controls access from classes that inherit from that derived class:

Click here to view code image

```
struct Derived_from_Public : public Pub_Derv {
    // ok: Base::prot_mem remains protected in Pub_Derv
    int use_base() { return prot_mem; }
};
struct Derived_from_Private : public Priv_Derv {
    // error: Base::prot_mem is private in Priv_Derv
    int use_base() { return prot_mem; }
};
```

Classes derived from `Pub_Derv` may access `prot_mem` from `Base` because that member remains a `protected` member in `Pub_Derv`. In contrast, classes derived from `Priv_Derv` have no such access. To them, all the members that `Priv_Derv` inherited from `Base` are `private`.

Had we defined another class, say, `Prot_Derv`, that used `protected` inheritance, the `public` members of `Base` would be `protected` members in that class. Users of `Prot_Derv` would have no access to `pub_mem`, but the members and friends of `Prot_Derv` could access that inherited member.

Accessibility of Derived-to-Base Conversion

Whether the derived-to-base conversion (§ 15.2.2, p. 597) is accessible depends on which code is trying to use the conversion and may depend on the access specifier used in the derived class' derivation. Assuming `D` inherits from `B`:

- User code may use the derived-to-base conversion *only* if `D` inherits publicly from `B`. User code may not use the conversion if `D` inherits from `B` using either `protected` or `private`.
- Member functions and friends of `D` can use the conversion to `B` regardless of how `D` inherits from `B`. The derived-to-base conversion to a direct base class is always accessible to members and friends of a derived class.
- Member functions and friends of classes derived from `D` may use the derived-to-base conversion if `D` inherits from `B` using either `public` or `protected`. Such code may not use the conversion if `D` inherits privately from `B`.

Tip

For any given point in your code, if a `public` member of the base class would be accessible, then the derived-to-base conversion is also accessible, and not otherwise.

Key Concept: Class Design and `protected` Members

In the absence of inheritance, we can think of a class as having two different kinds of users: ordinary users and implementors. Ordinary users write code that uses objects of the class type; such code can access only the `public` (interface) members of the class. Implementors write the code contained in the members and friends of the class. The members and friends of the class can access both the `public` and `private` (implementation) sections.

Under inheritance, there is a third kind of user, namely, derived classes. A base class makes `protected` those parts of its implementation that it is willing to let its derived classes use. The `protected` members remain inaccessible to ordinary user code; `private` members remain inaccessible to derived classes and their friends.

Like any other class, a class that is used as a base class makes its interface members `public` . A class that is used as a base class may divide its implementation into those members that are accessible to derived classes and those that remain accessible only to the base class and its friends. An implementation member should be `protected` if it provides an operation or data that a derived class will need to use in its own implementation. Otherwise, implementation members should be `private`.

Friendship and Inheritance

Just as friendship is not transitive (§7.3.4, p. 279), friendship is also not inherited. Friends of the base have no special access to members of its derived classes, and friends of a derived class have no special access to the base class:

Click here to view code image

```
class Base {
    // added friend declaration; other members as before
    friend class Pal; // Pal has no access to classes derived from Base
};
class Pal {
public:
    int f(Base b) { return b.prot_mem; } // ok: Pal is a friend of Base
    int f2(Sneaky s) { return s.j; } // error: Pal not friend of Sneaky
    // access to a base class is controlled by the base class, even inside a derived object
    int f3(Sneaky s) { return s.prot_mem; } // ok: Pal is a friend
};
```

The fact that `f3` is legal may seem surprising, but it follows directly from the notion that each class controls access to its own members. `Pal` is a friend of `Base`, so `Pal` can access the members of `Base` objects. That access includes access to `Base` objects that are embedded in an object of a type derived from `Base`.

When a class makes another class a friend, it is only that class to which friendship is granted. The base classes of, and classes derived from, the friend have no special access to the befriending class:

Click here to view code image

```
// D2 has no access to protected or private members in Base
class D2 : public Pal {
public:
   int mem(Base b)
       { return b.prot_mem; } // error: friendship doesn't inherit
};
```

Note

Friendship is not inherited; each class controls access to its members.

Exempting Individual Members

Sometimes we need to change the access level of a name that a derived class inherits. We can do so by providing a `using` declaration (§3.1, p. 82):

Click here to view code image

```
class Base {
public:
    std::size_t size() const { return n; }
protected:
    std::size_t n;
};
class Derived : private Base {    //  note: private inheritance
public:
    // maintain access levels for members related to the size of the object
    using Base::size;
protected:
    using Base::n;
};
```

Because `Derived` uses `private` inheritance, the inherited members, `size` and `n`, are (by default) `private` members of `Derived`. The `using` declarations adjust the accessibility of these members. Users of `Derived` can access the `size` member, and classes subsequently derived from `Derived` can access `n`.

A `using` declaration inside a class can name any accessible (e.g., not `private`) member of a direct or indirect base class. Access to a name specified in a `using` declaration depends on the access specifier preceding the `using` declaration. That is, if a `using` declaration appears in a `private` part of the class, that name is accessible to members and friends only. If the declaration is in a `public` section, the name is available to all users of the class. If the declaration is in a `protected` section, the name is accessible to the members, friends, and derived classes.

Note

A derived class may provide a `using` declaration only for names it is permitted to access.

Default Inheritance Protection Levels

In §7.2 (p. 268) we saw that classes defined with the `struct` and `class` keywords have different default access specifiers. Similarly, the default derivation specifier depends on which keyword is used to define a derived class. By default, a derived class defined with the `class` keyword has `private` inheritance; a derived class defined with `struct` has `public` inheritance:

Click here to view code image

```
class Base { /* ...   */ };
struct D1 : Base { /* ...   */ };    // public  inheritance by default
class D2 : Base { /* ...   */ };     // private  inheritance by default
```

It is a common misconception to think that there are deeper differences between classes defined using the `struct` keyword and those defined using `class`. The only differences are the default access specifier for members and the default derivation access specifier. There are no other distinctions.

Best Practices

A privately derived class should specify `private` explicitly rather than rely on the default. Being explicit makes it clear that private inheritance is intended and not an oversight.

Exercises Section 15.5

Exercise 15.18: Given the classes from page 612 and page 613, and assuming each object has the type specified in the comments, determine which of these assignments are legal. Explain why those that are illegal aren’t allowed:

Click here to view code image

```
Base *p = &d1;    //  d1 has type Pub_Derv
p = &d2;          //  d2 has type Priv_Derv
p = &d3;          //  d3 has type Prot_Derv
p = &dd1;         //  dd1 has type Derived_from_Public
p = &dd2;         //  dd2 has type Derived_from_Private
p = &dd3;         //  dd3 has type Derived_from_Protected
```

Exercise 15.19: Assume that each of the classes from page 612 and page 613 has a member function of the form:

```
void memfcn(Base &b) { b = *this; }
```

For each class, determine whether this function would be legal.

Exercise 15.20: Write code to test your answers to the previous two exercises.

Exercise 15.21: Choose one of the following general abstractions containing a family of types (or choose one of your own). Organize the types into an inheritance hierarchy:

(a) Graphical file formats (such as gif, tiff, jpeg, bmp)

(b) Geometric primitives (such as box, circle, sphere, cone)

(c) C++ language types (such as class, function, member function)

Exercise 15.22: For the class you chose in the previous exercise, identify some of the likely virtual functions as well as `public` and `protected` members.

15.6. Class Scope under Inheritance

Each class defines its own scope (§7.4, p. 282) within which its members are defined. Under inheritance, the scope of a derived class is nested (§2.2.4, p. 48) inside the scope of its base classes. If a name is unresolved within the scope of the derived class, the enclosing base-class scopes are searched for a definition of that name.

The fact that the scope of a derived class nests inside the scope of its base classes can be surprising. After all, the base and derived classes are defined in separate parts of our program’s text. However, it is this hierarchical nesting of class scopes that allows the members of a derived class to use members of its base class as if those members were part of the derived class. For example, when we write

```
Bulk_quote bulk;
```

```
cout << bulk.isbn();
```

the use of the name isbn is resolved as follows:

• Because we called isbn on an object of type Bulk_quote, the search starts in the Bulk_quote class. The name isbn is not found in that class.

• Because Bulk_quote is derived from Disc_quote, the Disc_quote class is searched next. The name is still not found.

• Because Disc_quote is derived from Quote, the Quote class is searched next. The name isbn is found in that class; the use of isbn is resolved to the isbn in Quote.

Name Lookup Happens at Compile Time

The static type (§15.2.3, p. 601) of an object, reference, or pointer determines which members of that object are visible. Even when the static and dynamic types might differ (as can happen when a reference or pointer to a base class is used), the static type determines what members can be used. As an example, we might add a member to the Disc_quote class that returns a pair (§11.2.3, p. 426) holding the minimum (or maximum) quantity and the discounted price:

Click here to view code image

```
class Disc_quote : public Quote {
public:
    std::pair<size_t, double> discount_policy() const
        { return {quantity, discount}; }
    // other members as before
};
```

We can use discount_policy only through an object, pointer, or reference of type Disc_quote or of a class derived from Disc_quote:

Click here to view code image

```
Bulk_quote bulk;
Bulk_quote *bulkP = &bulk; //  static and dynamic types are the same
Quote *itemP = &bulk;      //  static and dynamic types differ
bulkP->discount_policy();  //  ok: bulkP has type Bulk_quote*
itemP->discount_policy();  //  error: itemP has type Quote*
```

Even though bulk has a member named discount_policy, that member is not visible through itemP. The type of itemP is a pointer to Quote, which means that the search for discount_policy starts in class Quote. The Quote class has no member named discount_policy, so we cannot call that member on an object, reference, or pointer of type Quote.

Name Collisions and Inheritance

Like any other scope, a derived class can reuse a name defined in one of its direct or indirect base classes. As usual, names defined in an inner scope (e.g., a derived class) hide uses of that name in the outer scope (e.g., a base class) (§2.2.4, p. 48):

Click here to view code image

```
struct Base {
    Base(): mem(0) { }
protected:
    int mem;
};
struct Derived : Base {
    Derived(int i): mem(i) { } // initializes Derived::mem to i
                               // Base::mem is default initialized
    int get_mem() { return mem; }  // returns Derived::mem
protected:
    int mem;    // hides mem in the base
};
```

The reference to `mem` inside `get_mem` is resolved to the name inside `Derived`. Were we to write

Click here to view code image

```
Derived d(42);
cout << d.get_mem() << endl;       // prints 42
```

then the output would be `42`.

Note

A derived-class member with the same name as a member of the base class hides direct use of the base-class member.

Using the Scope Operator to Use Hidden Members

We can use a hidden base-class member by using the scope operator:

Click here to view code image

```
struct Derived : Base {
    int get_base_mem() { return Base::mem; }
    // ...
};
```

The scope operator overrides the normal lookup and directs the compiler to look for `mem` starting in the scope of class `Base`. If we ran the code above with this version of `Derived`, the result of `d.get_mem()` would be `0`.

Best Practices

Aside from overriding inherited virtual functions, a derived class usually should not reuse names defined in its base class.

Key Concept: Name Lookup and Inheritance

Understanding how function calls are resolved is crucial to understanding inheritance in C++. Given the call `p->mem()` (or `obj.mem()`), the following four steps happen:

- First determine the static type of `p` (or `obj`). Because we're calling a member, that type must be a class type.
- Look for `mem` in the class that corresponds to the static type of `p` (or `obj`). If `mem` is not found, look in the direct base class and continue up the chain of classes until `mem` is found or the last class is searched. If `mem` is not found in the class or its enclosing base classes, then the call will not compile.
- Once `mem` is found, do normal type checking (§6.1, p. 203) to see if this call is legal given the definition that was found.
- Assuming the call is legal, the compiler generates code, which varies depending on whether the call is virtual or not:

  – If `mem` is virtual and the call is made through a reference or pointer, then the compiler generates code to determine at run time which version to run based on the dynamic type of the object.

  – Otherwise, if the function is nonvirtual, or if the call is on an object (not a reference or pointer), the compiler generates a normal function call.

As Usual, Name Lookup Happens before Type Checking

As we've seen, functions declared in an inner scope do not overload functions declared in an outer scope (§6.4.1, p. 234). As a result, functions defined in a derived class do *not* overload members defined in its base class(es). As in any other scope, if a member in a derived class (i.e., in an inner scope) has the same name as a base-class member (i.e., a name defined in an outer scope), then the derived member hides the base-class member within the scope of the derived class. The base member is hidden even if the functions have different parameter lists:

Click here to view code image

```
struct Base {
    int memfcn();
};
struct Derived : Base {
    int memfcn(int);    // hides memfcn in the base
};
Derived d; Base b;
b.memfcn();         //  calls Base::memfcn
d.memfcn(10);       //  calls Derived::memfcn
d.memfcn();         //  error: memfcn with no arguments is hidden
d.Base::memfcn(); //  ok: calls Base::memfcn
```

The declaration of `memfcn` in `Derived` hides the declaration of `memfcn` in `Base`. Not surprisingly, the first call through `b`, which is a `Base` object, calls the version in the base class. Similarly, the second call (through `d`) calls the one from `Derived`. What can be surprising is that the third call, `d.memfcn()`, is illegal.

To resolve this call, the compiler looks for the name `memfcn` in `Derived`. That class defines a member named `memfcn` and the search stops. Once the name is found, the compiler looks no further. The version of `memfcn` in `Derived` expects an `int` argument. This call provides no such argument; it is in error.

Virtual Functions and Scope

We can now understand why virtual functions must have the same parameter list in the base and derived classes (§15.3, p. 605). If the base and derived members took arguments that differed from one another, there would be no way to call the derived version through a reference or pointer to the base class. For example:

Click here to view code image

```
class Base {
public:
    virtual int fcn();
};
class D1 : public Base {
public:
    // hides fcn in the base; this fcn is not virtual
    // D1 inherits the definition of Base::fcn()
    int fcn(int);        // parameter list differs from fcn in Base
    virtual void f2(); // new virtual function that does not exist in Base
};
class D2 : public D1 {
public:
    int fcn(int); // nonvirtual function hides D1::fcn(int)
    int fcn();      // overrides virtual fcn from Base
    void f2();      // overrides virtual f2 from D1
    };
```

The `fcn` function in `D1` does not override the virtual `fcn` from `Base` because they have different parameter lists. Instead, it *hides* `fcn` from the base. Effectively, `D1` has two functions named `fcn`: `D1` inherits a virtual named `fcn` from `Base` and defines its own, nonvirtual member named `fcn` that takes an `int` parameter.

Calling a Hidden Virtual through the Base Class

Given the classes above, let's look at several different ways to call these functions:

Click here to view code image

```
    Base bobj;  D1 d1obj; D2 d2obj;
    Base *bp1 = &bobj, *bp2 = &d1obj, *bp3 = &d2obj;
    bp1->fcn(); // virtual call, will call Base::fcn at run time
    bp2->fcn(); // virtual call, will call Base::fcn at run time
    bp3->fcn(); // virtual call, will call D2::fcn at run time
    D1 *d1p = &d1obj; D2 *d2p = &d2obj;
    bp2->f2(); // error: Base has no member named f2
    d1p->f2(); // virtual call, will call D1::f2() at run time
    d2p->f2(); // virtual call, will call D2::f2() at run time
```

The first three calls are all made through pointers to the base class. Because `fcn` is virtual, the compiler generates code to decide at run time which version to call. That decision will be based on the actual type of the object to which the pointer is bound. In the case of `bp2`, the underlying object is a `D1`. That class did not override the `fcn` function that takes no arguments. Thus, the call through `bp2` is resolved (at run time) to the version defined in `Base`.

The next three calls are made through pointers with differing types. Each pointer points to one of the types in this hierarchy. The first call is illegal because there is no `f2()` in class `Base`. The fact that the pointer happens to point to a derived object is irrelevant.

For completeness, let's look at calls to the nonvirtual function `fcn(int)`:

Click here to view code image

```
    Base *p1 = &d2obj; D1 *p2 = &d2obj; D2 *p3 =  &d2obj;
    p1->fcn(42);   // error: Base has no version of fcn that takes an int
    p2->fcn(42);   // statically bound, calls D1::fcn(int)
    p3->fcn(42);   // statically bound, calls D2::fcn(int)
```

In each call the pointer happens to point to an object of type `D2`. However, the dynamic type doesn't matter when we call a nonvirtual function. The version that is called depends only on the static type of the pointer.

Overriding Overloaded Functions

As with any other function, a member function (virtual or otherwise) can be overloaded. A derived class can override zero or more instances of the overloaded functions it inherits. If a derived class wants to make all the overloaded versions available through its type, then it must override all of them or none of them.

Sometimes a class needs to override some, but not all, of the functions in an overloaded set. It would be tedious in such cases to have to override every base-class version in order to override the ones that the class needs to specialize.

Instead of overriding every base-class version that it inherits, a derived class can provide a `using` declaration (§15.5, p. 615) for the overloaded member. A `using` declaration specifies only a name; it may not specify a parameter list. Thus, a `using` declaration for a base-class member function adds all the overloaded instances of that function to the scope of the derived class. Having brought all the names into its scope, the derived class needs to define only those functions that truly depend on its type. It can use the inherited definitions for the others.

The normal rules for a `using` declaration inside a class apply to names of overloaded functions (§15.5, p. 615); every overloaded instance of the function in the base class must be accessible to the derived class. The access to the overloaded versions that are not otherwise redefined by the derived class will be the access in effect at the point of the `using` declaration.

Exercises Section 15.6

Exercise 15.23: Assuming class `D1` on page 620 had intended to override its inherited `fcn` function, how would you fix that class? Assuming you fixed the class so that `fcn` matched the definition in `Base`, how would the calls in that section be resolved?

15.7. Constructors and Copy Control

Like any other class, a class in an inheritance hierarchy controls what happens when objects of its type are created, copied, moved, assigned, or destroyed. As for any other class, if a class (base or derived) does not itself define one of the copy-control operations, the compiler will synthesize that operation. Also, as usual, the synthesized version of any of these members might be a deleted function.

15.7.1. Virtual Destructors

The primary direct impact that inheritance has on copy control for a base class is that a base class generally should define a virtual destructor (§15.2.1, p. 594). The destructor needs to be virtual to allow objects in the inheritance hierarchy to be dynamically allocated.

Recall that the destructor is run when we `delete` a pointer to a dynamically allocated object (§13.1.3, p. 502). If that pointer points to a type in an inheritance hierarchy, it is possible that the static type of the pointer might differ from the dynamic type of the object being destroyed (§15.2.2, p. 597). For example, if we `delete` a pointer of type `Quote*`, that pointer might point at a `Bulk_quote` object. If the pointer points at a `Bulk_quote`, the compiler has to know that it should run the `Bulk_quote` destructor. As with any other function, we arrange to run the proper destructor by defining the destructor as virtual in the base class:

Click here to view code image

```
class Quote {
public:
    // virtual destructor needed if a base pointer pointing to a derived object is deleted
    virtual ~Quote() = default; // dynamic binding for the destructor
};
```

Like any other virtual, the virtual nature of the destructor is inherited. Thus, classes derived from `Quote` have virtual destructors, whether they use the synthesized destructor or define their own version. So long as the base class destructor is virtual, when we `delete` a pointer to base, the correct destructor will be run:

Click here to view code image

```
Quote *itemP = new Quote;    //  same static and dynamic type
delete itemP;                //  destructor for Quote called
itemP = new Bulk_quote;      //  static and dynamic types differ
delete itemP;                //  destructor for Bulk_quote called
```

Warning

Executing `delete` on a pointer to base that points to a derived object has undefined behavior if the base's destructor is not virtual.

Destructors for base classes are an important exception to the rule of thumb that if a class needs a destructor, it also needs copy and assignment (§13.1.4, p. 504). A base class almost always needs a destructor, so that it can make the destructor virtual. If a base class has an empty destructor in order to make it virtual, then the fact that the class has a destructor does not indicate that the assignment operator or copy constructor is also needed.

Virtual Destructors Turn Off Synthesized Move

The fact that a base class needs a virtual destructor has an important indirect impact on the definition of base and derived classes: If a class defines a destructor—even if it uses `= default` to use the synthesized version—the compiler will not synthesize a move operation for that class (§13.6.2, p. 537).

Exercises Section 15.7.1

Exercise 15.24: What kinds of classes need a virtual destructor? What operations must a virtual destructor perform?

15.7.2. Synthesized Copy Control and Inheritance

The synthesized copy-control members in a base or a derived class execute like any other synthesized constructor, assignment operator, or destructor: They memberwise initialize, assign, or destroy the members of the class itself. In addition, these synthesized members initialize, assign, or destroy the direct base part of an object by using the corresponding operation from the base class. For example,

- The synthesized `Bulk_quote` default constructor runs the `Disc_Quote` default constructor, which in turn runs the `Quote` default constructor.
- The `Quote` default constructor default initializes the `bookNo` member to the empty string and uses the in-class initializer to initialize `price` to zero.
- When the `Quote` constructor finishes, the `Disc_Quote` constructor continues, which uses the in-class initializers to initialize `qty` and `discount`.
- When the `Disc_quote` constructor finishes, the `Bulk_quote` constructor continues but has no other work to do.

Similarly, the synthesized `Bulk_quote` copy constructor uses the (synthesized) `Disc_quote` copy constructor, which uses the (synthesized) `Quote` copy constructor. The `Quote` copy constructor copies the `bookNo` and `price` members; and the `Disc_Quote` copy constructor copies the `qty` and `discount` members.

It is worth noting that it doesn't matter whether the base-class member is itself synthesized (as is the case in our `Quote` hierarchy) or has a an user-provided definition. All that matters is that the corresponding member is accessible (§15.5, p. 611) and that it is not a deleted function.

Each of our `Quote` classes use the synthesized destructor. The derived classes do so implicitly, whereas the `Quote` class does so explicitly by defining its (virtual) destructor as `= default`. The synthesized destructor is (as usual) empty and its implicit destruction part destroys the members of the class (§13.1.3, p. 501). In addition to destroying its own members, the destruction phase of a destructor in a derived class also destroys its direct base. That destructor in turn invokes the destructor for its own direct base, if any. And, so on up to the root of the hierarchy.

As we've seen, `Quote` does not have synthesized move operations because it defines a destructor. The (synthesized) copy operations will be used whenever we move a `Quote` object (§13.6.2, p. 540). As we're about to see, the fact that `Quote` does not have move operations means that its derived classes don't either.

Base Classes and Deleted Copy Control in the Derived

The synthesized default constructor, or any of the copy-control members of either a base or a derived class, may be defined as deleted for the same reasons as in any other class (§13.1.6, p. 508, and §13.6.2, p. 537). In addition, the way in which a base class is defined can cause a derived-class member to be defined as deleted:

- If the default constructor, copy constructor, copy-assignment operator, or destructor in the base class is deleted or inaccessible (§15.5, p. 612), then the corresponding member in the derived class is defined as deleted, because the compiler can't use the base-class member to construct, assign, or destroy the base-class part of the object.
- If the base class has an inaccessible or deleted destructor, then the synthesized default and copy constructors in the derived classes are defined as deleted, because there is no way to destroy the base part of the derived object.
- As usual, the compiler will not synthesize a deleted move operation. If we use `= default` to request a move operation, it will be a deleted function in the derived if the corresponding operation in the base is deleted or inaccessible, because the base class part cannot be moved. The move constructor will also be deleted if the base class destructor is deleted or inaccessible.

As an example, this base class, `B`,

Click here to view code image

```
class B {
public:
    B();
    B(const B&) = delete;
    //  other members, not including a move constructor
};
class D : public B {
    //  no constructors
};
D d;       //  ok:  D's synthesized default constructor uses  B's default constructor
D d2(d); //  error:  D's synthesized copy constructor is  deleted
D d3(std::move(d)); //  error: implicitly uses  D's deleted copy constructor
```

has an accessible default constructor and an explicitly deleted copy constructor. Because the copy constructor is defined, the compiler will not synthesize a move constructor for class B (§13.6.2, p. 537). As a result, we can neither move nor copy objects of type B. If a class derived from B wanted to allow its objects to be copied or moved, that derived class would have to define its own versions of these constructors. Of course, that class would have to decide how to copy or move the members in it base-class part. In practice, if a base class does not have a default, copy, or move constructor, then its derived classes usually don't either.

Move Operations and Inheritance

As we've seen, most base classes define a virtual destructor. As a result, by default, base classes generally do not get synthesized move operations. Moreover, by default, classes derived from a base class that doesn't have move operations don't get synthesized move operations either.

Because lack of a move operation in a base class suppresses synthesized move for its derived classes, base classes ordinarily should define the move operations if it is sensible to do so. Our Quote class can use the synthesized versions. However, Quote must define these members explicitly. Once it defines its move operations, it must also explicitly define the copy versions as well (§13.6.2, p. 539):

Click here to view code image

```
class Quote {
public:
    Quote() = default;                  //  memberwise default initialize
    Quote(const Quote&) = default; //  memberwise copy
    Quote(Quote&&) = default;      //  memberwise copy
    Quote& operator=(const Quote&) = default; //  copy assign
    Quote& operator=(Quote&&) = default;      //  move assign
    virtual ~Quote() = default;
    //  other members as before
};
```

Now, Quote objects will be memberwise copied, moved, assigned, and destroyed. Moreover, classes derived from Quote will automatically obtain synthesized move operations as well, unless they have members that otherwise preclude move.

Exercises Section 15.7.2

Exercise 15.25: Why did we define a default constructor for Disc_quote? What effect, if any, would removing that constructor have on the behavior of Bulk_quote?

15.7.3. Derived-Class Copy-Control Members

As we saw in §15.2.2 (p. 598), the initialization phase of a derived-class constructor initializes the base-class part(s) of a derived object as well as initializing its own members. As a result, the copy and move constructors for a derived class must copy/move the members of its base part as well as the members in the derived. Similarly, a derived-class assignment operator must assign the members in the base part of the derived object.

Unlike the constructors and assignment operators, the destructor is responsible only for destroying the resources allocated by the derived class. Recall that the members of an object are implicitly destroyed (§13.1.3, p. 502). Similarly, the base-class part of a derived object is destroyed automatically.

Warning

When a derived class defines a copy or move operation, that operation is responsible for copying or moving the entire object, including base-class members.

Defining a Derived Copy or Move Constructor

When we define a copy or move constructor (§13.1.1, p. 496, and §13.6.2, p. 534) for a derived class, we ordinarily use the corresponding base-class constructor to initialize the base part of the object:

Click here to view code image

```
class Base { /* …    */ } ;
class D: public Base {
public:
     // by default, the base class default constructor initializes the base part of an object
     // to use the copy or move constructor, we must explicitly call that
     // constructor in the constructor initializer list
     D(const D& d): Base(d)        // copy the base members
                      /* initializers for members of D */ { /* ...   */ }
     D(D&& d): Base(std::move(d)) // move the base members
                      /* initializers for members of D */ { /* ...   */ }
};
```

The initializer `Base(d)` passes a `D` object to a base-class constructor. Although in principle, `Base` could have a constructor that has a parameter of type `D`, in practice, that is very unlikely. Instead, `Base(d)` will (ordinarily) match the `Base` copy constructor. The `D` object, `d`, will be bound to the `Base&` parameter in that constructor. The `Base` copy constructor will copy the base part of `d` into the object that is being created. Had the initializer for the base class been omitted,

Click here to view code image

```
// probably incorrect definition of the D copy constructor
// base-class part is default initialized, not copied
D(const D& d) /* member initializers, but no base-class initializer    */
    { /* ...   */ }
```

the `Base` default constructor would be used to initialize the base part of a `D` object. Assuming `D`'s constructor copies the derived members from `d`, this newly constructed object would be oddly configured: Its `Base` members would hold default values, while its `D` members would be copies of the data from another object.

Warning

By default, the base-class default constructor initializes the base-class part of a derived object. If we want copy (or move) the base-class part, we must explicitly use the copy (or move) constructor for the base class in the derived's constructor initializer list.

Derived-Class Assignment Operator

Like the copy and move constructors, a derived-class assignment operator (§13.1.2, p. 500, and §13.6.2, p. 536), must assign its base part explicitly:

Click here to view code image

```
// Base::operator=(const Base&) is not invoked automatically
D &D::operator=(const D &rhs)
{
    Base::operator=(rhs); // assigns the base part
    // assign the members in the derived class, as usual,
    // handling self-assignment and freeing existing resources as appropriate
    return *this;
}
```

This operator starts by explicitly calling the base-class assignment operator to assign the members of the base part of the derived object. The base-class operator will (presumably) correctly handle self-assignment and, if appropriate, will free the old value in the base part of the left-hand operand and assign the new values from `rhs`. Once that operator finishes, we continue doing whatever is needed to assign the members in the derived class.

It is worth noting that a derived constructor or assignment operator can use its corresponding base class operation regardless of whether the base defined its own version of that operator or uses the synthesized version. For example, the call to `Base::operator=` executes the copy-assignment operator in class `Base`. It is immaterial whether that operator is defined explicitly by the `Base` class or is synthesized by the compiler.

Derived-Class Destructor

Recall that the data members of an object are implicitly destroyed after the destructor body completes (§13.1.3, p. 502). Similarly, the base-class parts of an object are also implicitly destroyed. As a result, unlike the constructors and assignment operators, a derived destructor is responsible only for destroying the resources allocated by the derived class:

Click here to view code image

```
class D: public Base {
public:
    // Base::~Base invoked automatically
    ~D() { /* do what it takes to clean up derived members */ }
};
```

Objects are destroyed in the opposite order from which they are constructed: The derived destructor is run first, and then the base-class destructors are invoked, back up through the inheritance hierarchy.

Calls to Virtuals in Constructors and Destructors

As we've seen, the base-class part of a derived object is constructed first. While the base-class constructor is executing, the derived part of the object is uninitialized. Similarly, derived objects are destroyed in reverse order, so that when a base class destructor runs, the derived part has already been destroyed. As a result, while these base-class members are executing, the object is incomplete.

To accommodate this incompleteness, the compiler treats the object as if its type changes during construction or destruction. That is, while an object is being constructed it is treated as if it has the same class as the constructor; calls to virtual functions will be bound as if the object has the same type as the constructor itself. Similarly, for destructors. This binding applies to virtuals called directly or that are called indirectly from a function that the constructor (or destructor) calls.

To understand this behavior, consider what would happen if the derived-class version of a virtual was called from a base-class constructor. This virtual probably accesses members of the derived object. After all, if the virtual didn't need to use members of the derived object, the derived class probably could use the version in its base class. However, those members are uninitialized while a base constructor is running. If such access were allowed, the program would probably crash.

Note

If a constructor or destructor calls a virtual, the version that is run is the one corresponding to the type of the constructor or destructor itself.

Exercises Section 15.7.3

Exercise 15.26: Define the `Quote` and `Bulk_quote` copy-control members to do the same job as the synthesized versions. Give them and the other constructors print statements that identify which function is running. Write programs using these classes and predict what objects will be created and destroyed. Compare your predictions with the output and continue experimenting until your predictions are reliably correct.

15.7.4. Inherited Constructors

Under the new standard, a derived class can reuse the constructors defined by its direct base class. Although, as we'll see, such constructors are not inherited in the normal sense of that term, it is nonetheless common to refer to such constructors as "inherited." For the same reasons that a class may initialize only its direct base class, a class may inherit constructors only from its direct base. A class cannot inherit the default, copy, and move constructors. If the derived class does not directly define these constructors, the compiler synthesizes them as usual.

A derived class inherits its base-class constructors by providing a `using` declaration that names its (direct) base class. As an example, we can redefine our `Bulk_quote` class (§15.4, p. 610) to inherit its constructors from `Disc_quote`:

Click here to view code image

```
class Bulk_quote : public Disc_quote {
public:
    using Disc_quote::Disc_quote; // inherit Disc_quote's constructors
    double net_price(std::size_t) const;
};
```

Ordinarily, a `using` declaration only makes a name visible in the current scope. When applied to a constructor, a `using` declaration causes the compiler to generate code. The compiler generates a derived constructor corresponding to each constructor in the base. That is, for each constructor in the base class, the compiler generates a constructor in the derived class that has the same parameter list.

These compiler-generated constructors have the form

```
derived(parms) : base(args) { }
```

where *derived* is the name of the derived class, *base* is the name of the base class, *parms* is the parameter list of the constructor, and *args* pass the parameters from the derived constructor to the base constructor. In our `Bulk_quote` class, the inherited constructor would be equivalent to

Click here to view code image

```
Bulk_quote(const std::string& book, double price,
              std::size_t qty, double disc):
        Disc_quote(book, price, qty, disc) { }
```

If the derived class has any data members of its own, those members are default initialized (§7.1.4, p. 266).

Characteristics of an Inherited Constructor

Unlike `using` declarations for ordinary members, a constructor `using` declaration does not change the access level of the inherited constructor(s). For example, regardless of where the `using` declaration appears, a `private` constructor in the base is a `private` constructor in the derived; similarly for `protected` and `public` constructors.

Moreover, a `using` declaration can't specify `explicit` or `constexpr`. If a constructor in the base is `explicit` (§7.5.4, p. 296) or `constexpr` (§7.5.6, p. 299), the inherited constructor has the same property.

If a base-class constructor has default arguments (§6.5.1, p. 236), those arguments are not inherited. Instead, the derived class gets multiple inherited constructors in which each parameter with a default argument is successively omitted. For example, if the base has a constructor with two parameters, the second of which has a default, the derived class will obtain two constructors: one with both parameters (and no default argument) and a second constructor with a single parameter corresponding to the left-most, non-defaulted parameter in the base class.

If a base class has several constructors, then with two exceptions, the derived class inherits each of the constructors from its base class. The first exception is that a derived class can inherit some constructors and define its own versions of other constructors. If the derived class defines a constructor with the same parameters as a constructor in the base, then that constructor is not inherited. The one defined in the derived class is used in place of the inherited constructor.

The second exception is that the default, copy, and move constructors are not inherited. These constructors are synthesized using the normal rules. An inherited constructor is not treated as a user-defined constructor. Therefore, a class that contains only inherited constructors will have a synthesized default constructor.

Exercises Section 15.7.4

Exercise 15.27: Redefine your `Bulk_quote` class to inherit its constructors.

15.8. Containers and Inheritance

When we use a container to store objects from an inheritance hierarchy, we generally must store those objects indirectly. We cannot put objects of types related by inheritance directly into a container, because there is no way to define a container that holds elements of differing types.

As an example, assume we want to define a `vector` to hold several books that a customer wants to buy. It should be easy to see that we can't use a `vector` that holds `Bulk_quote` objects. We can't convert `Quote` objects to `Bulk_quote` (§15.2.3, p. 602), so we wouldn't be able to put `Quote` objects into that `vector`.

It may be somewhat less obvious that we also can't use a `vector` that holds objects of type `Quote`. In this case, we can put `Bulk_quote` objects into the container. However, those objects would no longer be `Bulk_quote` objects:

Click here to view code image

```
vector<Quote> basket;
basket.push_back(Quote("0-201-82470-1", 50));
// ok, but copies only the Quote part of the object into basket
basket.push_back(Bulk_quote("0-201-54848-8", 50, 10, .25));
// calls version defined by Quote, prints 750, i.e., 15 * $50
cout << basket.back().net_price(15) << endl;
```

The elements in `basket` are `Quote` objects. When we add a `Bulk_quote` object to the `vector` its derived part is ignored (§15.2.3, p. 603).

Warning

Because derived objects are "sliced down" when assigned to a base-type object, containers and types related by inheritance do not mix well.

Put (Smart) Pointers, Not Objects, in Containers

When we need a container that holds objects related by inheritance, we typically define the container to hold pointers (preferably smart pointers (§12.1, p. 450)) to the base class. As usual, the dynamic type of the object to which those pointers point might be the base-class type or a type derived from that base:

Click here to view code image

```
vector<shared_ptr<Quote>> basket;
basket.push_back(make_shared<Quote>("0-201-82470-1", 50));
basket.push_back(
    make_shared<Bulk_quote>("0-201-54848-8", 50, 10, .25));
// calls the version defined by Quote; prints 562.5, i.e., 15 * $50 less the discount
cout << basket.back()->net_price(15) << endl;
```

Because `basket` holds `shared_ptr`s, we must dereference the value returned by `basket.back()` to get the object on which to run `net_price`. We do so by using -> in the call to `net_price`. As usual, the version of `net_price` that is called depends on the dynamic type of the object to which that pointer points.

It is worth noting that we defined `basket` as `shared_ptr<Quote>`, yet in the second `push_back` we passed a `shared_ptr` to a `Bulk_quote` object. Just as we can convert an ordinary pointer to a derived type to a pointer to an base-class type (§15.2.2, p. 597), we can also convert a smart pointer to a derived type to a smart pointer to an base-class type. Thus, `make_shared<Bulk_quote>` returns a `shared_ptr<Bulk_quote>` object, which is converted to `shared_ptr<Quote>` when we call `push_back`. As a result, despite appearances, all of the elements of `basket` have the same type.

Exercises Section 15.8

Exercise 15.28: Define a `vector` to hold `Quote` objects but put `Bulk_quote` objects into that `vector`. Compute the total `net_price` of all the elements in the `vector`.

Exercise 15.29: Repeat your program, but this time store `shared_ptrs` to objects of type `Quote`. Explain any discrepancy in the sum generated by the this version and the previous program. If there is no discrepancy, explain why there isn't one.

15.8.1. Writing a Basket Class

One of the ironies of object-oriented programming in C++ is that we cannot use objects directly to support it. Instead, we must use pointers and references. Because pointers impose complexity on our programs, we often define auxiliary classes to help manage that complexity. We'll start by defining a class to represent a basket:

Click here to view code image

```
class Basket {
public:
    // Basket uses synthesized default constructor and copy-control members
    void add_item(const std::shared_ptr<Quote> &sale)
        { items.insert(sale); }
    // prints the total price for each book and the overall total for all items in the basket
    double total_receipt(std::ostream&) const;
private:
    // function to compare shared_ptrs needed by the multiset member
    static bool compare(const std::shared_ptr<Quote> &lhs,
                        const std::shared_ptr<Quote> &rhs)
    { return lhs->isbn() < rhs->isbn(); }
    // multiset to hold multiple quotes, ordered by the compare member
    std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
                  items{compare};
};
```

Our class uses a `multiset` (§11.2.1, p. 423) to hold the transactions, so that we can store multiple transactions for the same book, and so that all the transactions for a given book will be kept together (§11.2.2, p. 424).

The elements in our `multiset` are `shared_ptr`s and there is no less-than operator for `shared_ptr`. As a result, we must provide our own comparison operation to order the elements (§11.2.2, p. 425). Here, we define a `private static` member, named `compare`, that compares the `isbn`s of the objects to which the `shared_ptr`s point. We initialize our `multiset` to use this comparison function through an in-class initializer (§7.3.1, p. 274):

Click here to view code image

```
// multiset to hold multiple quotes, ordered by the compare member
std::multiset<std::shared_ptr<Quote>, decltype(compare)*>
              items{compare};
```

This declaration can be hard to read, but reading from left to right, we see that we are defining a `multiset` of `shared_ptr`s to `Quote` objects. The `multiset` will use a function with the same type as our `compare` member to order the elements. The `multiset` member is named `items`, and we're initializing `items` to use our `compare` function.

Defining the Members of Basket

The `Basket` class defines only two operations. We defined the `add_item` member inside the class. That member takes a `shared_ptr` to a dynamically allocated `Quote` and puts that `shared_ptr` into the `multiset`. The second member, `total_receipt`, prints an itemized bill for the contents of the basket and returns the price for all the items in the basket:

Click here to view code image

```
double Basket::total_receipt(ostream &os) const
{
    double sum = 0.0;    // holds the running total
    // iter refers to the first element in a batch of elements with the same ISBN
    // upper_bound returns an iterator to the element just past the end of that batch
    for (auto iter = items.cbegin();
              iter != items.cend();
              iter = items.upper_bound(*iter)) {
        // we know there's at least one element with this key in the Basket
        // print the line item for this book
        sum += print_total(os, **iter, items.count(*iter));
    }
     os << "Total Sale: " << sum << endl; // print the final overall
total
    return sum;
}
```

Our `for` loop starts by defining and initializing `iter` to refer to the first element in the `multiset`. The condition checks whether `iter` is equal to `items.cend()`. If so, we've processed all the purchases and we drop out of the `for`. Otherwise, we process the next book.

The interesting bit is the "increment" expression in the `for`. Rather than the usual loop that reads each element, we advance `iter` to refer to the next key. We skip over all the elements that match the current key by calling `upper_bound` (§11.3.5, p. 438). The call to `upper_bound` returns the iterator that refers to the element just past the last one with the same key as in `iter`. The iterator we get back denotes either the end of the set or the next book.

Inside the `for` loop, we call `print_total` (§15.1, p. 593) to print the details for each book in the basket:

Click here to view code image

```
    sum += print_total(os, **iter, items.count(*iter));
```

The arguments to `print_total` are an `ostream` on which to write, a `Quote` object to process, and a count. When we dereference `iter`, we get a `shared_ptr` that points to the object we want to print. To get that object, we must dereference that `shared_ptr`. Thus, `**iter` is a `Quote` object (or an object of a type derived from `Quote`). We use the `multiset count` member (§11.3.5, p. 436) to determine how many elements in the `multiset` have the same key (i.e., the same ISBN).

As we've seen, `print_total` makes a virtual call to `net_price`, so the resulting price depends on the dynamic type of `**iter`. The `print_total` function prints the total for the given book and returns the total price that it calculated. We add that result into `sum`, which we print after we complete the `for` loop.

Hiding the Pointers

Users of `Basket` still have to deal with dynamic memory, because `add_item` takes a shared_ptr. As a result, users have to write code such as

Click here to view code image

```
Basket bsk;
bsk.add_item(make_shared<Quote>("123", 45));
bsk.add_item(make_shared<Bulk_quote>("345", 45, 3, .15));
```

Our next step will be to redefine add_item so that it takes a Quote object instead of a shared_ptr. This new version of add_item will handle the memory allocation so that our users no longer need to do so. We'll define two versions, one that will copy its given object and the other that will move from it (§13.6.3, p. 544):

Click here to view code image

```
void add_item(const Quote& sale);   // copy the given object
void add_item(Quote&& sale);        // move the given object
```

The only problem is that add_item doesn't know what type to allocate. When it does its memory allocation, add_item will copy (or move) its sale parameter. Somewhere there will be a new expression such as:

```
new Quote(sale)
```

Unfortunately, this expression won't do the right thing: new allocates an object of the type we request. This expression allocates an object of type Quote and copies the Quote portion of sale. However, sale might refer to a Bulk_quote object, in which case, that object will be sliced down.

Simulating Virtual Copy

We'll solve this problem by giving our Quote classes a virtual member that allocates a copy of itself.

Click here to view code image

```
class Quote {
public:
    // virtual function to return a dynamically allocated copy of itself
    // these members use reference qualifiers; see §13.6.3 (p. 546)
         virtual   Quote*   clone()   const   &   {return   new
Quote(*this);}
    virtual Quote* clone() &&
                                                  {return   new
Quote(std::move(*this));}
    // other members as before
};
class Bulk_quote : public Quote {
           Bulk_quote*   clone()   const   &   {return   new
Bulk_quote(*this);}
```

```
        Bulk_quote* clone() &&
                                                   {return  new
    Bulk_quote(std::move(*this));}
        // other members as before
    };
```

Because we have a copy and a move version of `add_item`, we defined lvalue and rvalue versions of `clone` (§13.6.3, p. 546). Each `clone` function allocates a new object of its own type. The `const` lvalue reference member copies itself into that newly allocated object; the rvalue reference member moves its own data.

Using `clone`, it is easy to write our new versions of `add_item`:

Click here to view code image

```
class Basket {
public:
    void add_item(const Quote& sale) // copy the given object
      { items.insert(std::shared_ptr<Quote>(sale.clone())); }
    void add_item(Quote&& sale)      // move the given object
      { items.insert(
          std::shared_ptr<Quote>(std::move(sale).clone())); }
    // other members as before
};
```

Like `add_item` itself, `clone` is overloaded based on whether it is called on an lvalue or an rvalue. Thus, the first version of `add_item` calls the `const` lvalue version of `clone`, and the second version calls the rvalue reference version. Note that in the rvalue version, although the type of `sale` is an rvalue reference type, `sale` (like any other variable) is an lvalue (§13.6.1, p. 533). Therefore, we call `move` to bind an rvalue reference to `sale`.

Our `clone` function is also virtual. Whether the `Quote` or `Bulk_quote` function is run, depends (as usual) on the dynamic type of `sale`. Regardless of whether we copy or move the data, `clone` returns a pointer to a newly allocated object, of its own type. We bind a `shared_ptr` to that object and call `insert` to add this newly allocated object to `items`. Note that because `shared_ptr` supports the derived-to-base conversion (§15.2.2, p. 597), we can bind a `shared_ptr<Quote` to a `Bulk_quote*`.

Exercises Section 15.8.1

Exercise 15.30: Write your own version of the `Basket` class and use it to compute prices for the same transactions as you used in the previous exercises.

15.9. Text Queries Revisited

As a final example of inheritance, we'll extend our text-query application from §12.3 (p. 484). The classes we wrote in that section let us look for occurrences of a given word in a file. We'd like to extend the system to support more complicated queries. In our examples, we'll run queries against the following simple story:

```
Alice Emma has long flowing red hair.
Her Daddy says when the wind blows
through her hair, it looks almost alive,
like a fiery bird in flight.
A beautiful fiery bird, he tells her,
magical but untamed.
"Daddy, shush, there is no such thing,"
she tells him, at the same time wanting
him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"
```

Our system should support the following queries:

• Word queries find all the lines that match a given `string`:

```
Executing Query for:
Daddy Daddy occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 7) "Daddy, shush, there is no such thing,"
(line 10) Shyly, she asks, "I mean, Daddy, is there?"
```

• Not queries, using the ~ operator, yield lines that don't match the query:

```
Executing Query for: ~(Alice)
~(Alice) occurs 9 times
(line 2) Her Daddy says when the wind blows
(line 3) through her hair, it looks almost alive,
(line 4) like a fiery bird in flight.
. . .
```

• Or queries, using the | operator, return lines matching either of two queries:

```
Executing Query for: (hair | Alice)
(hair | Alice) occurs 2 times
(line 1) Alice Emma has long flowing red hair.
(line 3) through her hair, it looks almost alive,
```

• And queries, using the & operator, return lines matching both queries:

```
Executing query for: (hair & Alice)
(hair & Alice) occurs 1 time
(line 1) Alice Emma has long flowing red hair.
```

Moreover, we want to be able to combine these operations, as in

```
fiery & bird | wind
```

We'll use normal C++ precedence rules (§4.1.2, p. 136) to evaluate compound expressions such as this example. Thus, this query will match a line in which both `fiery` and `bird` appear or one in which `wind` appears:

```
Executing Query for: ((fiery & bird) | wind)
((fiery & bird) | wind) occurs 3 times
(line 2) Her Daddy says when the wind blows
(line 4) like a fiery bird in flight.
(line 5) A beautiful fiery bird, he tells her,
```

Our output will print the query, using parentheses to indicate the way in which the query was interpreted. As with our original implementation, our system will display lines in ascending order and will not display the same line more than once.

15.9.1. An Object-Oriented Solution

We might think that we should use the `TextQuery` class from §12.3.2 (p. 487) to represent our word query and derive our other queries from that class.

However, this design would be flawed. To see why, consider a Not query. A Word query looks for a particular word. In order for a Not query to be a kind of Word query, we would have to be able to identify the word for which the Not query was searching. In general, there is no such word. Instead, a Not query has a query (a Word query or any other kind of query) whose value it negates. Similarly, an And query and an Or query have two queries whose results it combines.

This observation suggests that we model our different kinds of queries as independent classes that share a common base class:

```
WordQuery // Daddy
NotQuery  // ~Alice
OrQuery   // hair | Alice
AndQuery  // hair & Alice
```

These classes will have only two operations:

- `eval`, which takes a `TextQuery` object and returns a `QueryResult`. The `eval` function will use the given `TextQuery` object to find the query's the matching lines.
- `rep`, which returns the `string` representation of the underlying query. This function will be used by `eval` to create a `QueryResult` representing the match and by the output operator to print the query expressions.

Abstract Base Class

As we've seen, our four query types are not related to one another by inheritance; they are conceptually siblings. Each class shares the same interface, which suggests that we'll need to define an abstract base class (§15.4, p. 610) to represent that interface. We'll name our abstract base class `Query_base`, indicating that its role is to serve as the root of our query hierarchy.

Our `Query_base` class will define `eval` and `rep` as pure virtual functions (§15.4, p. 610). Each of our classes that represents a particular kind of query must override these functions. We'll derive `WordQuery` and `NotQuery` directly from `Query_base`. The `AndQuery` and `OrQuery` classes share one property that the other classes in our system do not: Each has two operands. To model this property, we'll define another abstract base class, named `BinaryQuery`, to represent queries with two operands. The `AndQuery` and `OrQuery` classes will inherit from `BinaryQuery`, which in turn will inherit from `Query_base`. These decisions give us the class design represented in Figure 15.2.

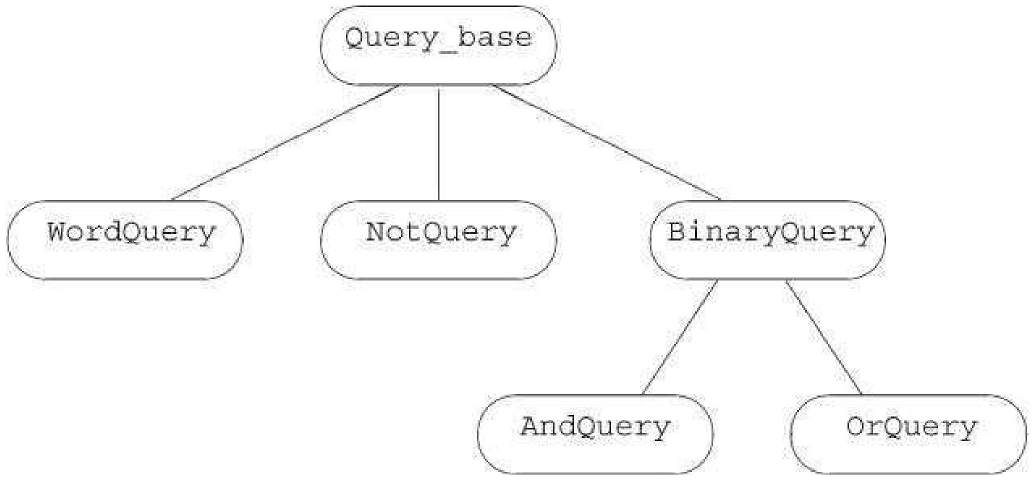


Figure 15.2. Query_base Inheritance Hierarchy

Key Concept: Inheritance versus Composition

The design of inheritance hierarchies is a complicated topic in its own right and well beyond the scope of this language Primer. However, there is one important design guide that is so fundamental that every programmer should be familiar with it.

When we define a class as publicly inherited from another, the derived class should reflect an "Is A" relationship to the base class. In well-designed class hierarchies, objects of a publicly derived class can be used wherever an object of the base class is expected.

Another common relationship among types is a "Has A" relationship. Types related by a "Has A" relationship imply membership.

In our bookstore example, our base class represents the concept of a quote for a book sold at a stipulated price. Our `Bulk_quote` "is a" kind of quote, but one with a different pricing strategy. Our bookstore classes "have a" price and an ISBN.

Hiding a Hierarchy in an Interface Class

Our program will deal with evaluating queries, not with building them. However, we need to be able to create queries in order to run our program. The simplest way to do so is to write C++ expressions to create the queries. For example, we'd like to generate the compound query previously described by writing code such as

Click here to view code image

```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

This problem description implicitly suggests that user-level code won't use the inherited classes directly. Instead, we'll define an interface class named `Query`, which will hide the hierarchy. The `Query` class will store a pointer to `Query_base`. That pointer will be bound to an object of a type derived from `Query_base`. The `Query` class will provide the same operations as the `Query_base` classes: `eval` to evaluate the associated query, and `rep` to generate a `string` version of the query. It will also define an overloaded output operator to display the associated query.

Users will create and manipulate `Query_base` objects only indirectly through operations on `Query` objects. We'll define three overloaded operators on `Query` objects, along with a `Query` constructor that takes a `string`. Each of these functions will dynamically allocate a new object of a type derived from `Query_base`:

- The `&` operator will generate a `Query` bound to a new `AndQuery`.
- The `|` operator will generate a `Query` bound to a new `OrQuery`.
- The `~` operator will generate a `Query` bound to a new `NotQuery`.
- The `Query` constructor that takes a `string` will generate a new `WordQuery`.

Understanding How These Classes Work

It is important to realize that much of the work in this application consists of building objects to represent the user's query. For example, an expression such as the one above generates the collection of interrelated objects illustrated in Figure 15.3.

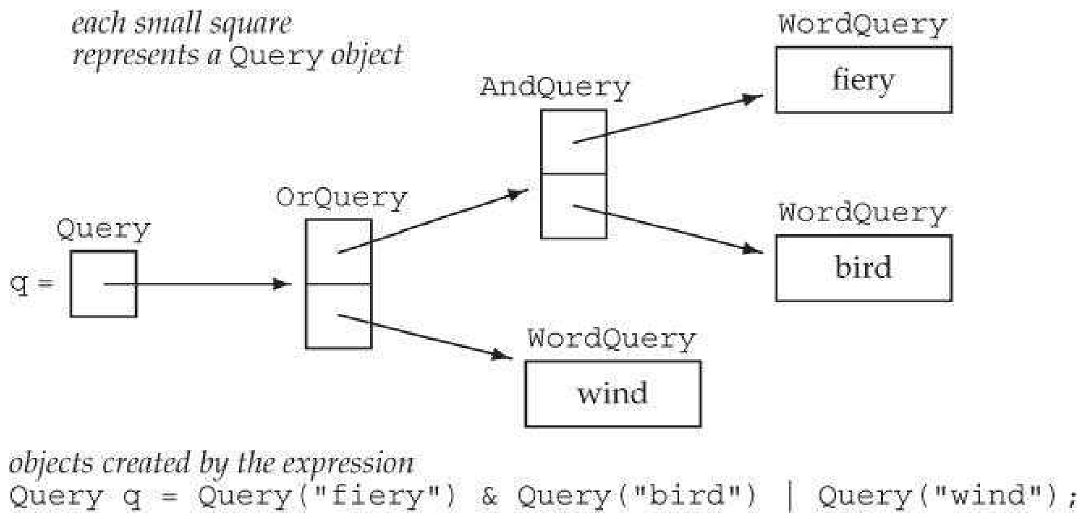
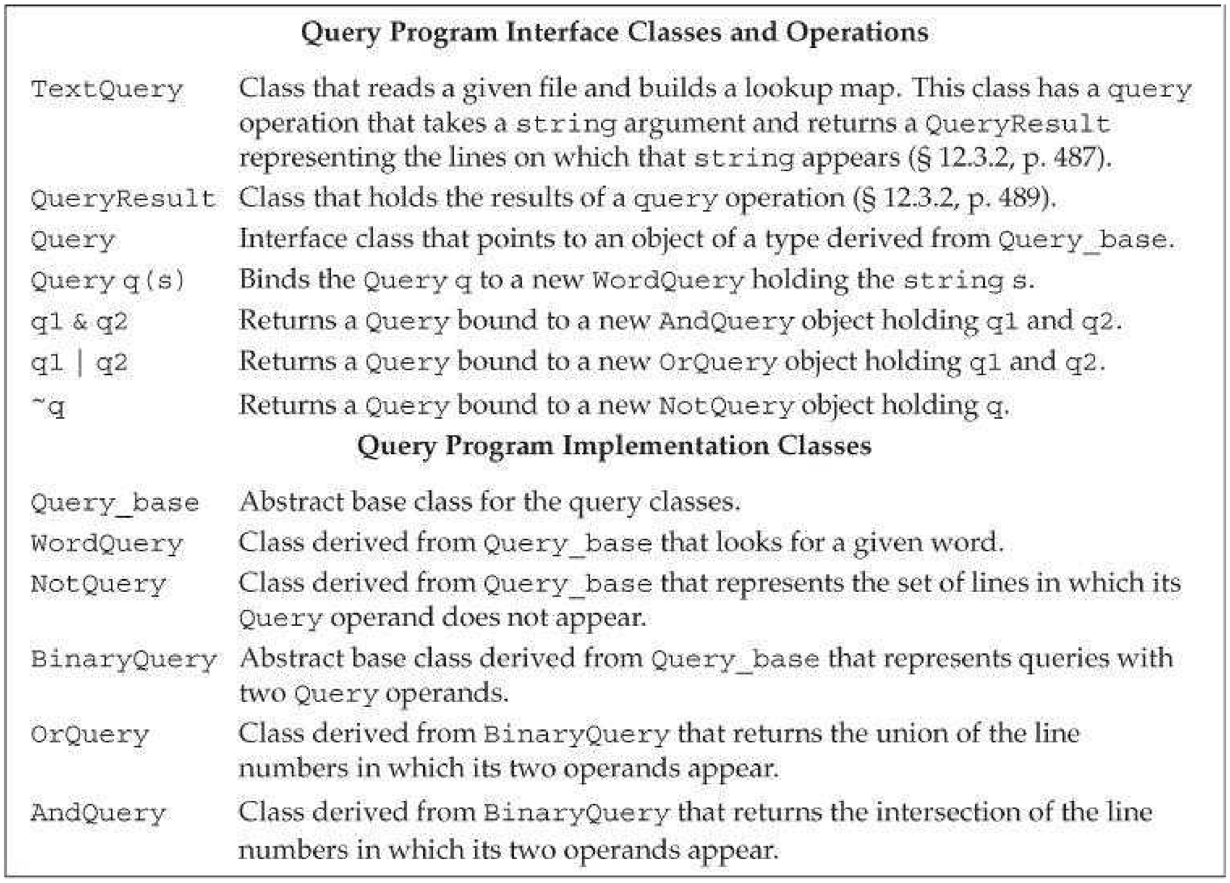

each small square represents a Query object
WordQuery
fiery
AndQuery
OrQuery
WordQuery
bird
Query
q =
WordQuery
wind
objects created by the expression
Query q = Query("fiery") & Query("bird") | Query("wind");

Figure 15.3. Objects Created by Query Expressions

Once the tree of objects is built up, evaluating (or generating the representation of) a query is basically a process (managed for us by the compiler) of following these links, asking each object to evaluate (or display) itself. For example, if we call `eval` on `q` (i.e., on the root of the tree), that call asks the `OrQuery` to which `q` points to `eval` itself. Evaluating this `OrQuery` calls `eval` on its two operands—on the `AndQuery` and the `WordQuery` that looks for the word `wind`. Evaluating the `AndQuery` evaluates its two `WordQuery`s, generating the results for the words `fiery` and `bird`, respectively.

When new to object-oriented programming, it is often the case that the hardest part in understanding a program is understanding the design. Once you are thoroughly comfortable with the design, the implementation flows naturally. As an aid to understanding this design, we've summarized the classes used in this example in Table 15.1 (overleaf).

Table 15.1. Recap: Query Program Design



Query Program Interface Classes and Operations	
`TextQuery`	Class that reads a given file and builds a lookup map. This class has a `query` operation that takes a `string` argument and returns a `QueryResult` representing the lines on which that `string` appears (§ 12.3.2, p. 487).
`QueryResult`	Class that holds the results of a `query` operation (§ 12.3.2, p. 489).
`Query`	Interface class that points to an object of a type derived from `Query_base`.
`Query q(s)`	Binds the `Query q` to a new `WordQuery` holding the `string s`.
`q1 & q2`	Returns a `Query` bound to a new `AndQuery` object holding `q1` and `q2`.
`q1 | q2`	Returns a `Query` bound to a new `OrQuery` object holding `q1` and `q2`.
`~q`	Returns a `Query` bound to a new `NotQuery` object holding `q`.
Query Program Implementation Classes	
`Query_base`	Abstract base class for the query classes.
`WordQuery`	Class derived from `Query_base` that looks for a given word.
`NotQuery`	Class derived from `Query_base` that represents the set of lines in which its `Query` operand does not appear.
`BinaryQuery`	Abstract base class derived from `Query_base` that represents queries with two `Query` operands.
`OrQuery`	Class derived from `BinaryQuery` that returns the union of the line numbers in which its two operands appear.
`AndQuery`	Class derived from `BinaryQuery` that returns the intersection of the line numbers in which its two operands appear.

Exercises Section 15.9.1

Exercise 15.31: Given that `s1`, `s2`, `s3`, and `s4` are all `strings`, determine what objects are created in the following expressions:

Click here to view code image

(a) `Query(s1) | Query(s2) & ~ Query(s3);`

(b) `Query(s1) | (Query(s2) & ~ Query(s3));`

(c) `(Query(s1) & (Query(s2)) | (Query(s3) & Query(s4)));`

15.9.2. The `Query_base` and `Query` Classes

We'll start our implementation by defining the `Query_base` class:

Click here to view code image

```
// abstract class acts as a base class for concrete query types; all members are private
class Query_base {
    friend class Query;
protected:
        using line_no = TextQuery::line_no; // used in the eval
functions
    virtual ~Query_base() = default;
private:
    // eval returns the QueryResult that matches this Query
    virtual QueryResult eval(const TextQuery&) const = 0;
    // rep is a string representation of the query
    virtual std::string rep() const = 0;
};
```

Both `eval` and `rep` are pure virtual functions, which makes `Query_base` an abstract base class (§15.4, p. 610). Because we don't intend users, or the derived classes, to use `Query_base` directly, `Query_base` has no `public` members. All use of `Query_base` will be through `Query` objects. We grant friendship to the `Query` class, because members of `Query` will call the virtuals in `Query_base`.

The `protected` member, `line_no`, will be used inside the `eval` functions. Similarly, the destructor is `protected` because it is used (implicitly) by the destructors in the derived classes.

The Query Class

The `Query` class provides the interface to (and hides) the `Query_base` inheritance hierarchy. Each `Query` object will hold a `shared_ptr` to a corresponding `Query_base` object. Because `Query` is the only interface to the `Query_base` classes, `Query` must define its own versions of `eval` and `rep`.

The `Query` constructor that takes a `string` will create a new `WordQuery` and bind its `shared_ptr` member to that newly created object. The `&`, `|`, and `~` operators will create `AndQuery`, `OrQuery`, and `NotQuery` objects, respectively. These operators will return a `Query` object bound to its newly generated object. To support these operators, `Query` needs a constructor that takes a `shared_ptr` to a `Query_base` and stores its given pointer. We'll make this constructor `private` because we don't intend general user code to define `Query_base` objects. Because this constructor is `private`, we'll need to make the operators `friends`.

Given the preceding design, the `Query` class itself is simple:

Click here to view code image

```
// interface class to manage the Query_base inheritance hierarchy
class Query {
    // these operators need access to the shared_ptr constructor
    friend Query operator~(const Query &);
    friend Query operator|(const Query&, const Query&);
    friend Query operator&(const Query&, const Query&);
public:
    Query(const std::string&);  // builds a new WordQuery
    // interface functions: call the corresponding Query_base operations
```

```
        QueryResult eval(const TextQuery &t) const
                                { return q->eval(t); }
        std::string rep() const { return q->rep(); }
    private:
        Query(std::shared_ptr<Query_base> query): q(query) { }
        std::shared_ptr<Query_base> q;
    };
```

We start by naming as friends the operators that create `Query` objects. These operators need to be friends in order to use the `private` constructor.

In the `public` interface for `Query`, we declare, but cannot yet define, the constructor that takes a `string`. That constructor creates a `WordQuery` object, so we cannot define this constructor until we have defined the `WordQuery` class.

The other two `public` members represent the interface for `Query_base`. In each case, the `Query` operation uses its `Query_base` pointer to call the respective (virtual) `Query_base` operation. The actual version that is called is determined at run time and will depend on the type of the object to which `q` points.

The Query Output Operator

The output operator is a good example of how our overall query system works:

Click here to view code image

```
    std::ostream &
    operator<<(std::ostream &os, const Query &query)
    {
        // Query::rep makes a virtual call through its Query_base pointer to rep()
        return os << query.rep();
    }
```

When we print a `Query`, the output operator calls the (public) `rep` member of class `Query`. That function makes a virtual call through its pointer member to the `rep` member of the object to which this `Query` points. That is, when we write

Click here to view code image

```
    Query andq = Query(sought1) & Query(sought2);
    cout << andq << endl;
```

the output operator calls `Query::rep` on `andq`. `Query::rep` in turn makes a virtual call through its `Query_base` pointer to the `Query_base` version of `rep`. Because `andq` points to an `AndQuery` object, that call will run `AndQuery::rep`.

Exercises Section 15.9.2

Exercise 15.32: What happens when an object of type `Query` is copied, moved, assigned, and destroyed?

Exercise 15.33: What about objects of type `Query_base`?

15.9.3. The Derived Classes

The most interesting part of the classes derived from `Query_base` is how they are represented. The `WordQuery` class is most straightforward. Its job is to hold the search word.

The other classes operate on one or two operands. A `NotQuery` has a single operand, and `AndQuery` and `OrQuery` have two operands. In each of these classes, the operand(s) can be an object of any of the concrete classes derived from `Query_base`: A `NotQuery` can be applied to a `WordQuery`, an `AndQuery`, an `OrQuery`, or another `NotQuery`. To allow this flexibility, the operands must be stored as pointers to `Query_base`. That way we can bind the pointer to whichever concrete class we need.

However, rather than storing a `Query_base` pointer, our classes will themselves use a `Query` object. Just as user code is simplified by using the interface class, we can simplify our own class code by using the same class.

Now that we know the design for these classes, we can implement them.

The WordQuery Class

A `WordQuery` looks for a given `string`. It is the only operation that actually performs a query on the given `TextQuery` object:

Click here to view code image

```
class WordQuery: public Query_base {
    friend class Query; // Query uses the WordQuery constructor
    WordQuery(const std::string &s): query_word(s) { }
    // concrete class: WordQuery defines all inherited pure virtual functions
    QueryResult eval(const TextQuery &t) const
                     { return t.query(query_word); }
    std::string rep() const { return query_word; }
    std::string query_word;    // word for which to search
};
```

Like `Query_base`, `WordQuery` has no `public` members; `WordQuery` must make `Query` a friend in order to allow `Query` to access the `WordQuery` constructor.

Each of the concrete query classes must define the inherited pure virtual functions, `eval` and `rep`. We defined both operations inside the `WordQuery` class body: `eval` calls the `query` member of its given `TextQuery` parameter, which does the actual search in the file; `rep` returns the `string` that this `WordQuery` represents (i.e., `query_word`).

Having defined the `WordQuery` class, we can now define the `Query` constructor that takes a `string`:

Click here to view code image

```
inline
Query::Query(const std::string &s): q(new WordQuery(s)) { }
```

This constructor allocates a `WordQuery` and initializes its pointer member to point to that newly allocated object.

The NotQuery Class and the ~ Operator

The ~ operator generates a `NotQuery`, which holds a `Query`, which it negates:

Click here to view code image

```
class NotQuery: public Query_base {
    friend Query operator~(const Query &);
    NotQuery(const Query &q): query(q) { }
    // concrete class: NotQuery defines all inherited pure virtual functions
      std::string  rep()  const  {return  "~("  +  query.rep()  +
")";}
    QueryResult eval(const TextQuery&) const;
    Query query;
};
inline Query operator~(const Query &operand)
{
                    return     std::shared_ptr<Query_base>(new
NotQuery(operand));
}
```

Because the members of `NotQuery` are all `private`, we start by making the ~ operator a friend. To `rep` a `NotQuery`, we concatenate the ~ symbol to the representation of the underlying `Query`. We parenthesize the output to ensure that precedence is clear to the reader.

It is worth noting that the call to `rep` in `NotQuery`’s own `rep` member ultimately makes a virtual call to `rep`: `query.rep()` is a nonvirtual call to the `rep` member of the `Query` class. `Query::rep` in turn calls `q->rep()`, which is a virtual call through its `Query_base` pointer.

The ~ operator dynamically allocates a new `NotQuery` object. The return (implicitly) uses the `Query` constructor that takes a `shared_ptr<Query_base>`. That is, the `return` statement is equivalent to

Click here to view code image

```
// allocate a new NotQuery object
// bind the resulting NotQuery pointer to a shared_ptr<Query_base
shared_ptr<Query_base> tmp(new NotQuery(expr));
```

```
    return Query(tmp); // use the Query constructor that takes a shared_ptr
```

The `eval` member is complicated enough that we will implement it outside the class body. We'll define the `eval` functions in §15.9.4 (p. 647).

The BinaryQuery Class

The `BinaryQuery` class is an abstract base class that holds the data needed by the query types that operate on two operands:

Click here to view code image

```
class BinaryQuery: public Query_base {
protected:
     BinaryQuery(const Query &l, const Query &r, std::string
s):
          lhs(l), rhs(r), opSym(s) { }
    // abstract class: BinaryQuery doesn't define eval
    std::string rep() const { return "(" + lhs.rep() + " "
                                     + opSym + " "
                                      + rhs.rep() + ")";
}
    Query lhs, rhs;    // right- and left-hand operands
    std::string opSym; // name of the operator
};
```

The data in a `BinaryQuery` are the two `Query` operands and the corresponding operator symbol. The constructor takes the two operands and the operator symbol, each of which it stores in the corresponding data members.

To `rep` a `BinaryOperator`, we generate the parenthesized expression consisting of the representation of the left-hand operand, followed by the operator, followed by the representation of the right-hand operand. As when we displayed a `NotQuery`, the calls to `rep` ultimately make virtual calls to the `rep` function of the `Query_base` objects to which `lhs` and `rhs` point.

Note

The `BinaryQuery` class does not define the `eval` function and so inherits a pure virtual. Thus, `BinaryQuery` is also an abstract base class, and we cannot create objects of `BinaryQuery` type.

The AndQuery and OrQuery Classes and Associated Operators

The `AndQuery` and `OrQuery` classes, and their corresponding operators, are quite similar to one another:

Click here to view code image

```
class AndQuery: public BinaryQuery {
    friend Query operator& (const Query&, const Query&);
    AndQuery(const Query &left, const Query &right):
                        BinaryQuery(left, right, "&") { }
    // concrete class: AndQuery inherits rep and defines the remaining pure virtual
    QueryResult eval(const TextQuery&) const;
};
inline Query operator&(const Query &lhs, const Query &rhs)
{
        return  std::shared_ptr<Query_base>(new  AndQuery(lhs,
rhs));
}

class OrQuery: public BinaryQuery {
    friend Query operator|(const Query&, const Query&);
    OrQuery(const Query &left, const Query &right):
                BinaryQuery(left, right, "|") { }
    QueryResult eval(const TextQuery&) const;
};
inline Query operator|(const Query &lhs, const Query &rhs)
{
        return  std::shared_ptr<Query_base>(new  OrQuery(lhs,
rhs));
}
```

These classes make the respective operator a friend and define a constructor to create their `BinaryQuery` base part with the appropriate operator. They inherit the `BinaryQuery` definition of `rep`, but each overrides the `eval` function.

Like the ~ operator, the `&` and `|` operators return a `shared_ptr` bound to a newly allocated object of the corresponding type. That `shared_ptr` gets converted to `Query` as part of the return statement in each of these operators.

Exercises Section 15.9.3

Exercise 15.34: For the expression built in Figure 15.3 (p. 638):

(a) List the constructors executed in processing that expression.

(b) List the calls to `rep` that are made from `cout << q`.

(c) List the calls to `eval` made from `q.eval()`.

Exercise 15.35: Implement the `Query` and `Query_base` classes, including a definition of `rep` but omitting the definition of `eval`.

Exercise 15.36: Put print statements in the constructors and `rep` members and run your code to check your answers to `(a)` and `(b)` from the first exercise.

Exercise 15.37: What changes would your classes need if the derived classes had members of type `shared_ptr<Query_base>` rather than of type `Query`?

Exercise 15.38: Are the following declarations legal? If not, why not? If so, explain what the declarations mean.

Click here to view code image

```
BinaryQuery a = Query("fiery") & Query("bird");
AndQuery b = Query("fiery") & Query("bird");
OrQuery c = Query("fiery") & Query("bird");
```

15.9.4. The eval Functions

The `eval` functions are the heart of our query system. Each of these functions calls `eval` on its operand(s) and then applies its own logic: The `OrQuery eval` operation returns the union of the results of its two operands; `AndQuery` returns the intersection. The `NotQuery` is more complicated: It must return the line numbers that are not in its operand's set.

To support the processing in the `eval` functions, we need to use the version of `QueryResult` that defines the members we added in the exercises to §12.3.2 (p. 490). We'll assume that `QueryResult` has `begin` and `end` members that will let us iterate through the `set` of line numbers that the `QueryResult` holds. We'll also assume that `QueryResult` has a member named `get_file` that returns a `shared_ptr` to the underlying file on which the query was executed.

Warning

Our `Query` classes use members defined for `QueryResult` in the exercises to §12.3.2 (p. 490).

OrQuery::eval

An `OrQuery` represents the union of the results for its two operands, which we obtain by calling `eval` on each of its operands. Because these operands are `Query` objects, calling `eval` is a call to `Query::eval`, which in turn makes a virtual call to `eval` on the underlying `Query_base` object. Each of these calls yields a `QueryResult` representing the line numbers in which its operand appears. We'll combine those line numbers into a new `set`:

Click here to view code image

```
// returns the union of its operands' result sets
QueryResult
OrQuery::eval(const TextQuery& text) const
```

```
{
    // virtual calls through the Query members, lhs and rhs
    // the calls to eval return the QueryResult for each operand
    auto right = rhs.eval(text), left = lhs.eval(text);
    // copy the line numbers from the left-hand operand into the result set
    auto ret_lines =
         make_shared<set<line_no>>(left.begin(), left.end());
    // insert lines from the right-hand operand
    ret_lines->insert(right.begin(), right.end());
    // return the new QueryResult representing the union of lhs and rhs
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

We initialize `ret_lines` using the `set` constructor that takes a pair of iterators. The `begin` and `end` members of a `QueryResult` return iterators into that object's `set` of line numbers. So, `ret_lines` is created by copying the elements from `left`'s `set`. We next call `insert` on `ret_lines` to insert the elements from `right`. After this call, `ret_lines` contains the line numbers that appear in either `left` or `right`.

The `eval` function ends by building and returning a `QueryResult` representing the combined match. The `QueryResult` constructor (§12.3.2, p. 489) takes three arguments: a `string` representing the query, a `shared_ptr` to the `set` of matching line numbers, and a `shared_ptr` to the `vector` that represents the input file. We call `rep` to generate the `string` and `get_file` to obtain the `shared_ptr` to the file. Because both `left` and `right` refer to the same file, it doesn't matter which of these we use for `get_file`.

AndQuery::eval

The `AndQuery` version of `eval` is similar to the `OrQuery` version, except that it calls a library algorithm to find the lines in common to both queries:

Click here to view code image

```
// returns the intersection of its operands' result sets
QueryResult
AndQuery::eval(const TextQuery& text) const
{
    // virtual calls through the Query operands to get result sets for the operands
    auto left = lhs.eval(text), right = rhs.eval(text);
    // set to hold the intersection of left and right
    auto ret_lines = make_shared<set<line_no>>();
    // writes the intersection of two ranges to a destination iterator
    // destination iterator in this call adds elements to ret
    set_intersection(left.begin(), left.end(),
                     right.begin(), right.end(),
                          inserter(*ret_lines, ret_lines-
>begin()));
    return QueryResult(rep(), ret_lines, left.get_file());
}
```

Here we use the library `set_intersection` algorithm, which is described in Appendix A.2.8 (p. 880), to merge these two `set`s.

The `set_intersection` algorithm takes five iterators. It uses the first four to denote two input sequences (§10.5.2, p. 413). Its last argument denotes a destination. The algorithm writes the elements that appear in both input sequences into the destination.

In this call we pass an insert iterator (§10.4.1, p. 401) as the destination. When `set_intersection` writes to this iterator, the effect will be to insert a new element into `ret_lines`.

Like the `OrQuery` `eval` function, this one ends by building and returning a `QueryResult` representing the combined match.

NotQuery::eval

`NotQuery` finds each line of the text within which the operand is not found:

Click here to view code image

```
// returns the lines not in its operand's result set
QueryResult
NotQuery::eval(const TextQuery& text) const
{
    // virtual call to eval through the Query operand
    auto result = query.eval(text);
    // start out with an empty result set
    auto ret_lines = make_shared<set<line_no>>();
    // we have to iterate through the lines on which our operand appears
    auto beg = result.begin(), end = result.end();
    // for each line in the input file, if that line is not in result,
    // add that line number to ret_lines
    auto sz = result.get_file()->size();
    for (size_t n = 0; n != sz; ++n) {
        // if we haven't processed all the lines in result
        // check whether this line is present
        if (beg == end || *beg != n)
            ret_lines->insert(n);  // if not in result, add this line
        else if (beg != end)
            ++beg; // otherwise get the next line number in result if there is
one
    }
    return QueryResult(rep(), ret_lines, result.get_file());
}
```

As in the other `eval` functions, we start by calling `eval` on this object's operand. That call returns the `QueryResult` containing the line numbers on which the operand appears, but we want the line numbers on which the operand does not appear. That is, we want every line in the file that is not already in `result`.

We generate that `set` by iterating through sequenital integers up to the size of the input file. We'll put each number that is not in `result` into `ret_lines`. We position `beg` and `end` to denote the first and one past the last elements in `result`. That object is a `set`, so when we iterate through it, we'll obtain the line numbers in ascending order.

The loop body checks whether the current number is in `result`. If not, we add that number to `ret_lines`. If the number is in `result`, we increment `beg`, which is our iterator into `result`.

Once we've processed all the line numbers, we return a `QueryResult` containing `ret_lines`, along with the results of running `rep` and `get_file` as in the previous `eval` functions.

Exercises Section 15.9.4

Exercise 15.39: Implement the `Query` and `Query_base` classes. Test your application by evaluating and printing a query such as the one in Figure 15.3 (p. 638).

Exercise 15.40: In the `OrQuery eval` function what would happen if its `rhs` member returned an empty set? What if its `lhs` member did so? What if both `rhs` and `lhs` returned empty sets?

Exercise 15.41: Reimplement your classes to use built-in pointers to `Query_base` rather than `shared_ptrs`. Remember that your classes will no longer be able to use the synthesized copy-control members.

Exercise 15.42: Design and implement one of the following enhancements:

(a) Print words only once per sentence rather than once per line.

(b) Introduce a history system in which the user can refer to a previous query by number, possibly adding to it or combining it with another.

(c) Allow the user to limit the results so that only matches in a given range of lines are displayed.

Chapter Summary

Inheritance lets us write new classes that share behavior with their base class(es) but override or add to that behavior as needed. Dynamic binding lets us ignore type differences by choosing, at run time, which version of a function to run based on an object's dynamic type. The combination of inheritance and dynamic binding lets us write type-independent, programs that have type-specific behavior.

In `C++`, dynamic binding applies *only* to functions declared as `virtual` and called through a reference or pointer.

A derived-class object contains a subobject corresponding to each of its base classes. Because every derived object contains a base part, we can convert a reference or pointer to a derived-class type to a reference or pointer to an accessible base class.

Inherited objects are constructed, copied, moved, and assigned by constructing, copying, moving, and assigning the base part(s) of the object before handling the derived part. Destructors execute in the opposite order; the derived type is destroyed first, followed by destructors for the base-class subobjects. Base classes usually should define a virtual destructor even if the class otherwise has no need for a destructor. The destructor must be virtual if a pointer to a base is ever deleted when it actually addresses a derived-class object.

A derived class specifies a protection level for each of its base class(es). Members of a `public` base are part of the interface of the derived class; members of a `private` base are inaccessible; members of a `protected` base are accessible to classes that derive from the derived class but not to users of the derived class.

Defined Terms

abstract base class Class that has one or more pure virtual functions. We cannot create objects of an abstract base-class type.

accessible Base class member that can be used through a derived object. Accessibility depends on the access specifier used in derivation list of the derived class and the access level of the member in the base class. For example, a `public` member of a class that is inherited via `public` inheritance is accessible to users of the derived class. A `public` base class member is inacceessible if the inheritance is `private`.

base class Class from which other classes inherit. The members of the base class become members of the derived class.

class derivation list List of base classes, each of which may have an optional access level, from which a derived class inherits. If no access specifier is provided, the inheritance is `public` if the derived class is defined with the `struct` keyword, and is `private` if the class is defined with the `class` keyword.

derived class Class that inherits from another class. A derived class can override the virtuals of its base and can define new members. A derived-class scope is nested in the scope of its base class(es); members of the derived class can use members of the base class directly.

derived-to-base conversion Implicit conversion of a derived object to a reference to a base class, or of a pointer to a derived object to a pointer to the base type.

direct base class Base class from which a derived class inherits directly. Direct base classes are specified in the derivation list of the derived class. A direct base class may itself be a derived class.

dynamic binding Delaying until run time the selection of which function to run. In C++, dynamic binding refers to the runtime choice of which virtual function to run based on the underlying type of the object to which a reference or pointer is bound.

dynamic type Type of an object at run time. The dynamic type of an object to which a reference refers or to which a pointer points may differ from the static type of the reference or pointer. A pointer or reference to a base-class type can refer to an to object of derived type. In such cases the static type is reference (or pointer) to base, but the dynamic type is reference (or pointer) to derived.

indirect base class Base class that does not appear in the derivation list of a derived class. A class from which the direct base class inherits, directly or indirectly, is an indirect base class to the derived class.

inheritance Programming technique for defining a new class (known as a derived class) in terms of an existing class (known as the base class). The derived class inherits the members of the base class.

object-oriented programming Method of writing programs using data abstraction, inheritance, and dynamic binding.

override Virtual function defined in a derived class that has the same parameter list as a virtual in a base class overrides the base-class definition.

polymorphism As used in object-oriented programming, refers to the ability to obtain type-specific behavior based on the dynamic type of a reference or pointer.

private inheritance In `private` inheritance, the `public` and `protected` members of the base class are `private` members of the derived.

protected access specifier Members defined after the `protected` keyword may be accessed by the members and friends of a derived class. However, these members are only accessible through derived objects. `protected` members are not accessible to ordinary users of the class.

protected inheritance In `protected` inheritance, the `protected` and `public` members of the base class are `protected` members of the derived class.

public inheritance The `public` interface of the base class is part of the `public` interface of the derived class.

pure virtual Virtual function declared in the class header using `= 0` just before the semicolon. A pure virtual function need not be (but may be) defined. Classes with pure virtuals are abstract classes. If a derived class does not define its own version of an inherited pure virtual, then the derived class is abstract as well.

refactoring Redesigning programs to collect related parts into a single abstraction, replacing the original code with uses of the new abstraction. Typically, classes are refactored to move data or function members to the highest common point in the hierarchy to avoid code duplication.

run-time binding See dynamic binding.

sliced down What happens when an object of derived type is used to initialize or assign an object of the base type. The derived portion of the object is “sliced down,” leaving only the base portion, which is assigned to the base.

static type Type with which a variable is defined or that an expression yields. Static type is known at compile time.

virtual function Member function that defines type-specific behavior. Calls to a virtual made through a reference or pointer are resolved at run time, based on the type of the object to which the reference or pointer is bound.

Chapter 16. Templates and Generic Programming

Contents

Both object-oriented programming (OOP) and generic programming deal with types that are not known at the time the program is written. The distinction between the two is that OOP deals with types that are not known until run time, whereas in generic programming the types become known during compilation.

The containers, iterators, and algorithms described in Part II are all examples of generic programming. When we write a generic program, we write the code in a way that is independent of any particular type. When we use a generic program, we supply the type(s) or value(s) on which that instance of the program will operate.

For example, the library provides a single, generic definition of each container, such as `vector`. We can use that generic definition to define many different types of `vector`s, each of which differs from the others as to the type of elements the `vector` contains.

Templates are the foundation of generic programming. We can use and have used templates without understanding how they are defined. In this chapter we'll see how to define our own templates.

Templates are the foundation for generic programming in C++. A template is a blueprint or formula for creating classes or functions. When we use a generic type, such as `vector`, or a generic function, such as `find`, we supply the information needed to transform that blueprint into a specific class or function. That transformation happens during compilation. In Chapter 3 and Part II we learned how to use templates. In this chapter we'll learn how to define them.

16.1. Defining a Template

Imagine that we want to write a function to compare two values and indicate whether the first is less than, equal to, or greater than the second. In practice, we'd want to define several such functions, each of which will compare values of a given type. Our first attempt might be to define several overloaded functions:

Click here to view code image

```
// returns 0 if the values are equal, -1 if v1 is smaller, 1 if v2 is smaller
int compare(const string &v1, const string &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
int compare(const double &v1, const double &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

These functions are nearly identical: The only difference between them is the type of their parameters. The function body is the same in each function.

Having to repeat the body of the function for each type that we compare is tedious and error-prone. More importantly, we need to know when we write the program all the types that we might ever want to `compare`. This strategy cannot work if we want to be able to use the function on types that our users might supply.

16.1.1. Function Templates

Rather than defining a new function for each type, we can define a **function template**. A function template is a formula from which we can generate type-specific versions of that function. The template version of `compare` looks like

Click here to view code image

```
template <typename T>
int compare(const T &v1, const T &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

A template definition starts with the keyword `template` followed by a **template parameter list**, which is a comma-separated list of one or more **template parameters** bracketed by the less-than (<) and greater-than (>) tokens.

Note

In a template definition, the template parameter list cannot be empty.

The template parameter list acts much like a function parameter list. A function parameter list defines local variable(s) of a specified type but does not say how to initialize them. At run time, arguments are supplied that initialize the parameters.

Analogously, template parameters represent types or values used in the definition of a class or function. When we use a template, we specify—either implicitly or explicitly—**template argument(s)** to bind to the template parameter(s).

Our `compare` function declares one type parameter named `T`. Inside `compare`, we use the name `T` to refer to a type. Which *actual type* `T` represents is determined at compile time based on how `compare` is used.

Instantiating a Function Template

When we call a function template, the compiler (ordinarily) uses the arguments of the call to deduce the template argument(s) for us. That is, when we call `compare`, the compiler uses the type of the arguments to determine what type to bind to the template parameter `T`. For example, in this call

Click here to view code image

```
cout << compare(1, 0) << endl;       // T is int
```

the arguments have type `int`. The compiler will deduce `int` as the template argument and will bind that argument to the template parameter `T`.

The compiler uses the deduced template parameter(s) to **instantiate** a specific version of the function for us. When the compiler instantiates a template, it creates a new "instance" of the template using the actual template argument(s) in place of the corresponding template parameter(s). For example, given the calls

Click here to view code image

```
// instantiates int compare(const int&, const int&)
cout << compare(1, 0) << endl;       // T is int
// instantiates int compare(const vector<int>&, const vector<int>&)
vector<int> vec1{1, 2, 3}, vec2{4, 5, 6};
cout << compare(vec1, vec2) << endl; // T is vector<int>
```

the compiler will instantiate two different versions of `compare`. For the first call, the compiler will write and compile a version of `compare` with `T` replaced by `int`:

Click here to view code image

```
int compare(const int &v1, const int &v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

For the second call, it will generate a version of `compare` with `T` replaced by `vector<int>`. These compiler-generated functions are generally referred to as an **instantiation** of the template.

Template Type Parameters

Our `compare` function has one template **type parameter**. In general, we can use a type parameter as a type specifier in the same way that we use a built-in or class type specifier. In particular, a type parameter can be used to name the return type or a function parameter type, and for variable declarations or casts inside the function body:

Click here to view code image

```
// ok: same type used for the return type and parameter
template <typename T> T foo(T* p)
{
    T tmp = *p; // tmp will have the type to which p points
    // ...
    return tmp;
}
```

Each type parameter must be preceded by the keyword `class` or `typename`:

Click here to view code image

```
// error: must precede U with either typename or class
template <typename T, U> T calc(const T&, const U&);
```

These keywords have the same meaning and can be used interchangeably inside a template parameter list. A template parameter list can use both keywords:

Click here to view code image

```
// ok: no distinction between typename and class in a template parameter list
template <typename T, class U> calc (const T&, const U&);
```

It may seem more intuitive to use the keyword `typename` rather than `class` to designate a template type parameter. After all, we can use built-in (nonclass) types as a template type argument. Moreover, `typename` more clearly indicates that the name that follows is a type name. However, `typename` was added to C++ after templates were already in widespread use; some programmers continue to use `class` exclusively.

Nontype Template Parameters

In addition to defining type parameters, we can define templates that take **nontype parameters**. A nontype parameter represents a value rather than a type. Nontype parameters are specified by using a specific type name instead of the `class` or `typename` keyword.

When the template is instantiated, nontype parameters are replaced with a value supplied by the user or deduced by the compiler. These values must be constant expressions (§ 2.4.4, p. 65), which allows the compiler to instantiate the templates during compile time.

As an example, we can write a version of `compare` that will handle string literals. Such literals are arrays of `const char`. Because we cannot copy an array, we'll define our parameters as references to an array (§ 6.2.4, p. 217). Because we'd like to be able to compare literals of different lengths, we'll give our template two nontype parameters. The first template parameter will represent the size of the first array, and the second parameter will represent the size of the second array:

Click here to view code image

```
template<unsigned N, unsigned M>
int compare(const char (&p1)[N], const char (&p2)[M])
{
    return strcmp(p1, p2);
}
```

When we call this version of `compare`:

```
compare("hi", "mom")
```

the compiler will use the size of the literals to instantiate a version of the template with the sizes substituted for `N` and `M`. Remembering that the compiler inserts a null terminator at the end of a string literal (§ 2.1.3, p. 39), the compiler will instantiate

Click here to view code image

```
int compare(const char (&p1)[3], const char (&p2)[4])
```

A nontype parameter may be an integral type, or a pointer or (lvalue) reference to an object or to a function type. An argument bound to a nontype integral parameter must be a constant expression. Arguments bound to a pointer or reference nontype parameter must have static lifetime (Chapter 12, p. 450). We may not use an ordinary (`nonstatic`) local object or a dynamic object as a template argument for reference or pointer nontype template parameters. A pointer parameter can also be instantiated by `nullptr` or a zero-valued constant expression.

A template nontype parameter is a constant value inside the template definition. A nontype parameter can be used when constant expressions are required, for example, to specify the size of an array.

Note

Template arguments used for nontype template parameters must be constant expressions.

`inline` and `constexpr` Function Templates

A function template can be declared `inline` or `constexpr` in the same ways as nontemplate functions. The `inline` or `constexpr` specifier follows the template parameter list and precedes the return type:

Click here to view code image

```
// ok: inline specifier follows the template parameter list
template <typename T> inline T min(const T&, const T&);
// error: incorrect placement of the inline specifier
inline template <typename T> T min(const T&, const T&);
```

Writing Type-Independent Code

Simple though it is, our initial `compare` function illustrates two important principles for writing generic code:

- The function parameters in the template are references to `const`.

• The tests in the body use only < comparisons.

By making the function parameters references to `const`, we ensure that our function can be used on types that cannot be copied. Most types—including the built-in types and, except for `unique_ptr` and the IO types, all the library types we've used—do allow copying. However, there can be class types that do not allow copying. By making our parameters references to `const`, we ensure that such types can be used with our `compare` function. Moreover, if `compare` is called with large objects, then this design will also make the function run faster.

You might think it would be more natural for the comparisons to be done using both the < and > operators:

```
// expected comparison
if (v1 < v2) return -1;
if (v1 > v2) return 1;
return 0;
```

However, by writing the code using only the < operator, we reduce the requirements on types that can be used with our `compare` function. Those types must support <, but they need not also support >.

In fact, if we were truly concerned about type independence and portability, we probably should have defined our function using the `less` (§ 14.8.2, p. 575):

Click here to view code image

```
// version of compare that will be correct even if used on pointers; see § 14.8.2 (p. 575)
template <typename T> int compare(const T &v1, const T &v2)
{
    if (less<T>()(v1, v2)) return -1;
    if (less<T>()(v2, v1)) return 1;
    return 0;
}
```

The problem with our original version is that if a user calls it with two pointers and those pointers do not point to the same array, then our code is undefined.

Best Practices

Template programs should try to minimize the number of requirements placed on the argument types.

Template Compilation

When the compiler sees the definition of a template, it does not generate code. It generates code only when we instantiate a specific instance of the template. The fact that code is generated only when we use a template (and not when we define it) affects how we organize our source code and when errors are detected.

Ordinarily, when we call a function, the compiler needs to see only a declaration for the function. Similarly, when we use objects of class type, the class definition must be available, but the definitions of the member functions need not be present. As a result, we put class definitions and function declarations in header files and definitions of ordinary and class-member functions in source files.

Templates are different: To generate an instantiation, the compiler needs to have the code that defines a function template or class template member function. As a result, unlike nontemplate code, headers for templates typically include definitions as well as declarations

Note

Definitions of function templates and member functions of class templates are ordinarily put into header files.

Key Concept: Templates and Headers

Templates contain two kinds of names:

- Those that do not depend on a template parameter
- Those that do depend on a template parameter

It is up to the provider of a template to ensure that all names that do not depend on a template parameter are visible when the template is used. Moreover, the template provider must ensure that the definition of the template, including the definitions of the members of a class template, are visible when the template is instantiated.

It is up to *users* of a template to ensure that declarations for all functions, types, and operators associated with the types used to instantiate the template are visible.

Both of these requirements are easily satisfied by well-structured programs that make appropriate use of headers. Authors of templates should provide a header that contains the template definition along with declarations for all the names used in the class template or in the definitions of its members. Users of the template must include the header for the template and for any types used to instantiate that template.

Compilation Errors Are Mostly Reported during Instantiation

The fact that code is not generated until a template is instantiated affects when we learn about compilation errors in the code inside the template. In general, there are three stages during which the compiler might flag an error.

The first stage is when we compile the template itself. The compiler generally can’t find many errors at this stage. The compiler can detect syntax errors—such as forgetting a semicolon or misspelling a variable name—but not much else.

The second error-detection time is when the compiler sees a use of the template. At this stage, there is still not much the compiler can check. For a call to a function template, the compiler typically will check that the number of the arguments is appropriate. It can also detect whether two arguments that are supposed to have the same type do so. For a class template, the compiler can check that the right number of template arguments are provided but not much more.

The third time when errors are detected is during instantiation. It is only then that type-related errors can be found. Depending on how the compiler manages instantiation, these errors may be reported at link time.

When we write a template, the code may not be overtly type specific, but template code usually makes some assumptions about the types that will be used. For example, the code inside our original `compare` function:

Click here to view code image

```
if (v1 < v2) return -1;   // requires < on objects of type T
if (v2 < v1) return 1;    // requires < on objects of type T
return 0;                 // returns int; not dependent on T
```

assumes that the argument type has a < operator. When the compiler processes the body of this template, it cannot verify whether the conditions in the `if` statements are legal. If the arguments passed to `compare` have a < operation, then the code is fine, but not otherwise. For example,

Click here to view code image

```
Sales_data data1, data2;
cout << compare(data1, data2) << endl; // error: no < on
Sales_data
```

This call instantiates a version of `compare` with `T` replaced by `Sales_data`. The `if` conditions attempt to use < on `Sales_data` objects, but there is no such operator. This instantiation generates a version of the function that will not compile. However, errors such as this one cannot be detected until the compiler instantiates the definition of `compare` on type `Sales_data`.

Warning

It is up to the caller to guarantee that the arguments passed to the template support any operations that template uses, and that those operations behave correctly in the context in which the template uses them.

Exercises Section 16.1.1

Exercise 16.1: Define instantiation.

Exercise 16.2: Write and test your own versions of the `compare` functions.

Exercise 16.3: Call your `compare` function on two `Sales_data` objects to see how your compiler handles errors during instantiation.

Exercise 16.4: Write a template that acts like the library `find` algorithm. The function will need two template type parameters, one to represent the function's iterator parameters and the other for the type of the value. Use your function to find a given value in a `vector<int>` and in a `list<string>`.

Exercise 16.5: Write a template version of the `print` function from § 6.2.4 (p. 217) that takes a reference to an array and can handle arrays of any size and any element type.

Exercise 16.6: How do you think the library `begin` and `end` functions that take an array argument work? Define your own versions of these functions.

Exercise 16.7: Write a `constexpr` template that returns the size of a given array.

Exercise 16.8: In the "Key Concept" box on page 108, we noted that as a matter of habit C++ programmers prefer using `!=` to using `<`. Explain the rationale for this habit.

16.1.2. Class Templates

A **class template** is a blueprint for generating classes. Class templates differ from function templates in that the compiler cannot deduce the template parameter type(s) for a class template. Instead, as we've seen many times, to use a class template we must supply additional information inside angle brackets following the template's name (§ 3.3, p. 97). That extra information is the list of template arguments to use in place of the template parameters.

Defining a Class Template

As an example, we'll implement a template version of `StrBlob` (§ 12.1.1, p. 456). We'll name our template `Blob` to indicate that it is no longer specific to `string`s.

Like `StrBlob`, our template will provide shared (and checked) access to the elements it holds. Unlike that class, our template can be used on elements of pretty much any type. As with the library containers, our users will have to specify the element type when they use a `Blob`.

Like function templates, class templates begin with the keyword `template` followed by a template parameter list. In the definition of the class template (and its members), we use the template parameters as stand-ins for types or values that will be supplied when the template is used:

Click here to view code image

```
template <typename T> class Blob {
public:
    typedef T value_type;
    typedef typename std::vector<T>::size_type size_type;
    // constructors
    Blob();
    Blob(std::initializer_list<T> il);
    // number of elements in the Blob
    size_type size() const { return data->size(); }
    bool empty() const { return data->empty(); }
    // add and remove elements
    void push_back(const T &t) {data->push_back(t);}
    // move version; see § 13.6.3 (p. 548)
    void push_back(T &&t) { data->push_back(std::move(t)); }
    void pop_back();
    // element access
    T& back();
    T& operator[](size_type i); // defined in § 14.5 (p. 566)
private:
    std::shared_ptr<std::vector<T>> data;
    // throws msg if data[i] isn't valid
    void check(size_type i, const std::string &msg) const;
};
```

Our `Blob` template has one template type parameter, named `T`. We use the type parameter anywhere we refer to the element type that the `Blob` holds. For example, we define the return type of the operations that provide access to the elements in the `Blob` as `T&`. When a user instantiates a `Blob`, these uses of `T` will be replaced by the specified template argument type.

With the exception of the template parameter list, and the use of `T` instead of `string`, this class is the same as the version we defined in § 12.1.1 (p. 456) and updated in § 12.1.6 (p. 475) and in Chapters 13 and 14.

Instantiating a Class Template

As we’ve seen many times, when we use a class template, we must supply extra information. We can now see that that extra information is a list of **explicit template arguments** that are bound to the template's parameters. The compiler uses these template arguments to instantiate a specific class from the template.

For example, to define a type from our `Blob` template, we must provide the element type:

Click here to view code image

```
Blob<int> ia;                // empty Blob<int>
Blob<int> ia2 = {0,1,2,3,4}; // Blob<int> with five elements
```

Both `ia` and `ia2` use the same type-specific version of `Blob` (i.e., `Blob<int>`). From these definitions, the compiler will instantiate a class that is equivalent to

Click here to view code image

```
template <> class Blob<int> {
    typedef typename std::vector<int>::size_type size_type;
    Blob();
    Blob(std::initializer_list<int> il);
    // ...
    int& operator[](size_type i);
private:
    std::shared_ptr<std::vector<int>> data;
    void check(size_type i, const std::string &msg) const;
};
```

When the compiler instantiates a class from our `Blob` template, it rewrites the `Blob` template, replacing each instance of the template parameter `T` by the given template argument, which in this case is `int`.

The compiler generates a different class for each element type we specify:

Click here to view code image

```
// these definitions instantiate two distinct Blob types
Blob<string> names; // Blob that holds strings
Blob<double> prices;// different element type
```

These definitions would trigger instantiations of two distinct classes: The definition of `names` creates a `Blob` class in which each occurrence of `T` is replaced by `string`. The definition of `prices` generates a `Blob` with `T` replaced by `double`.

Note

Each instantiation of a class template constitutes an independent class. The type `Blob<string>` has no relationship to, or any special access to, the members of any other `Blob` type.

References to a Template Type in the Scope of the Template

In order to read template class code, it can be helpful to remember that the name of a class template is not the name of a type (§ 3.3, p. 97). A class template is used to instantiate a type, and an instantiated type always includes template argument(s).

What can be confusing is that code in a class template generally doesn't use the name of an actual type (or value) as a template argument. Instead, we often use the template's own parameter(s) as the template argument(s). For example, our `data` member uses two templates, `vector` and `shared_ptr`. Whenever we use a template, we must supply template arguments. In this case, the template argument we supply is the same type that is used to instantiate the `Blob`. Therefore, the definition of `data`

Click here to view code image

```
    std::shared_ptr<std::vector<T>> data;
```

uses `Blob`'s type parameter to say that `data` is the instantiation of `shared_ptr` that points to the instantiation of `vector` that holds objects of type `T`. When we instantiate a particular kind of `Blob`, such as `Blob<string>`, then `data` will be

Click here to view code image

```
    shared_ptr<vector<string>>
```

If we instantiate `Blob<int>`, then `data` will be `shared_ptr<vector<int>>`, and so on.

Member Functions of Class Templates

As with any class, we can define the member functions of a class template either inside or outside of the class body. As with any other class, members defined inside the class body are implicitly inline.

A class template member function is itself an ordinary function. However, each instantiation of the class template has its own version of each member. As a result, a member function of a class template has the same template parameters as the class itself. Therefore, a member function defined outside the class template body starts with the keyword `template` followed by the class' template parameter list.

As usual, when we define a member outside its class, we must say to which class the member belongs. Also as usual, the name of a class generated from a template includes its template arguments. When we define a member, the template argument(s) are the same as the template parameter(s). That is, for a given member function of `StrBlob` that was defined as

Click here to view code image

```
ret-type StrBlob::member-name(parm-list)
```

the corresponding `Blob` member will look like

Click here to view code image

```
template <typename T>
ret-type Blob<T>::member-name(parm-list)
```

The `check` and Element Access Members

We'll start by defining the `check` member, which verifies a given index:

Click here to view code image

```
template <typename T>
void  Blob<T>::check(size_type  i,  const  std::string  &msg)
const
{
    if (i >= data->size())
        throw std::out_of_range(msg);
}
```

Aside from the differences in the class name and the use of the template parameter list, this function is identical to the original `StrBlob` member.

The subscript operator and `back` function use the template parameter to specify the return type but are otherwise unchanged:

Click here to view code image

```
template <typename T>
T& Blob<T>::back()
{
    check(0, "back on empty Blob");
    return data->back();
}
template <typename T>
T& Blob<T>::operator[](size_type i)
{
    // if i is too big, check will throw, preventing access to a nonexistent element
    check(i, "subscript out of range");
    return (*data)[i];
}
```

In our original `StrBlob` class these operators returned `string&`. The template versions will return a reference to whatever type is used to instantiate `Blob`.

The `pop_back` function is nearly identical to our original `StrBlob` member:

Click here to view code image

```
template <typename T> void Blob<T>::pop_back()
```

```
{
    check(0, "pop_back on empty Blob");
    data->pop_back();
}
```

The subscript operator and `back` members are overloaded on `const`. We leave the definition of these members, and of the `front` members, as an exercise.

`Blob` Constructors

As with any other member defined outside a class template, a constructor starts by declaring the template parameters for the class template of which it is a member:

Click here to view code image

```
template <typename T>
Blob<T>::Blob(): data(std::make_shared<std::vector<T>>()) { }
```

Here we are defining the member named `Blob` in the scope of `Blob<T>`. Like our `StrBlob` default constructor (§ 12.1.1, p. 456), this constructor allocates an empty `vector` and stores the pointer to that `vector` in `data`. As we've seen, we use the class' own type parameter as the template argument of the `vector` we allocate.

Similarly, the constructor that takes an `initializer_list` uses its type parameter `T` as the element type for its `initializer_list` parameter:

Click here to view code image

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il):
              data(std::make_shared<std::vector<T>>(il)) { }
```

Like the default constructor, this constructor allocates a new `vector`. In this case, we initialize that `vector` from the parameter, `il`.

To use this constructor, we must pass an `initializer_list` in which the elements are compatible with the element type of the `Blob`:

Click here to view code image

```
Blob<string> articles = {"a", "an", "the"};
```

The parameter in this constructor has type `initializer_list<string>`. Each string literal in the list is implicitly converted to `string`.

Instantiation of Class-Template Member Functions

By default, a member function of a class template is instantiated *only* if the program uses that member function. For example, this code

Click here to view code image

```
//
```

```
    instantiates  Blob<int>  and the  initializer_list<int>  constructor
Blob<int> squares = {0,1,2,3,4,5,6,7,8,9};
//  instantiates  Blob<int>::size() const
for (size_t i = 0; i != squares.size(); ++i)
    squares[i] = i*i; //  instantiates  Blob<int>::operator[](size_t)
```

instantiates the `Blob<int>` class and three of its member functions: `operator[]`, `size`, and the `initializer_list<int>` constructor.

If a member function isn't used, it is not instantiated. The fact that members are instantiated only if we use them lets us instantiate a class with a type that may not meet the requirements for some of the template's operations (§ 9.2, p. 329).

Note

By default, a member of an instantiated class template is instantiated only if the member is used.

Simplifying Use of a Template Class Name inside Class Code

There is one exception to the rule that we must supply template arguments when we use a class template type. Inside the scope of the class template itself, we may use the name of the template without arguments:

Click here to view code image

```
//  BlobPtr  throws an exception on attempts to access a nonexistent element
template <typename T> class BlobPtr
public:
    BlobPtr(): curr(0) { }
    BlobPtr(Blob<T> &a, size_t sz = 0):
            wptr(a.data), curr(sz) { }
    T& operator*() const
    { auto p = check(curr, "dereference past end");
      return (*p)[curr];  //  (*p)  is the  vector  to which this object points
    }
    //  increment and decrement
    BlobPtr& operator++();          //  prefix operators
    BlobPtr& operator--();
private:
    //  check  returns a  shared_ptr  to the  vector  if the check succeeds
    std::shared_ptr<std::vector<T>>
        check(std::size_t, const std::string&) const;
    //  store a  weak_ptr ,  which means the underlying  vector  might be destroyed
    std::weak_ptr<std::vector<T>> wptr;
    std::size_t curr;       //  current position within the array
};
```

Careful readers will have noted that the prefix increment and decrement members of `BlobPtr` return `BlobPtr&`, not `BlobPtr<T>&`. When we are inside the scope of a class template, the compiler treats references to the template itself as if we had supplied template arguments matching the template's own parameters. That is, it is as if we had written:

```
BlobPtr<T>& operator++();
BlobPtr<T>& operator--();
```

Using a Class Template Name outside the Class Template Body

When we define members outside the body of a class template, we must remember that we are not in the scope of the class until the class name is seen (§ 7.4, p. 282):

Click here to view code image

```
// postfix: increment/decrement the object but return the unchanged value
template <typename T>
BlobPtr<T> BlobPtr<T>::operator++(int)
{
    // no check needed here; the call to prefix increment will do the check
    BlobPtr ret = *this;   // save the current value
    ++*this;     // advance one element; prefix ++ checks the increment
    return ret;  // return the saved state
}
```

Because the return type appears outside the scope of the class, we must specify that the return type returns a `BlobPtr` instantiated with the same type as the class. Inside the function body, we are in the scope of the class so do not need to repeat the template argument when we define `ret`. When we do not supply template arguments, the compiler assumes that we are using the same type as the member's instantiation. Hence, the definition of `ret` is as if we had written:

```
BlobPtr<T> ret = *this;
```

Note

Inside the scope of a class template, we may refer to the template without specifying template argument(s).

Class Templates and Friends

When a class contains a friend declaration (§ 7.2.1, p. 269), the class and the friend can independently be templates or not. A class template that has a nontemplate friend grants that friend access to all the instantiations of the template. When the friend is itself a template, the class granting friendship controls whether friendship includes all instantiations of the template or only specific instantiation(s).

One-to-One Friendship

The most common form of friendship from a class template to another template (class or function) establishes friendship between corresponding instantiations of the class and its friend. For example, our `Blob` class should declare the `BlobPtr` class and a template version of the `Blob` equality operator (originally defined for `StrBlob` in the exercises in § 14.3.1 (p. 562)) as friends.

In order to refer to a specific instantiation of a template (class or function) we must first declare the template itself. A template declaration includes the template's template parameter list:

Click here to view code image

```
// forward declarations needed for friend declarations in Blob
template <typename> class BlobPtr;
template <typename> class Blob; // needed for parameters in operator==
template <typename T>
    bool operator==(const Blob<T>&, const Blob<T>&);
template <typename T> class Blob {
    // each instantiation of Blob grants access to the version of
    // BlobPtr and the equality operator instantiated with the same type
    friend class BlobPtr<T>;
    friend bool operator==<T>
           (const Blob<T>&, const Blob<T>&);
    // other members as in § 12.1.1 (p. 456)
};
```

We start by declaring that `Blob`, `BlobPtr`, and `operator==` are templates. These declarations are needed for the parameter declaration in the `operator==` function and the friend declarations in `Blob`.

The friend declarations use `Blob`'s template parameter as their own template argument. Thus, the friendship is restricted to those instantiations of `BlobPtr` and the equality operator that are instantiated with the same type:

Click here to view code image

```
Blob<char> ca; // BlobPtr<char> and operator==<char> are friends
Blob<int> ia;  // BlobPtr<int> and operator==<int> are friends
```

The members of `BlobPtr<char>` may access the nonpublic parts of `ca` (or any other `Blob<char>` object), but `ca` has no special access to `ia` (or any other `Blob<int>`) or to any other instantiation of `Blob`.

General and Specific Template Friendship

A class can also make every instantiation of another template its friend, or it may limit friendship to a specific instantiation:

Click here to view code image

```
// forward declaration necessary to befriend a specific instantiation of a template
template <typename T> class Pal;
class C {  //  C is an ordinary, nontemplate class
     friend class Pal<C>;  // Pal instantiated with class C is a friend to C
     // all instances of Pal2 are friends to C;
     // no forward declaration required when we befriend all instantiations
     template <typename T> friend class Pal2;
};
template <typename T> class C2 { // C2 is itself a class template
     // each instantiation of C2 has the same instance of Pal as a friend
     friend class Pal<T>;  // a template declaration for Pal must be in scope
     // all instances of Pal2 are friends of each instance of C2, prior declaration needed
     template <typename X> friend class Pal2;
     // Pal3 is a nontemplate class that is a friend of every instance of C2
     friend class Pal3;     // prior declaration for Pal3 not needed
};
```

To allow all instantiations as friends, the friend declaration must use template parameter(s) that differ from those used by the class itself.

Befriending the Template's Own Type Parameter

C++ 11

Under the new standard, we can make a template type parameter a friend:

Click here to view code image

```
template <typename Type> class Bar {
friend Type; // grants access to the type used to instantiate Bar
     //  ...
};
```

Here we say that whatever type is used to instantiate `Bar` is a friend. Thus, for some type named `Foo`, `Foo` would be a friend of `Bar<Foo>`, `Sales_data` a friend of `Bar<Sales_data>`, and so on.

It is worth noting that even though a friend ordinarily must be a class or a function, it is okay for `Bar` to be instantiated with a built-in type. Such friendship is allowed so that we can instantiate classes such as `Bar` with built-in types.

Template Type Aliases

An instantiation of a class template defines a class type, and as with any other class type, we can define a `typedef` (§ 2.5.1, p. 67) that refers to that instantiated class:

```
    typedef Blob<string> StrBlob;
```

This `typedef` will let us run the code we wrote in § 12.1.1 (p. 456) using our template version of `Blob` instantiated with `string`. Because a template is not a type, we cannot define a `typedef` that refers to a template. That is, there is no way to define a `typedef` that refers to `Blob<T>`.

C++ 11

However, the new standard lets us define a type alias for a class template:

Click here to view code image

```
    template<typename T> using twin = pair<T, T>;
    twin<string> authors; // authors is a pair<string, string>
```

Here we've defined `twin` as a synonym for `pair`s in which the members have the same type. Users of `twin` need to specify that type only once.

A template type alias is a synonym for a family of classes:

Click here to view code image

```
    twin<int> win_loss;   // win_loss is a pair<int, int>
    twin<double> area;    // area is a pair<double, double>
```

Just as we do when we use a class template, when we use `twin`, we specify which particular kind of `twin` we want.

When we define a template type alias, we can fix one or more of the template parameters:

Click here to view code image

```
    template <typename T> using partNo = pair<T, unsigned>;
    partNo<string> books;   // books is a pair<string, unsigned>
    partNo<Vehicle> cars;   // cars is a pair<Vehicle, unsigned>
    partNo<Student> kids;   // kids is a pair<Student, unsigned>
```

Here we've defined `partNo` as a synonym for the family of types that are `pair`s in which the `second` member is an `unsigned`. Users of `partNo` specify a type for the `first` member of the `pair` but have no choice about `second`.

static Members of Class Templates

Like any other class, a class template can declare `static` members (§ 7.6, p. 300):

Click here to view code image

```
template <typename T> class Foo {
public:
   static std::size_t count() { return ctr; }
   // other interface members
private:
   static std::size_t ctr;
   // other implementation members
};
```

Here `Foo` is a class template that has a `public static` member function named `count` and a `private static` data member named `ctr`. Each instantiation of `Foo` has its own instance of the `static` members. That is, for any given type `X`, there is one `Foo<X>::ctr` and one `Foo<X>::count` member. All objects of type `Foo<X>` share the same `ctr` object and `count` function. For example,

Click here to view code image

```
// instantiates static members Foo<string>::ctr and Foo<string>::count
Foo<string> fs;
// all three objects share the same Foo<int>::ctr and Foo<int>::count members
Foo<int> fi, fi2, fi3;
```

As with any other `static` data member, there must be exactly one definition of each `static` data member of a template class. However, there is a distinct object for each instantiation of a class template. As a result, we define a `static` data member as a template similarly to how we define the member functions of that template:

Click here to view code image

```
template <typename T>
size_t Foo<T>::ctr = 0; // define and initialize ctr
```

As with any other member of a class template, we start by defining the template parameter list, followed by the type of the member we are defining and the member's name. As usual, a member's name includes the member's class name, which for a class generated from a template includes its template arguments. Thus, when `Foo` is instantiated for a particular template argument type, a separate `ctr` will be instantiated for that class type and initialized to `0`.

As with static members of nontemplate classes, we can access a `static` member of a class template through an object of the class type or by using the scope operator to access the member directly. Of course, to use a `static` member through the class, we must refer to a specific instantiation:

Click here to view code image

```
Foo<int> fi;                 // instantiates Foo<int> class
                              // and the static data member ctr
auto ct = Foo<int>::count(); // instantiates Foo<int>::count
```

```
ct = fi.count();              // uses  Foo<int>::count
ct = Foo::count();            // error: which template instantiation?
```

Like any other member function, a `static` member function is instantiated only if it is used in a program.

Exercises Section 16.1.2

Exercise 16.9: What is a function template? What is a class template?

Exercise 16.10: What happens when a class template is instantiated?

Exercise 16.11: The following definition of `List` is incorrect. How would you fix it?

Click here to view code image

```
template <typename elemType> class ListItem;
template <typename elemType> class List {
public:
    List<elemType>();
    List<elemType>(const List<elemType> &);
    List<elemType>& operator=(const List<elemType> &);
    ~List();
    void insert(ListItem *ptr, elemType value);
private:
    ListItem *front, *end;
};
```

Exercise 16.12: Write your own version of the `Blob` and `BlobPtr` templates. including the various `const` members that were not shown in the text.

Exercise 16.13: Explain which kind of friendship you chose for the equality and relational operators for `BlobPtr`.

Exercise 16.14: Write a `Screen` class template that uses nontype parameters to define the height and width of the `Screen`.

Exercise 16.15: Implement input and output operators for your `Screen` template. Which, if any, friends are necessary in class `Screen` to make the input and output operators work? Explain why each friend declaration, if any, was needed.

Exercise 16.16: Rewrite the `StrVec` class (§ 13.5, p. 526) as a template named `Vec`.

16.1.3. Template Parameters

Like the names of function parameters, a template parameter name has no intrinsic meaning. We ordinarily name type parameters T, but we can use any name:

Click here to view code image

```
template <typename Foo> Foo calc(const Foo& a, const Foo& b)
{
    Foo tmp = a; // tmp has the same type as the parameters and return type
    // ...
    return tmp;  // return type and parameters have the same type
}
```

Template Parameters and Scope

Template parameters follow normal scoping rules. The name of a template parameter can be used after it has been declared and until the end of the template declaration or definition. As with any other name, a template parameter hides any declaration of that name in an outer scope. Unlike most other contexts, however, a name used as a template parameter may not be reused within the template:

Click here to view code image

```
typedef double A;
template <typename A, typename B> void f(A a, B b)
{
    A tmp = a; // tmp has same type as the template parameter A, not double
    double B;  // error: redeclares template parameter B
}
```

Normal name hiding says that the typedef of A is hidden by the type parameter named A. Thus, tmp is not a double; it has whatever type gets bound to the template parameter A when calc is used. Because we cannot reuse names of template parameters, the declaration of the variable named B is an error.

Because a parameter name cannot be reused, the name of a template parameter can appear only once with in a given template parameter list:

Click here to view code image

```
// error: illegal reuse of template parameter name V
template <typename V, typename V> // ...
```

Template Declarations

A template declaration must include the template parameters :

Click here to view code image

```
// declares but does not define compare and Blob
template <typename T> int compare(const T&, const T&);
template <typename T> class Blob;
```

As with function parameters, the names of a template parameter need not be the same across the declaration(s) and the definition of the same template:

Click here to view code image

```
// all three uses of calc refer to the same function template
template <typename T> T calc(const T&, const T&); // declaration
template <typename U> U calc(const U&, const U&); // declaration
// definition of the template
template <typename Type>
Type calc(const Type& a, const Type& b) { /* ... */ }
```

Of course, every declaration and the definition of a given template must have the same number and kind (i.e., type or nontype) of parameters.

Best Practices

For reasons we'll explain in § 16.3 (p. 698), declarations for all the templates needed by a given file usually should appear together at the beginning of a file before any code that uses those names.

Using Class Members That Are Types

Recall that we use the scope operator (`::`) to access both `static` members and type members (§ 7.4, p. 282, and § 7.6, p. 301). In ordinary (nontemplate) code, the compiler has access to the class defintion. As a result, it knows whether a name accessed through the scope operator is a type or a `static` member. For example, when we write `string::size_type`, the compiler has the definition of `string` and can see that `size_type` is a type.

Assuming `T` is a template type parameter, When the compiler sees code such as `T::mem` it won't know until instantiation time whether `mem` is a type or a `static` data member. However, in order to process the template, the compiler must know whether a name represents a type. For example, assuming `T` is the name of a type parameter, when the compiler sees a statement of the following form:

```
T::size_type * p;
```

it needs to know whether we're defining a variable named `p` or are multiplying a `static` data member named `size_type` by a variable named `p`.

By default, the language assumes that a name accessed through the scope operator is not a type. As a result, if we want to use a type member of a template type parameter, we must explicitly tell the compiler that the name is a type. We do so by using the keyword `typename`:

Click here to view code image

```
template <typename T>
typename T::value_type top(const T& c)
{
    if (!c.empty())
        return c.back();
    else
        return typename T::value_type();
}
```

Our `top` function expects a container as its argument and uses `typename` to specify its return type and to generate a value initialized element (§ 7.5.3, p. 293) to return if `c` has no elements.

Note

When we want to inform the compiler that a name represents a type, we must use the keyword `typename`, not `class`.

Default Template Arguments

Just as we can supply default arguments to function parameters (§ 6.5.1, p. 236), we can also supply **default template arguments**. Under the new standard, we can supply default arguments for both function and class templates. Earlier versions of the language, allowed default arguments only with class templates.

C++ 11

As an example, we'll rewrite `compare` to use the library `less` function-object template (§ 14.8.2, p. 574) by default:

Click here to view code image

```
// compare has a default template argument, less<T>
// and a default function argument, F()
template <typename T, typename F = less<T>>
int compare(const T &v1, const T &v2, F f = F())
{
    if (f(v1, v2)) return -1;
    if (f(v2, v1)) return 1;
    return 0;
}
```

Here we've given our template a second type parameter, named `F`, that represents the type of a callable object (§ 10.3.2, p. 388) and defined a new function parameter, `f`, that will be bound to a callable object.

We've also provided defaults for this template parameter and its corresponding function parameter. The default template argument specifies that `compare` will use the library `less` function-object class, instantiated with the same type parameter as `compare`. The default function argument says that `f` will be a default-initialized object of type `F`.

When users call this version of `compare`, they may supply their own comparison operation but are not required to do so:

Click here to view code image

```
bool i = compare(0, 42); // uses less; i is -1
// result depends on the isbns in item1 and item2
Sales_data item1(cin), item2(cin);
bool j = compare(item1, item2, compareIsbn);
```

The first call uses the default function argument, which is a default-initialized object of type `less<T>`. In this call, `T` is `int` so that object has type `less<int>`. This instantiation of `compare` will use `less<int>` to do its comparisons.

In the second call, we pass `compareIsbn` (§ 11.2.2, p. 425) and two objects of type `Sales_data`. When `compare` is called with three arguments, the type of the third argument must be a callable object that returns a type that is convertible to `bool` and takes arguments of a type compatible with the types of the first two arguments. As usual, the types of the template parameters are deduced from their corresponding function arguments. In this call, the type of `T` is deduced as `Sales_data` and `F` is deduced as the type of `compareIsbn`.

As with function default arguments, a template parameter may have a default argument only if all of the parameters to its right also have default arguments.

Template Default Arguments and Class Templates

Whenever we use a class template, we must always follow the template's name with brackets. The brackets indicate that a class must be instantiated from a template. In particular, if a class template provides default arguments for all of its template parameters, and we want to use those defaults, we must put an empty bracket pair following the template's name:

Click here to view code image

```
template <class T = int> class Numbers {   // by default T is int
public:
    Numbers(T v = 0): val(v) { }
    // various operations on numbers
private:
    T val;
};
Numbers<long double> lots_of_precision;
Numbers<> average_precision; // empty <> says we want the default type
```

Here we instantiate two versions of `Numbers`: `average_precision` instantiates `Numbers` with `T` replaced by `int`; `lots_of_precision` instantiates `Numbers` with

T replaced by `long double`.

Exercises Section 16.1.3

Exercise 16.17: What, if any, are the differences between a type parameter that is declared as a `typename` and one that is declared as a `class`? When must `typename` be used?

Exercise 16.18: Explain each of the following function template declarations and identify whether any are illegal. Correct each error that you find.

Click here to view code image

```
(a) template <typename T, U, typename V> void f1(T, U, V);
(b) template <typename T> T f2(int &T);
(c) inline template <typename T> T foo(T, unsigned int*);
(d) template <typename T> f4(T, T);
(e) typedef char Ctype;
     template <typename Ctype> Ctype f5(Ctype a);
```

Exercise 16.19: Write a function that takes a reference to a container and prints the elements in that container. Use the container's `size_type` and `size` members to control the loop that prints the elements.

Exercise 16.20: Rewrite the function from the previous exercise to use iterators returned from `begin` and `end` to control the loop.

16.1.4. Member Templates

A class—either an ordinary class or a class template—may have a member function that is itself a template. Such members are referred to as **member templates**. Member templates may not be virtual.

Member Templates of Ordianary (Nontemplate) Classes

As an example of an ordinary class that has a member template, we'll define a class that is similar to the default deleter type used by `unique_ptr` (§ 12.1.5, p. 471). Like the default deleter, our class will have an overloaded function-call operator (§ 14.8, p. 571) that will take a pointer and execute `delete` on the given pointer. Unlike the default deleter, our class will also print a message whenever the deleter is executed. Because we want to use our deleter with any type, we'll make the call operator a template:

Click here to view code image

```
// function-object class that calls delete on a given pointer
```

```
class DebugDelete {
public:
    DebugDelete(std::ostream &s = std::cerr): os(s) { }
    // as with any function template, the type of T is deduced by the compiler
    template <typename T> void operator()(T *p) const
       { os << "deleting unique_ptr" << std::endl; delete p;
}
private:
    std::ostream &os;
};
```

Like any other template, a member template starts with its own template parameter list. Each `DebugDelete` object has an `ostream` member on which to write, and a member function that is itself a template. We can use this class as a replacement for `delete`:

Click here to view code image

```
double* p = new double;
DebugDelete d;     // an object that can act like a delete expression
d(p); // calls DebugDelete::operator()(double*), which deletes p
int* ip = new int;
// calls operator()(int*) on a temporary DebugDelete object
DebugDelete()(ip);
```

Because calling a `DebugDelete` object `delete`s its given pointer, we can also use `DebugDelete` as the deleter of a `unique_ptr`. To override the deleter of a `unique_ptr`, we supply the type of the deleter inside brackets and supply an object of the deleter type to the constructor (§ 12.1.5, p. 471):

Click here to view code image

```
// destroying the the object to which p points
// instantiates DebugDelete::operator()<int>(int *)
unique_ptr<int, DebugDelete> p(new int, DebugDelete());
// destroying the the object to which sp points
// instantiates DebugDelete::operator()<string>(string*)
unique_ptr<string,      DebugDelete>      sp(new      string,
DebugDelete());
```

Here, we’ve said that p’s deleter will have type `DebugDelete`, and we have supplied an unnamed object of that type in p’s constructor.

The `unique_ptr` destructor calls the `DebugDelete`’s call operator. Thus, whenever `unique_ptr`’s destructor is instantiated, `DebugDelete`’s call operator will also be instantiated: Thus, the definitions above will instantiate:

Click here to view code image

```
// sample instantiations for member templates of DebugDelete
void DebugDelete::operator()(int *p) const { delete p; }
void DebugDelete::operator()(string *p) const { delete p; }
```

Member Templates of Class Templates

We can also define a member template of a class template. In this case, both the class and the member have their own, independent, template parameters.

As an example, we'll give our `Blob` class a constructor that will take two iterators denoting a range of elements to copy. Because we'd like to support iterators into varying kinds of sequences, we'll make this constructor a template:

Click here to view code image

```
template <typename T> class Blob {
    template <typename It> Blob(It b, It e);
    // ...
};
```

This constructor has its own template type parameter, `It`, which it uses for the type of its two function parameters.

Unlike ordinary function members of class templates, member templates *are* function templates. When we define a member template outside the body of a class template, we must provide the template parameter list for the class template and for the function template. The parameter list for the class template comes first, followed by the member's own template parameter list:

Click here to view code image

```
template <typename T>     // type parameter for the class
template <typename It>    // type parameter for the constructor
    Blob<T>::Blob(It b, It e):
              data(std::make_shared<std::vector<T>>(b, e)) {
}
```

Here we are defining a member of a class template that has one template type parameter, which we have named `T`. The member itself is a function template that has a type parameter named `It`.

Instantiation and Member Templates

To instantiate a member template of a class template, we must supply arguments for the template parameters for both the class and the function templates. As usual, argument(s) for the class template parameter(s) are determined by the type of the object through which we call the member template. Also as usual, the compiler typically deduces template argument(s) for the member template's own parameter(s) from the arguments passed in the call (§ 16.1.1, p. 653):

Click here to view code image

```
int ia[] = {0,1,2,3,4,5,6,7,8,9};
```

```
vector<long> vi = {0,1,2,3,4,5,6,7,8,9};
list<const char*> w = {"now", "is", "the", "time"};
// instantiates the Blob<int> class
// and the Blob<int> constructor that has two int* parameters
Blob<int> a1(begin(ia), end(ia));
// instantiates the Blob<int> constructor that has
// two vector<long>::iterator parameters
Blob<int> a2(vi.begin(), vi.end());
// instantiates the Blob<string> class and the Blob<string>
// constructor that has two (list<const char*>::iterator parameters
Blob<string> a3(w.begin(), w.end());
```

When we define `a1`, we explicitly specify that the compiler should instantiate a version of `Blob` with the template parameter bound to `int`. The type parameter for the constructor's own parameters will be deduced from the type of `begin(ia)` and `end(ia)`. That type is `int*`. Thus, the definition of `a1` instantiates:

```
Blob<int>::Blob(int*, int*);
```

The definition of `a2` uses the already instantiated `Blob<int>` class, and instantiates the constructor with `It` replaced by `vector<short>::iterator`. The definition of `a3` (explicitly) instantiates the `Blob` with its template parameter bound to `string` and (implicitly) instantiates the member template constructor of that class with its parameter bound to `list<const char*>`.

Exercises Section 16.1.4

Exercise 16.21: Write your own version of `DebugDelete`.

Exercise 16.22: Revise your `TextQuery` programs from § 12.3 (p. 484) so that the `shared_ptr` members use a `DebugDelete` as their deleter (§ 12.1.4, p. 468).

Exercise 16.23: Predict when the call operator will be executed in your main query program. If your expectations and what happens differ, be sure you understand why.

Exercise 16.24: Add a constructor that takes two iterators to your `Blob` template.

16.1.5. Controlling Instantiations

The fact that instantiations are generated when a template is used (§ 16.1.1, p. 656) means that the same instantiation may appear in multiple object files. When two or more separately compiled source files use the same template with the same template arguments, there is an instantiation of that template in each of those files.

In large systems, the overhead of instantiating the same template in multiple files can become significant. Under the new standard, we can avoid this overhead through an **explicit instantiation**. An explicit instantiation has the form

Click here to view code image

```
extern template declaration; // instantiation declaration
template declaration;        // instantiation definition
```

where *declaration* is a class or function declaration in which all the template parameters are replaced by the template arguments. For example,

Click here to view code image

```
// instantion declaration and definition
extern template class Blob<string>;             // declaration
template int compare(const int&, const int&);   // definition
```

When the compiler sees an `extern` template declaration, it will not generate code for that instantiation in that file. Declaring an instantiation as `extern` is a promise that there will be a non`extern` use of that instantiation elsewhere in the program. There may be several `extern` declarations for a given instantiation but there must be exactly one definition for that instantiation.

Because the compiler automatically instantiates a template when we use it, the `extern` declaration must appear before any code that uses that instantiation:

Click here to view code image

```
// Application.cc
// these template types must be instantiated elsewhere in the program
extern template class Blob<string>;
extern template int compare(const int&, const int&);
Blob<string> sa1, sa2; // instantiation will appear elsewhere
// Blob<int> and its initializer_list constructor instantiated in this file
Blob<int> a1 = {0,1,2,3,4,5,6,7,8,9};
Blob<int> a2(a1);  // copy constructor instantiated in this file
int i = compare(a1[0], a2[0]); // instantiation will appear elsewhere
```

The file `Application.o` will contain instantiations for `Blob<int>`, along with the `initializer_list` and copy constructors for that class. The `compare<int>` function and `Blob<string>` class will not be instantiated in that file. There must be definitions of these templates in some other file in the program:

Click here to view code image

```
// templateBuild.cc
// instantiation file must provide a (nonextern) definition for every
// type and function that other files declare as extern
```

```
template int compare(const int&, const int&);
template class Blob<string>; // instantiates all members of the class template
```

When the compiler sees an instantiation definition (as opposed to a declaration), it generates code. Thus, the file `templateBuild.o` will contain the definitions for `compare` instantiated with `int` and for the `Blob<string>` class. When we build the application, we must link `templateBuild.o` with the `Application.o` files.

Warning

There must be an explicit instantiation definition somewhere in the program for every instantiation declaration.

Instantiation Definitions Instantiate All Members

An instantiation definition for a class template instantiates *all* the members of that template including inline member functions. When the compiler sees an instantiation definition it cannot know which member functions the program uses. Hence, unlike the way it handles ordinary class template instantiations, the compiler instantiates *all* the members of that class. Even if we do not use a member, that member will be instantiated. Consequently, we can use explicit instantiation only for types that can be used with all the members of that template.

Note

An instantiation definition can be used only for types that can be used with every member function of a class template.

16.1.6. Efficiency and Flexibility

The library smart pointer types (§ 12.1, p. 450) offer a good illustration of design choices faced by designers of templates.

The obvious difference between `shared_ptr` and `unique_ptr` is the strategy they use in managing the pointer they hold—one class gives us shared ownership; the other owns the pointer that it holds. This difference is essential to what these classes do.

These classes also differ in how they let users override their default deleter. We can easily override the deleter of a `shared_ptr` by passing a callable object when we create or `reset` the pointer. In contrast, the type of the deleter is part of the type of a `unique_ptr` object. Users must supply that type as an explicit template argument when they define a `unique_ptr`. As a result, it is more complicated for users of `unique_ptr` to provide their own deleter.

Exercises Section 16.1.5

Exercise 16.25: Explain the meaning of these declarations:

Click here to view code image

```
extern template class vector<string>;
template class vector<Sales_data>;
```

Exercise 16.26: Assuming `NoDefault` is a class that does not have a default constructor, can we explicitly instantiate `vector<NoDefault>`? If not, why not?

Exercise 16.27: For each labeled statement explain what, if any, instantiations happen. If a template is instantiated, explain why; if not, explain why not.

Click here to view code image

```
template <typename T> class Stack { };
void f1(Stack<char>);                   // (a)
class Exercise {
    Stack<double> &rsd;                 // (b)
    Stack<int>    si;                   // (c)
};
int main() {
    Stack<char> *sc;                    // (d)
    f1(*sc);                            // (e)
    int iObj = sizeof(Stack< string >); // (f)
}
```

The difference in how the deleter is handled is incidental to the functionality of these classes. However, as we'll see, this difference in implementation strategy may have important performance impacts.

Binding the Deleter at Run Time

Although we don't know how the library types are implemented, we can infer that `shared_ptr` must access its deleter indirectly. That is the deleter must be stored as a pointer or as a class (such as `function` (§ 14.8.3, p. 577)) that encapsulates a pointer.

We can be certain that `shared_ptr` does not hold the deleter as a direct member, because the type of the deleter isn't known until run time. Indeed, we can change the type of the deleter in a given `shared_ptr` during that `shared_ptr`'s lifetime. We can construct a `shared_ptr` using a deleter of one type, and subsequently use `reset` to give that same `shared_ptr` a different type of deleter. In general, we cannot have a member whose type changes at run time. Hence, the deleter must be stored indirectly.

To think about how the deleter must work, let's assume that `shared_ptr` stores the pointer it manages in a member named `p`, and that the deleter is accessed through a member named `del`. The `shared_ptr` destructor must include a statement such as

Click here to view code image

```
// value of del known only at run time; call through a pointer
del ? del(p) : delete p; // del(p) requires run-time jump to del's location
```

Because the deleter is stored indirectly, the call `del(p)` requires a run-time jump to the location stored in `del` to execute the code to which `del` points.

Binding the Deleter at Compile Time

Now, let's think about how `unique_ptr` might work. In this class, the type of the deleter is part of the type of the `unique_ptr`. That is, `unique_ptr` has two template parameters, one that represents the pointer that the `unique_ptr` manages and the other that represents the type of the deleter. Because the type of the deleter is part of the type of a `unique_ptr`, the type of the deleter member is known at compile time. The deleter can be stored directly in each `unique_ptr` object.

The `unique_ptr` destructor operates similarly to its `shared_ptr` counterpart in that it calls a user-supplied deleter or executes `delete` on its stored pointer:

Click here to view code image

```
// del bound at compile time; direct call to the deleter is instantiated
del(p);   // no run-time overhead
```

The type of `del` is either the default deleter type or a user-supplied type. It doesn't matter; either way the code that will be executed is known at compile time. Indeed, if the deleter is something like our `DebugDelete` class (§ 16.1.4, p. 672) this call might even be inlined at compile time.

By binding the deleter at compile time, `unique_ptr` avoids the run-time cost of an indirect call to its deleter. By binding the deleter at run time, `shared_ptr` makes it easier for users to override the deleter.

Exercises Section 16.1.6

Exercise 16.28: Write your own versions of `shared_ptr` and `unique_ptr`.

Exercise 16.29: Revise your `Blob` class to use your version of `shared_ptr` rather than the library version.

Exercise 16.30: Rerun some of your programs to verify your `shared_ptr` and revised `Blob` classes. (Note: Implementing the `weak_ptr` type is beyond the scope of this Primer, so you will not be able to use the `BlobPtr` class with your revised `Blob`.)

Exercise 16.31: Explain how the compiler might inline the call to the deleter if we used `DebugDelete` with `unique_ptr`.

16.2. Template Argument Deduction

We've seen that, by default, the compiler uses the arguments in a call to determine the template parameters for a function template. The process of determining the template arguments from the function arguments is known as **template argument deduction**. During template argument deduction, the compiler uses types of the arguments in the call to find the template arguments that generate a version of the function that best matches the given call.

16.2.1. Conversions and Template Type Parameters

As with a nontemplate function, the arguments we pass in a call to a function template are used to initialize that function's parameters. Function parameters whose type uses a template type parameter have special initialization rules. Only a very limited number of conversions are automatically applied to such arguments. Rather than converting the arguments, the compiler generates a new instantiation.

As usual, top-level `const`s (§ 2.4.3, p. 63) in either the parameter or the argument are ignored. The only other conversions performed in a call to a function template are

- `const` conversions: A function parameter that is a reference (or pointer) to a `const` can be passed a reference (or pointer) to a non`const` object (§ 4.11.2, p. 162).
- Array- or function-to-pointer conversions: If the function parameter is not a reference type, then the normal pointer conversion will be applied to arguments of array or function type. An array argument will be converted to a pointer to its first element. Similarly, a function argument will be converted to a pointer to the function's type (§ 4.11.2, p. 161).

Other conversions, such as the arithmetic conversions (§ 4.11.1, p. 159), derived-to-base (§ 15.2.2, p. 597), and user-defined conversions (§ 7.5.4, p. 294, and § 14.9, p. 579), are not performed.

As examples, consider calls to the functions `fobj` and `fref`. The `fobj` function copies its parameters, whereas `fref`'s parameters are references:

Click here to view code image

```
template <typename T> T fobj(T, T); // arguments are copied
template <typename T> T fref(const T&, const T&); // references
string s1("a value");
const string s2("another value");
fobj(s1, s2); // calls fobj(string, string); const is ignored
fref(s1, s2); // calls fref(const string&, const string&)
              // uses premissible conversion to const on s1
int a[10], b[42];
fobj(a, b); // calls f(int*, int*)
fref(a, b); // error: array types don't match
```

In the first pair of calls, we pass a `string` and a `const string`. Even though these types do not match exactly, both calls are legal. In the call to `fobj`, the arguments are copied, so whether the original object is `const` doesn't matter. In the call to `fref`, the parameter type is a reference to `const`. Conversion to `const` for a reference parameter is a permitted conversion, so this call is legal.

In the next pair of calls, we pass array arguments in which the arrays are different sizes and hence have different types. In the call to `fobj`, the fact that the array types differ doesn't matter. Both arrays are converted to pointers. The template parameter type in `fobj` is `int*`. The call to `fref`, however, is illegal. When the parameter is a reference, the arrays are not converted to pointers (§ 6.2.4, p. 217). The types of a and b don't match, so the call is in error.

Note

`const` conversions and array or function to pointer are the only automatic conversions for arguments to parameters with template types.

Function Parameters That Use the Same Template Parameter Type

A template type parameter can be used as the type of more than one function parameter. Because there are limited conversions, the arguments to such parameters must have essentially the same type. If the deduced types do not match, then the call is an error. For example, our `compare` function (§ 16.1.1, p. 652) takes two `const T&` parameters. Its arguments must have essentially the same type:

Click here to view code image

```
long lng;
```

```
compare(lng, 1024); // error: cannot instantiate compare(long, int)
```

This call is in error because the arguments to `compare` don't have the same type. The template argument deduced from the first argument is `long`; the one for the second is `int`. These types don't match, so template argument deduction fails.

If we want to allow normal conversions on the arguments, we can define the function with two type parameters:

Click here to view code image

```
// argument types can differ but must be compatible
template <typename A, typename B>
int flexibleCompare(const A& v1, const B& v2)
{
    if (v1 < v2) return -1;
    if (v2 < v1) return 1;
    return 0;
}
```

Now the user may supply arguments of different types:

Click here to view code image

```
long lng;
flexibleCompare(lng, 1024); // ok: calls flexibleCompare(long, int)
```

Of course, a < operator must exist that can compare values of those types.

Normal Conversions Apply for Ordinary Arguments

A function template can have parameters that are defined using ordinary types—that is, types that do not involve a template type parameter. Such arguments have no special processing; they are converted as usual to the corresponding type of the parameter (§ 6.1, p. 203). For example, consider the following template:

Click here to view code image

```
template <typename T> ostream &print(ostream &os, const T
&obj)
{
    return os << obj;
}
```

The first function parameter has a known type, `ostream&`. The second parameter, `obj`, has a template parameter type. Because the type of `os` is fixed, normal conversions are applied to arguments passed to `os` when `print` is called:

Click here to view code image

```
print(cout, 42); // instantiates print(ostream&, int)
ofstream f("output");
print(f, 10);    // uses print(ostream&, int); converts f to ostream&
```

In the first call, the type of the first argument exactly matches the type of the first parameter. This call will cause a version of `print` that takes an `ostream&` and an `int` to be instantiated. In the second call, the first argument is an `ofstream` and there is a conversion from `ofstream` to `ostream&` (§ 8.2.1, p. 317). Because the type of this parameter does not depend on a template parameter, the compiler will implicitly convert `f` to `ostream&`.

Note

Normal conversions are applied to arguments whose type is not a template parameter.

Exercises Section 16.2.1

Exercise 16.32: What happens during template argument deduction?

Exercise 16.33: Name two type conversions allowed on function arguments involved in template argument deduction.

Exercise 16.34: Given only the following code, explain whether each of these calls is legal. If so, what is the type of `T`? If not, why not?

Click here to view code image

```
template <class T> int compare(const T&, const T&);
```

(a) `compare("hi", "world");`

(b) `compare("bye", "dad");`

Exercise 16.35: Which, if any, of the following calls are errors? If the call is legal, what is the type of `T`? If the call is not legal, what is the problem?

Click here to view code image

```
template <typename T> T calc(T, int);
template <typename T> T fcn(T, T);
double d;    float f;    char c;
```

(a) `calc(c, 'c');`

(b) `calc(d, f);`

(c) `fcn(c, 'c');`

(d) `fcn(d, f);`

Exercise 16.36: What happens in the following calls:

Click here to view code image

```
template <typename T> f1(T, T);
template <typename T1, typename T2) f2(T1, T2);
```

```
int i = 0, j = 42, *p1 = &i, *p2 = &j;
const int *cp1 = &i, *cp2 = &j;
```

(a) `f1(p1, p2);`

(b) `f2(p1, p2);`

(c) `f1(cp1, cp2);`

(d) `f2(cp1, cp2);`

(e) `f1(p1, cp1);`

(f) `f2(p1, cp1);`

16.2.2. Function-Template Explicit Arguments

In some situations, it is not possible for the compiler to deduce the types of the template arguments. In others, we want to allow the user to control the template instantiation. Both cases arise most often when a function return type differs from any of those used in the parameter list.

Specifying an Explicit Template Argument

As an example in which we want to let the user specify which type to use, we'll define a function template named `sum` that takes arguments of two different types. We'd like to let the user specify the type of the result. That way the user can choose whatever precision is appropriate.

We can let the user control the type of the return by defining a third template parameter to represent the return type:

Click here to view code image

```
// T1 cannot be deduced: it doesn't appear in the function parameter list
template <typename T1, typename T2, typename T3>
T1 sum(T2, T3);
```

In this case, there is no argument whose type can be used to deduce the type of `T1`. The caller must provide an **explicit template argument** for this parameter on each call to `sum`.

We supply an explicit template argument to a call the same way that we define an instance of a class template. Explicit template arguments are specified inside angle brackets after the function name and before the argument list:

Click here to view code image

```
// T1 is explicitly specified; T2 and T3 are inferred from the argument types
```

```
auto val3 = sum<long long>(i, lng);  // long long sum(int, long)
```

This call explicitly specifies the type for `T1`. The compiler will deduce the types for `T2` and `T3` from the types of `i` and `lng`.

Explicit template argument(s) are matched to corresponding template parameter(s) from left to right; the first template argument is matched to the first template parameter, the second argument to the second parameter, and so on. An explicit template argument may be omitted only for the trailing (right-most) parameters, and then only if these can be deduced from the function parameters. If our `sum` function had been written as

Click here to view code image

```
// poor design: users must explicitly specify all three template parameters
template <typename T1, typename T2, typename T3>
T3 alternative_sum(T2, T1);
```

then we would always have to specify arguments for all three parameters:

Click here to view code image

```
// error: can't infer initial template parameters
auto val3 = alternative_sum<long long>(i, lng);
// ok: all three parameters are explicitly specified
auto val2 = alternative_sum<long long, int, long>(i, lng);
```

Normal Conversions Apply for Explicitly Specified Arguments

For the same reasons that normal conversions are permitted for parameters that are defined using ordinary types (§ 16.2.1, p. 680), normal conversions also apply for arguments whose template type parameter is explicitly specified:

Click here to view code image

```
long lng;
compare(lng, 1024);        // error: template parameters don't match
compare<long>(lng, 1024);  // ok: instantiates compare(long, long)
compare<int>(lng, 1024);   // ok: instantiates compare(int, int)
```

As we've seen, the first call is in error because the arguments to `compare` must have the same type. If we explicitly specify the template parameter type, normal conversions apply. Thus, the call to `compare<long>` is equivalent to calling a function taking two `const long&` parameters. The `int` parameter is automatically converted to `long`. In the second call, `T` is explicitly specified as `int`, so `lng` is converted to `int`.

Exercises Section 16.2.2

Exercise 16.37: The library `max` function has two function parameters and returns the larger of its arguments. This function has one template type parameter. Could you call `max` passing it an `int` and a `double`? If so, how? If not, why not?

Exercise 16.38: When we call `make_shared` (§ 12.1.1, p. 451), we have to provide an explicit template argument. Explain why that argument is needed and how it is used.

Exercise 16.39: Use an explicit template argument to make it sensible to pass two string literals to the original version of `compare` from § 16.1.1 (p. 652).

16.2.3. Trailing Return Types and Type Transformation

Using an explicit template argument to represent a template function's return type works well when we want to let the user determine the return type. In other cases, requiring an explicit template argument imposes a burden on the user with no compensating advantage. For example, we might want to write a function that takes a pair of iterators denoting a sequence and returns a reference to an element in the sequence:

Click here to view code image

```
template <typename It>
??? &fcn(It beg, It end)
{
    // process the range
    return *beg;  // return a reference to an element from the range
}
```

We don't know the exact type we want to return, but we do know that we want that type to be a reference to the element type of the sequence we're processing:

Click here to view code image

```
vector<int> vi = {1,2,3,4,5};
Blob<string> ca = { "hi", "bye" };
auto &i = fcn(vi.begin(), vi.end()); // fcn should return int&
auto &s = fcn(ca.begin(), ca.end()); // fcn should return string&
```

Here, we know that our function will return `*beg`, and we know that we can use `decltype(*beg)` to obtain the type of that expression. However, `beg` doesn't exist until the parameter list has been seen. To define this function, we must use a trailing return type (§ 6.3.3, p. 229). Because a trailing return appears after the parameter list, it can use the function's parameters:

Click here to view code image

```
// a trailing return lets us declare the return type after the parameter list is seen
template <typename It>
auto fcn(It beg, It end) -> decltype(*beg)
{
    // process the range
    return *beg;   // return a reference to an element from the range
}
```

Here we've told the compiler that `fcn`'s return type is the same as the type returned by dereferencing its `beg` parameter. The dereference operator returns an lvalue (§ 4.1.1, p. 136), so the type deduced by `decltype` is a reference to the type of the element that `beg` denotes. Thus, if `fcn` is called on a sequence of `string`s, the return type will be `string&`. If the sequence is `int`, the return will be `int&`.

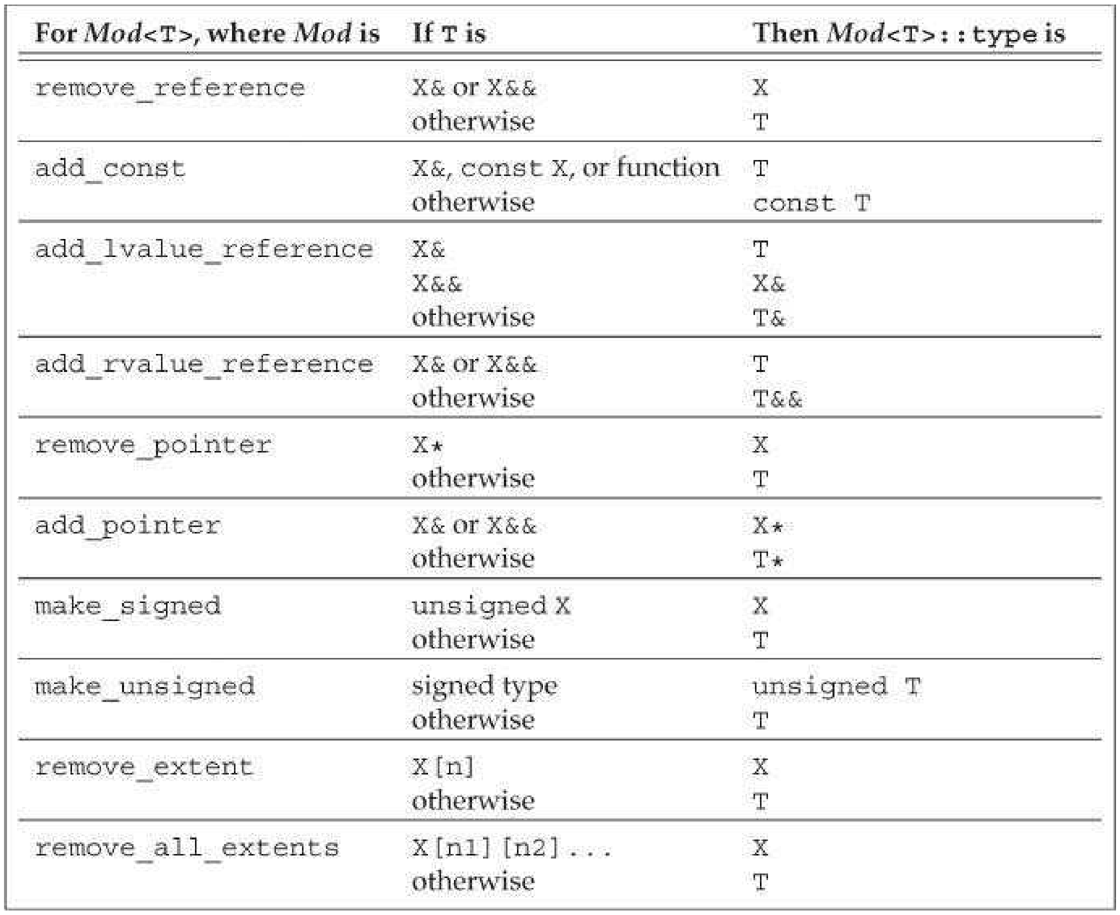
The Type Transformation Library Template Classes

Sometimes we do not have direct access to the type that we need. For example, we might want to write a function similar to `fcn` that returns an element by value (§ 6.3.2, p. 224), rather than a reference to an element.

The problem we face in writing this function is that we know almost nothing about the types we're passed. In this function, the only operations we know we can use are iterator operations, and there are no iterator operations that yield elements (as opposed to references to elements).

To obtain the element type, we can use a library **type transformation** template. These templates are defined in the `type_traits` header. In general the classes in `type_traits` are used for so-called template metaprogramming, a topic that is beyond the scope of this Primer. However, the type transformation templates are useful in ordinary programming as well. These templates are described in Table 16.1 and we'll see how they are implemented in § 16.5 (p. 710).

Table 16.1. Standard Type Transformation Templates



For *Mod*`<T>`, where *Mod* is	If `T` is	Then *Mod*`<T>::type` is
`remove_reference`	`X&` or `X&&`	`X`
	otherwise	`T`
`add_const`	`X&`, `const X`, or function	`T`
	otherwise	`const T`
`add_lvalue_reference`	`X&`	`T`
	`X&&`	`X&`
	otherwise	`T&`
`add_rvalue_reference`	`X&` or `X&&`	`T`
	otherwise	`T&&`
`remove_pointer`	`X*`	`X`
	otherwise	`T`
`add_pointer`	`X&` or `X&&`	`X*`
	otherwise	`T*`
`make_signed`	`unsigned X`	`X`
	otherwise	`T`
`make_unsigned`	signed type	`unsigned T`
	otherwise	`T`
`remove_extent`	`X[n]`	`X`
	otherwise	`T`
`remove_all_extents`	`X[n1][n2]...`	`X`
	otherwise	`T`

In this case, we can use `remove_reference` to obtain the element type. The `remove_reference` template has one template type parameter and a (`public`) type member named `type`. If we instantiate `remove_reference` with a reference type, then `type` will be the referred-to type. For example, if we instantiate `remove_reference<int&>`, the `type` member will be `int`. Similarly, if we instantiate `remove_reference<string&>`, `type` will be `string`, and so on. More generally, given that `beg` is an iterator:

Click here to view code image

```
remove_reference<decltype(*beg)>::type
```

will be the type of the element to which `beg` refers: `decltype(*beg)` returns the reference type of the element type. `remove_reference::type` strips off the reference, leaving the element type itself.

Using `remove_reference` and a trailing return with `decltype`, we can write our function to return a copy of an element's value:

Click here to view code image

```
// must use typename to use a type member of a template parameter; see § 16.1.3 (p. 670)
template <typename It>
```

```
auto fcn2(It beg, It end) ->
    typename remove_reference<decltype(*beg)>::type
{
    // process the range
    return *beg;  // return a copy of an element from the range
}
```

Note that `type` is member of a class that depends on a template parameter. As a result, we must use `typename` in the declaration of the return type to tell the compiler that `type` represents a type (§ 16.1.3, p. 670).

Each of the type transformation templates described in Table 16.1 works similarly to `remove_reference`. Each template has a `public` member named `type` that represents a type. That type may be related to the template's own template type parameter in a way that is indicated by the template's name. If it is not possible (or not necessary) to transform the template's parameter, the `type` member is the template parameter type itself. For example, if `T` is a pointer type, then `remove_pointer<T>::type` is the type to which `T` points. If `T` isn't a pointer, then no transformation is needed. In this case, `type` is the same type as `T`.

Exercises Section 16.2.3

Exercise 16.40: Is the following function legal? If not, why not? If it is legal, what, if any, are the restrictions on the argument type(s) that can be passed, and what is the return type?

Click here to view code image

```
template <typename It>
auto fcn3(It beg, It end) -> decltype(*beg + 0)
{
    // process the range
    return *beg;  // return a copy of an element from the range
}
```

Exercise 16.41: Write a version of `sum` with a return type that is guaranteed to be large enough to hold the result of the addition.

16.2.4. Function Pointers and Argument Deduction

When we initialize or assign a function pointer (§ 6.7, p. 247) from a function template, the compiler uses the type of the pointer to deduce the template argument(s).

As an example, assume we have a function pointer that points to a function returning an `int` that takes two parameters, each of which is a reference to a `const` `int`. We can use that pointer to point to an instantiation of `compare`:

Click here to view code image

```
template <typename T> int compare(const T&, const T&);
// pf1 points to the instantiation int compare(const int&, const int&)
int (*pf1)(const int&, const int&) = compare;
```

The type of the parameters in `pf1` determines the type of the template argument for `T`. The template argument for `T` is `int`. The pointer `pf1` points to the instantiation of `compare` with `T` bound to `int`. It is an error if the template arguments cannot be determined from the function pointer type:

Click here to view code image

```
// overloaded versions of func; each takes a different function pointer type
void func(int(*)(const string&, const string&));
void func(int(*)(const int&, const int&));
func(compare); // error: which instantiation of compare?
```

The problem is that by looking at the type of `func`'s parameter, it is not possible to determine a unique type for the template argument. The call to `func` could instantiate the version of `compare` that takes `int`s or the version that takes `string`s. Because it is not possible to identify a unique instantiation for the argument to `func`, this call won't compile.

We can disambiguate the call to `func` by using explicit template arguments:

Click here to view code image

```
// ok: explicitly specify which version of compare to instantiate
func(compare<int>);  // passing compare(const int&, const int&)
```

This expression calls the version of `func` that takes a function pointer with two `const int&` parameters.

Note

When the address of a function-template instantiation is taken, the context must be such that it allows a unique type or value to be determined for each template parameter.

16.2.5. Template Argument Deduction and References

In order to understand type deduction from a call to a function such as

Click here to view code image

```
template <typename T> void f(T &p);
```

in which the function's parameter `p` is a reference to a template type parameter `T`, it is important to keep in mind two points: Normal reference binding rules apply; and `const`s are low level, not top level.

Type Deduction from Lvalue Reference Function Parameters

When a function parameter is an ordinary (lvalue) reference to a template type parameter (i.e., that has the form `T&`), the binding rules say that we can pass only an lvalue (e.g., a variable or an expression that returns a reference type). That argument might or might not have a `const` type. If the argument is `const`, then `T` will be deduced as a `const` type:

Click here to view code image

```
template <typename T> void f1(T&);   // argument must be an lvalue
// calls to f1 use the referred-to type of the argument as the template parameter type
f1(i);    //  i is an int; template parameter T is int
f1(ci);   //  ci is a const int; template parameter T is const int
f1(5);    //  error: argument to a & parameter must be an lvalue
```

If a function parameter has type `const T&`, normal binding rules say that we can pass any kind of argument—an object (`const` or otherwise), a temporary, or a literal value. When the function parameter is itself `const`, the type deduced for `T` will not be a `const` type. The `const` is already part of the *function* parameter type; therefore, it does not also become part of the *template* parameter type:

Click here to view code image

```
template <typename T> void f2(const T&); // can take an rvalue
// parameter in f2 is const &; const in the argument is irrelevant
// in each of these three calls, f2's function parameter is inferred as const int&
f2(i);   // i is an int; template parameter T is int
f2(ci);  // ci is a const int, but template parameter T is int
f2(5);   // a const & parameter can be bound to an rvalue; T is int
```

Type Deduction from Rvalue Reference Function Parameters

When a function parameter is an rvalue reference (§ 13.6.1, p. 532) (i.e., has the form `T&&`), normal binding rules say that we can pass an rvalue to this parameter. When we do so, type deduction behaves similarly to deduction for an ordinary lvalue reference function parameter. The deduced type for `T` is the type of the rvalue:

Click here to view code image

```
template <typename T> void f3(T&&);
```

```
    f3(42); // argument is an rvalue of type  int; template parameter  T  is  int
```

Reference Collapsing and Rvalue Reference Parameters

Assuming `i` is an `int` object, we might think that a call such as `f3(i)` would be illegal. After all, `i` is an lvalue, and normally we cannot bind an rvalue reference to an lvalue. However, the language defines two exceptions to normal binding rules that allow this kind of usage. These exceptions are the foundation for how library facilities such as `move` operate.

The first exception affects how type deduction is done for rvalue reference parameters. When we pass an lvalue (e.g., `i`) to a function parameter that is an rvalue reference to a template type parameter (e.g, `T&&`), the compiler deduces the template type parameter as the argument's lvalue reference type. So, when we call `f3(i)`, the compiler deduces the type of `T` as `int&`, not `int`.

Deducing `T` as `int&` would seem to mean that `f3`'s function parameter would be an rvalue reference to the type `int&`. Ordinarily, we cannot (directly) define a reference to a reference (§ 2.3.1, p. 51). However, it is possible to do so indirectly through a type alias (§ 2.5.1, p. 67) or through a template type parameter.

C++ 11

In such contexts, we see the second exception to the normal binding rules: If we indirectly create a reference to a reference, then those references "collapse." In all but one case, the references collapse to form an ordinary lvalue reference type. The new standard, expanded the collapsing rules to include rvalue references. References collapse to form an rvalue reference only in the specific case of an rvalue reference to an rvalue reference. That is, for a given type `X`:

- `X& &`, `X& &&`, and `X&& &` all collapse to type `X&`
- The type `X&& &&` collapses to `X&&`

Note

Reference collapsing applies only when a reference to a reference is created indirectly, such as in a type alias or a template parameter.

The combination of the reference collapsing rule and the special rule for type deduction for rvalue reference parameters means that we can call `f3` on an lvalue. When we pass an lvalue to `f3`'s (rvalue reference) function parameter, the compiler will deduce `T` as an lvalue reference type:

Click here to view code image

```
    f3(i);   // argument is an lvalue; template parameter  T  is  int&
    f3(ci); // argument is an lvalue; template parameter  T  is  const int&
```

When a *template* parameter T is deduced as a reference type, the collapsing rule says that the *function* parameter T&& collapses to an lvalue reference type. For example, the resulting instantiation for f3(i) would be something like

Click here to view code image

```
// invalid code, for illustration purposes only
void f3<int&>(int& &&); // when T is int&, function parameter is int& &&
```

The function parameter in f3 is T&& and T is int&, so T&& is int& &&, which collapses to int&. Thus, even though the form of the function parameter in f3 is an rvalue reference (i.e., T&&), this call instantiates f3 with an lvalue reference type (i.e., int&):

Click here to view code image

```
void f3<int&>(int&); // when T is int&, function parameter collapses to int&
```

There are two important consequences from these rules:

- A function parameter that is an rvalue reference to a template type parameter (e.g., T&&) can be bound to an lvalue; and
- If the argument is an lvalue, then the deduced template argument type will be an lvalue reference type and the function parameter will be instantiated as an (ordinary) lvalue reference parameter (T&)

It is also worth noting that by implication, we can pass any type of argument to a T&& function parameter. A parameter of such a type can (obviously) be used with rvalues, and as we've just seen, can be used by lvalues as well.

Note

An argument of any type can be passed to a function parameter that is an rvalue reference to a template parameter type (i.e., T&&). When an lvalue is passed to such a parameter, the function parameter is instantiated as an ordinary, lvalue reference (T&).

Writing Template Functions with Rvalue Reference Parameters

The fact that the template parameter can be deduced to a reference type can have surprising impacts on the code inside the template:

Click here to view code image

```
template <typename T> void f3(T&& val)
{
    T t = val;   // copy or binding a reference?
    t = fcn(t); // does the assignment change only t or val and t?
    if (val == t) { /* ... */ } // always true if T is a reference type
}
```

When we call `f3` on an rvalue, such as the literal 42, `T` is `int`. In this case, the local variable `t` has type `int` and is initialized by copying the value of the parameter `val`. When we assign to `t`, the parameter `val` remains unchanged.

On the other hand, when we call `f3` on the lvalue `i`, then `T` is `int&`. When we define and initialize the local variable `t`, that variable has type `int&`. The initialization of `t` binds `t` to `val`. When we assign to `t`, we change `val` at the same time. In this instantiation of `f3`, the `if` test will always yield `true`.

It is surprisingly hard to write code that is correct when the types involved might be plain (nonreference) types or reference types (although the type transformation classes such as `remove_reference` can help (§ 16.2.3, p. 684)).

In practice, rvalue reference parameters are used in one of two contexts: Either the template is forwarding its arguments, or the template is overloaded. We'll look at forwarding in § 16.2.7 (p. 692) and at template overloading in § 16.3 (p. 694).

For now, it's worth noting that function templates that use rvalue references often use overloading in the same way as we saw in § 13.6.3 (p. 544):

Click here to view code image

```
template <typename T> void f(T&&);        // binds to nonconst rvalues
template <typename T> void f(const T&);   // lvalues and const rvalues
```

As with nontemplate functions, the first version will bind to modifiable rvalues and the second to lvalues or to `const` rvalues.

Exercises Section 16.2.5

Exercise 16.42: Determine the type of `T` and of `val` in each of the following calls:

Click here to view code image

```
template <typename T> void g(T&& val);
int i = 0; const int ci = i;
```

(a) `g(i);`

(b) `g(ci);`

(c) `g(i * ci);`

Exercise 16.43: Using the function defined in the previous exercise, what would the template parameter of g be if we called `g(i = ci)`?

Exercise 16.44: Using the same three calls as in the first exercise, determine the types for T if g's function parameter is declared as T (not `T&&`). What if g's function parameter is `const T&`?

Exercise 16.45: Given the following template, explain what happens if we call g on a literal value such as 42. What if we call g on a variable of type `int`?

Click here to view code image

```
template <typename T> void g(T&& val) { vector<T> v; }
```

16.2.6. Understanding std::move

The library move function (§ 13.6.1, p. 533) is a good illustration of a template that uses rvalue references. Fortunately, we can use move without understanding the template mechanisms that it uses. However, looking at how move works can help cement our general understanding, and use, of templates.

In § 13.6.2 (p. 534) we noted that although we cannot directly bind an rvalue reference to an lvalue, we can use move to obtain an rvalue reference bound to an lvalue. Because move can take arguments of essentially any type, it should not be surprising that move is a function template.

How std::move Is Defined

The standard defines move as follows:

Click here to view code image

```
// for the use of typename in the return type and the cast see § 16.1.3 (p. 670)
// remove_reference is covered in § 16.2.3 (p. 684)
template <typename T>
typename remove_reference<T>::type&& move(T&& t)
{
    // static_cast covered in § 4.11.3 (p. 163)
                                return        static_cast<typename
remove_reference<T>::type&&>(t);
}
```

This code is short but subtle. First, move's function parameter, `T&&`, is an rvalue reference to a template parameter type. Through reference collapsing, this parameter can match arguments of any type. In particular, we can pass either an lvalue or an rvalue to move:

Click here to view code image

```
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // ok: moving from an rvalue
s2 = std::move(s1);   // ok: but after the assigment s1 has indeterminate
value
```

How std::move Works

In the first assignment, the argument to `move` is the rvalue result of the `string` constructor, `string("bye")`. As we've seen, when we pass an rvalue to an rvalue reference function parameter, the type deduced from that argument is the referred-to type (§ 16.2.5, p. 687). Thus, in `std::move(string("bye!"))`:

• The deduced type of `T` is `string`.

• Therefore, `remove_reference` is instantiated with `string`.

• The `type` member of `remove_reference<string>` is `string`.

• The return type of `move` is `string&&`.

• `move`'s function parameter, `t`, has type `string&&`.

Accordingly, this call instantiates `move<string>`, which is the function

```
string&& move(string &&t)
```

The body of this function returns `static_cast<string&&>(t)`. The type of `t` is already `string&&`, so the cast does nothing. Therefore, the result of this call is the rvalue reference it was given.

Now consider the second assignment, which calls `std::move(s1)`. In this call, the argument to `move` is an lvalue. This time:

• The deduced type of `T` is `string&` (reference to `string`, not plain `string`).

• Therefore, `remove_reference` is instantiated with `string&`.

• The `type` member of `remove_reference<string&>` is `string`,

• The return type of `move` is still `string&&`.

• `move`'s function parameter, `t`, instantiates as `string& &&`, which collapses to `string&`.

Thus, this call instantiates `move<string&>`, which is

```
string&& move(string &t)
```

and which is exactly what we're after—we want to bind an rvalue reference to an lvalue. The body of this instantiation returns `static_cast<string&&>(t)`. In this case, the type of `t` is `string&`, which the cast converts to `string&&`.

static_cast from an Lvalue to an Rvalue Reference Is Permitted

Ordinarily, a `static_cast` can perform only otherwise legitimate conversions (§ 4.11.3, p. 163). However, there is again a special dispensation for rvalue references: Even though we cannot implicitly convert an lvalue to an rvalue reference, we can *explicitly* cast an lvalue to an rvalue reference using `static_cast`.

Binding an rvalue reference to an lvalue gives code that operates on the rvalue reference permission to clobber the lvalue. There are times, such as in our `StrVec` `reallocate` function in § 13.6.1 (p. 533), when we know it is safe to clobber an lvalue. By *letting* us do the cast, the language allows this usage. By *forcing* us to use a cast, the language tries to prevent us from doing so accidentally.

Finally, although we can write such casts directly, it is much easier to use the library `move` function. Moreover, using `std::move` consistently makes it easy to find the places in our code that might potentially clobber lvalues.

Exercises Section 16.2.6

Exercise 16.46: Explain this loop from `StrVec::reallocate` in § 13.5 (p. 530):

Click here to view code image

```
for (size_t i = 0; i != size(); ++i)
    alloc.construct(dest++, std::move(*elem++));
```

16.2.7. Forwarding

Some functions need to forward one or more of their arguments with their types *unchanged* to another, forwarded-to, function. In such cases, we need to preserve everything about the forwarded arguments, including whether or not the argument type is `const`, and whether the argument is an lvalue or an rvalue.

As an example, we'll write a function that takes a callable expression and two additional arguments. Our function will call the given callable with the other two arguments in reverse order. The following is a first cut at our flip function:

Click here to view code image

```
// template that takes a callable and two parameters
// and calls the given callable with the parameters "flipped"
// flip1 is an incomplete implementation: top-level const and references are lost
```

```
template <typename F, typename T1, typename T2>
void flip1(F f, T1 t1, T2 t2)
{
    f(t2, t1);
}
```

This template works fine until we want to use it to call a function that has a reference parameter:

Click here to view code image

```
void f(int v1, int &v2) // note v2 is a reference
{
    cout << v1 << " " << ++v2 << endl;
}
```

Here `f` changes the value of the argument bound to `v2`. However, if we call `f` through `flip1`, the changes made by `f` do not affect the original argument:

Click here to view code image

```
f(42, i);         // f changes its argument i
flip1(f, j, 42); // f called through flip1 leaves j unchanged
```

The problem is that `j` is passed to the `t1` parameter in `flip1`. That parameter has is a plain, nonreference type, `int`, not an `int&`. That is, the instantiation of this call to `flip1` is

Click here to view code image

```
void flip1(void(*fcn)(int, int&), int t1, int t2);
```

The value of `j` is copied into `t1`. The reference parameter in `f` is bound to `t1`, not to `j`.

Defining Function Parameters That Retain Type Information

To pass a reference through our flip function, we need to rewrite our function so that its parameters preserve the “lvalueness” of its given arguments. Thinking ahead a bit, we can imagine that we’d also like to preserve the `const`ness of the arguments as well.

We can preserve all the type information in an argument by defining its corresponding function parameter as an rvalue reference to a template type parameter. Using a reference parameter (either lvalue or rvalue) lets us preserve `const`ness, because the `const` in a reference type is low-level. Through reference collapsing (§ 16.2.5, p. 688), if we define the function parameters as T1&& and T2&&, we can preserve the lvalue/rvalue property of flip’s arguments (§ 16.2.5, p. 687):

Click here to view code image

```
template <typename F, typename T1, typename T2>
void flip2(F f, T1 &&t1, T2 &&t2)
{
    f(t2, t1);
}
```

As in our earlier call, if we call `flip2(f, j, 42)`, the lvalue `j` is passed to the parameter `t1`. However, in `flip2`, the type deduced for `T1` is `int&`, which means that the type of `t1` collapses to `int&`. The reference `t1` is bound to `j`. When `flip2` calls `f`, the reference parameter `v2` in `f` is bound to `t1`, which in turn is bound to `j`. When `f` increments `v2`, it is changing the value of `j`.

Note

A function parameter that is an rvalue reference to a template type parameter (i.e., `T&&`) preserves the `const`ness and lvalue/rvalue property of its corresponding argument.

This version of `flip2` solves one half of our problem. Our `flip2` function works fine for functions that take lvalue references but cannot be used to call a function that has an rvalue reference parameter. For example:

Click here to view code image

```
void g(int &&i, int& j)
{
    cout << i << " " << j << endl;
}
```

If we try to call `g` through `flip2`, we will be passing the parameter `t2` to `g`'s rvalue reference parameter. Even if we pass an rvalue to `flip2`:

Click here to view code image

```
flip2(g, i, 42); // error: can't initialize int&& from an lvalue
```

what is passed to `g` will be the parameter named `t2` inside `flip2`. A function parameter, like any other variable, is an lvalue expression (§ 13.6.1, p. 533). As a result, the call to `g` in `flip2` passes an lvalue to `g`'s rvalue reference parameter.

Using `std::forward` to Preserve Type Information in a Call

We can use a new library facility named `forward` to pass `flip2`'s parameters in a way that preserves the types of the original arguments. Like `move`, `forward` is defined in the `utility` header. Unlike `move`, `forward` must be called with an explicit template argument (§ 16.2.2, p. 682). `forward` returns an rvalue reference to that explicit argument type. That is, the return type of `forward<T>` is `T&&`.

C++ 11

Ordinarily, we use `forward` to pass a function parameter that is defined as an rvalue reference to a template type parameter. Through reference collapsing on its return type, `forward` preserves the lvalue/rvalue nature of its given argument:

Click here to view code image

```
template <typename Type> intermediary(Type &&arg)
{
    finalFcn(std::forward<Type>(arg));
    // ...
}
```

Here we use `Type`—which is deduced from `arg`—as `forward`'s explicit template argument type. Because `arg` is an rvalue reference to a template type parameter, `Type` will represent all the type information in the argument passed to `arg`. If that argument was an rvalue, then `Type` is an ordinary (nonreference) type and `forward<Type>` will return `Type&&`. If the argument was an lvalue, then—through reference collapsing—`Type` itself is an lvalue reference type. In this case, the return type is an rvalue reference to an lvalue reference type. Again through reference collapsing—this time on the return type—`forward<Type>` will return an lvalue reference type.

Note

When used with a function parameter that is an rvalue reference to template type parameter (`T&&`), `forward` preserves all the details about an argument's type.

Using `forward`, we'll rewrite our flip function once more:

Click here to view code image

```
template <typename F, typename T1, typename T2>
void flip(F f, T1 &&t1, T2 &&t2)
{
    f(std::forward<T2>(t2), std::forward<T1>(t1));
}
```

If we call `flip(g, i, 42)`, `i` will be passed to g as an `int&` and `42` will be passed as an `int&&`.

Note

As with `std::move`, it's a good idea not to provide a `using` declaration for `std::forward`. § 18.2.3 (p. 798) will explain why.

16.3. Overloading and Templates

Function templates can be overloaded by other templates or by ordinary, nontemplate functions. As usual, functions with the same name must differ either as to the number or the type(s) of their parameters.

Exercises Section 16.2.7

Exercise 16.47: Write your own version of the flip function and test it by calling functions that have lvalue and rvalue reference parameters.

Function matching (§ 6.4, p. 233) is affected by the presence of function templates in the following ways:

• The candidate functions for a call include any function-template instantiation for which template argument deduction (§ 16.2, p. 678) succeeds.

• The candidate function templates are always viable, because template argument deduction will have eliminated any templates that are not viable.

• As usual, the viable functions (template and nontemplate) are ranked by the conversions, if any, needed to make the call. Of course, the conversions used to call a function template are quite limited (§ 16.2.1, p. 679).

• Also as usual, if exactly one function provides a better match than any of the others, that function is selected. However, if there are several functions that provide an equally good match, then:

– If there is only one nontemplate function in the set of equally good matches, the nontemplate function is called.

– If there are no nontemplate functions in the set, but there are multiple function templates, and one of these templates is more specialized than any of the others, the more specialized function template is called.

– Otherwise, the call is ambiguous.

Warning

Correctly defining a set of overloaded function templates requires a good understanding of the relationship among types and of the restricted conversions applied to arguments in template functions.

Writing Overloaded Templates

As an example, we’ll build a set of functions that might be useful during debugging. We’ll name our debugging functions `debug_rep`, each of which will return a `string` representation of a given object. We’ll start by writing the most general version of this function as a template that takes a reference to a `const` object:

Click here to view code image

```
// print any type we don't otherwise handle
template <typename T> string debug_rep(const T &t)
{
    ostringstream ret; // see § 8.3 (p. 321)
    ret << t; // uses T's output operator to print a representation of t
    return ret.str(); // return a copy of the string to which ret is bound
}
```

This function can be used to generate a `string` corresponding to an object of any type that has an output operator.

Next, we’ll define a version of `debug_rep` to print pointers:

Click here to view code image

```
// print pointers as their pointer value, followed by the object to which the pointer points
// NB: this function will not work properly with char*; see § 16.3 (p. 698)
template <typename T> string debug_rep(T *p)
{
    ostringstream ret;
    ret << "pointer: " << p;          // print the pointer's own value
    if (p)
         ret << " " << debug_rep(*p); // print the value to which p
points
    else
        ret << " null pointer";       // or indicate that the p is null
    return ret.str(); // return a copy of the string to which ret is bound
}
```

This version generates a `string` that contains the pointer’s own value and calls `debug_rep` to print the object to which that pointer points. Note that this function can’t be used to print character pointers, because the IO library defines a version of the << for `char*` values. That version of << assumes the pointer denotes a null-terminated character array, and prints the contents of the array, not its address. We’ll see in § 16.3 (p. 698) how to handle character pointers.

We might use these functions as follows:

Click here to view code image

```
string s("hi");
```

```
cout << debug_rep(s) << endl;
```

For this call, only the first version of `debug_rep` is viable. The second version of `debug_rep` requires a pointer parameter, and in this call we passed a nonpointer object. There is no way to instantiate a function template that expects a pointer type from a nonpointer argument, so argument deduction fails. Because there is only one viable function, that is the one that is called.

If we call `debug_rep` with a pointer:

Click here to view code image

```
cout << debug_rep(&s) << endl;
```

both functions generate viable instantiations:

- `debug_rep(const string* &)`, which is the instantiation of the first version of `debug_rep` with `T` bound to `string*`
- `debug_rep(string*)`, which is the instantiation of the second version of `debug_rep` with `T` bound to `string`

The instantiation of the second version of `debug_rep` is an exact match for this call. The instantiation of the first version requires a conversion of the plain pointer to a pointer to `const`. Normal function matching says we should prefer the second template, and indeed that is the one that is run.

Multiple Viable Templates

As another example, consider the following call:

Click here to view code image

```
const string *sp = &s;
cout << debug_rep(sp) << endl;
```

Here both templates are viable and both provide an exact match:

- `debug_rep(const string* &)`, the instantiation of the first version of the template with `T` bound to `const string*`
- `debug_rep(const string*)`, the instantiation of the second version of the template with `T` bound to `const string`

In this case, normal function matching can't distinguish between these two calls. We might expect this call to be ambiguous. However, due to the special rule for overloaded function templates, this call resolves to `debug_rep(T*)`, which is the more specialized template.

The reason for this rule is that without it, there would be no way to call the pointer version of `debug_rep` on a pointer to `const`. The problem is that the template `debug_rep(const T&)` can be called on essentially any type, including pointer types. That template is more general than `debug_rep(T*)`, which can be called only on pointer types. Without this rule, calls that passed pointers to `const` would always be ambiguous.

Note

When there are several overloaded templates that provide an equally good match for a call, the most specialized version is preferred.

Nontemplate and Template Overloads

For our next example, we'll define an ordinary nontemplate version of `debug_rep` to print `string`s inside double quotes:

Click here to view code image

```
// print strings inside double quotes
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
```

Now, when we call `debug_rep` on a `string`,

Click here to view code image

```
string s("hi");
cout << debug_rep(s) << endl;
```

there are two equally good viable functions:

- `debug_rep<string>(const string&)`, the first template with `T` bound to `string`
- `debug_rep(const string&)`, the ordinary, nontemplate function

In this case, both functions have the same parameter list, so obviously, each function provides an equally good match for this call. However, the nontemplate version is selected. For the same reasons that the most specialized of equally good function templates is preferred, a nontemplate function is preferred over equally good match(es) to a function template.

Note

When a nontemplate function provides an equally good match for a call as a function template, the nontemplate version is preferred.

Overloaded Templates and Conversions

There's one case we haven't covered so far: pointers to C-style character strings and string literals. Now that we have a version of `debug_rep` that takes a `string`, we might expect that a call that passes character strings would match that version. However, consider this call:

Click here to view code image

```
cout << debug_rep("hi world!") << endl; // calls debug_rep(T*)
```

Here all three of the `debug_rep` functions are viable:

- `debug_rep(const T&)`, with `T` bound to `char[10]`
- `debug_rep(T*)`, with `T` bound to `const char`
- `debug_rep(const string&)`, which requires a conversion from `const char*` to `string`

Both templates provide an exact match to the argument—the second template requires a (permissible) conversion from array to pointer, and that conversion is considered as an exact match for function-matching purposes (§ 6.6.1, p. 245). The nontemplate version is viable but requires a user-defined conversion. That function is less good than an exact match, leaving the two templates as the possible functions to call. As before, the `T*` version is more specialized and is the one that will be selected.

If we want to handle character pointers as `string`s, we can define two more nontemplate overloads:

Click here to view code image

```
// convert the character pointers to string and call the string version of debug_rep
string debug_rep(char *p)
{
    return debug_rep(string(p));
}
string debug_rep(const char *p)
{
    return debug_rep(string(p));
}
```

Missing Declarations Can Cause the Program to Misbehave

It is worth noting that for the `char*` versions of `debug_rep` to work correctly, a declaration for `debug_rep(const string&)` must be in scope when these functions are defined. If not, the wrong version of `debug_rep` will be called:

Click here to view code image

```
template <typename T> string debug_rep(const T &t);
template <typename T> string debug_rep(T *p);
// the following declaration must be in scope
// for the definition of debug_rep(char*) to do the right thing
string debug_rep(const string &);
string debug_rep(char *p)
{
    // if the declaration for the version that takes a const string& is not in scope
    // the return will call debug_rep(const T&) with T instantiated to string
    return debug_rep(string(p));
}
```

Ordinarily, if we use a function that we forgot to declare, our code won't compile. Not so with functions that overload a template function. If the compiler can instantiate the call from the template, then the missing declaration won't matter. In this example, if we forget to declare the version of `debug_rep` that takes a `string`, the compiler will *silently* instantiate the template version that takes a `const T&`.

Tip

Declare every function in an overload set before you define any of the functions. That way you don't have to worry whether the compiler will instantiate a call before it sees the function you intended to call.

Exercises Section 16.3

Exercise 16.48: Write your own versions of the `debug_rep` functions.

Exercise 16.49: Explain what happens in each of the following calls:

Click here to view code image

```
template <typename T> void f(T);
template <typename T> void f(const T*);
template <typename T> void g(T);
template <typename T> void g(T*);
int i = 42, *p = &i;
const int ci = 0, *p2 = &ci;
g(42);   g(p);   g(ci);   g(p2);
f(42);   f(p);   f(ci);   f(p2);
```

Exercise 16.50: Define the functions from the previous exercise so that they print an identifying message. Run the code from that exercise. If the calls behave differently from what you expected, make sure you understand why.

16.4. Variadic Templates

A **variadic template** is a template function or class that can take a varying number of parameters. The varying parameters are known as a **parameter pack**. There are two kinds of parameter packs: A **template parameter pack** represents zero or more template parameters, and a **function parameter pack** represents zero or more function parameters.

We use an ellipsis to indicate that a template or function parameter represents a pack. In a template parameter list, `class...` or `typename...` indicates that the following parameter represents a list of zero or more types; the name of a type followed by an ellipsis represents a list of zero or more nontype parameters of the given type. In the function parameter list, a parameter whose type is a template parameter pack is a function parameter pack. For example:

Click here to view code image

```
// Args is a template parameter pack; rest is a function parameter pack
// Args represents zero or more template type parameters
// rest represents zero or more function parameters
template <typename T, typename... Args>
void foo(const T &t, const Args& ... rest);
```

declares that `foo` is a variadic function that has one type parameter named `T` and a template parameter pack named `Args`. That pack represents zero or more additional type parameters. The function parameter list of `foo` has one parameter, whose type is a `const &` to whatever type `T` has, and a function parameter pack named `rest`. That pack represents zero or more function parameters.

As usual, the compiler deduces the template parameter types from the function's arguments. For a variadic template, the compiler also deduces the number of parameters in the pack. For example, given these calls:

Click here to view code image

```
int i = 0; double d = 3.14; string s = "how now brown cow";
foo(i, s, 42, d);    // three parameters in the pack
foo(s, 42, "hi");    // two parameters in the pack
foo(d, s);           // one parameter in the pack
foo("hi");           // empty pack
```

the compiler will instantiate four different instances of `foo`:

Click here to view code image

```
void  foo(const  int&,  const  string&,  const  int&,  const
double&);
void foo(const string&, const int&, const char[3]&);
void foo(const double&, const string&);
void foo(const char[3]&);
```

In each case, the type of `T` is deduced from the type of the first argument. The remaining arguments (if any) provide the number of, and types for, the additional arguments to the function.

The sizeof... Operator

When we need to know how many elements there are in a pack, we can use the `sizeof...` operator. Like `sizeof` (§ 4.9, p. 156), `sizeof...` returns a constant expression (§ 2.4.4, p. 65) and does not evaluate its argument:

Click here to view code image

```
template<typename ... Args> void g(Args ... args) {
    cout << sizeof...(Args) << endl;  // number of type parameters
    cout << sizeof...(args) << endl;  // number of function parameters
}
```

Exercises Section 16.4

Exercise 16.51: Determine what `sizeof...(Args)` and `sizeof...(rest)` return for each call to `foo` in this section.

Exercise 16.52: Write a program to check your answer to the previous question.

16.4.1. Writing a Variadic Function Template

In § 6.2.6 (p. 220) we saw that we can use an `initializer_list` to define a function that can take a varying number of arguments. However, the arguments must have the same type (or types that are convertible to a common type). Variadic functions are used when we know neither the number nor the types of the arguments we want to process. As an example, we'll define a function like our earlier `error_msg` function, only this time we'll allow the argument types to vary as well. We'll start by defining a variadic function named `print` that will print the contents of a given list of arguments on a given stream.

Variadic functions are often recursive (§ 6.3.2, p. 227). The first call processes the first argument in the pack and calls itself on the remaining arguments. Our `print` function will execute this way—each call will print its second argument on the stream denoted by its first argument. To stop the recursion, we'll also need to define a nonvariadic `print` function that will take a stream and an object:

Click here to view code image

```
// function to end the recursion and print the last element
// this function must be declared before the variadic version of print is defined
template<typename T>
ostream &print(ostream &os, const T &t)
{
    return os << t; // no separator after the last element in the pack
}
// this version of print will be called for all but the last element in the pack
template <typename T, typename... Args>
ostream &print(ostream &os, const T &t, const Args&... rest)
{
    os << t << ", ";            // print the first argument
      return print(os, rest...); // recursive call; print the other
arguments
}
```

The first version of `print` stops the recursion and prints the last argument in the initial call to `print`. The second, variadic, version prints the argument bound to `t` and calls itself to print the remaining values in the function parameter pack.

The key part is the call to `print` inside the variadic function:

Click here to view code image

```
return print(os, rest...); // recursive call; print the other arguments
```

The variadic version of our `print` function takes three parameters: an `ostream&`, a `const T&`, and a parameter pack. Yet this call passes only two arguments. What happens is that the first argument in `rest` gets bound to `t`. The remaining arguments in `rest` form the parameter pack for the next call to `print`. Thus, on each call, the first argument in the pack is removed from the pack and becomes the argument bound to `t`. That is, given:

Click here to view code image

```
print(cout, i, s, 42);  // two parameters in the pack
```

the recursion will execute as follows:

Call	t	rest...
`print(cout, i, s, 42)`	`i`	`s, 42`
`print(cout, s, 42)`	`s`	`42`
`print(cout, 42)` *calls the nonvariadic version of* `print`		

The first two calls can match only the variadic version of `print` because the nonvariadic version isn’t viable. These calls pass four and three arguments, respectively, and the nonvariadic `print` takes only two arguments.

For the last call in the recursion, `print(cout, 42)`, both versions of `print` are viable. This call passes exactly two arguments, and the type of the first argument is `ostream&`. Thus, the nonvariadic version of `print` is viable.

The variadic version is also viable. Unlike an ordinary argument, a parameter pack can be empty. Hence, the variadic version of `print` can be instantiated with only two parameters: one for the `ostream&` parameter and the other for the `const T&` parameter.

Both functions provide an equally good match for the call. However, a nonvariadic template is more specialized than a variadic template, so the nonvariadic version is chosen for this call (§ 16.3, p. 695).

Warning

A declaration for the nonvariadic version of `print` must be in scope when the variadic version is defined. Otherwise, the variadic function will recurse indefinitely.

Exercises Section 16.4.1

Exercise 16.53: Write your own version of the `print` functions and test them by printing one, two, and five arguments, each of which should have different types.

Exercise 16.54: What happens if we call `print` on a type that doesn't have an << operator?

Exercise 16.55: Explain how the variadic version of `print` would execute if we declared the nonvariadic version of `print` after the definition of the variadic version.

16.4.2. Pack Expansion

Aside from taking its size, the only other thing we can do with a parameter pack is to **expand** it. When we expand a pack, we also provide a **pattern** to be used on each expanded element. Expanding a pack separates the pack into its constituent elements, applying the pattern to each element as it does so. We trigger an expansion by putting an ellipsis (. . .) to the right of the pattern.

For example, our `print` function contains two expansions:

Click here to view code image

```
template <typename T, typename... Args>
ostream &
print(ostream &os, const T &t, const Args&... rest)// expand Args
```

```
{
    os << t << ", ";
    return print(os, rest...);                          // expand rest
}
```

The first expansion expands the template parameter pack and generates the function parameter list for `print`. The second expansion appears in the call to `print`. That pattern generates the argument list for the call to `print`.

The expansion of `Args` applies the pattern `const Args&` to each element in the template parameter pack `Args`. The expansion of this pattern is a comma-separated list of zero or more parameter types, each of which will have the form `const` *type*`&`. For example:

Click here to view code image

```
print(cout, i, s, 42);   // two parameters in the pack
```

The types of the last two arguments along with the pattern determine the types of the trailing parameters. This call is instantiated as

Click here to view code image

```
ostream&
print(ostream&, const int&, const string&, const int&);
```

The second expansion happens in the (recursive) call to `print`. In this case, the pattern is the name of the function parameter pack (i.e., `rest`). This pattern expands to a comma-separated list of the elements in the pack. Thus, this call is equivalent to

```
print(os, s, 42);
```

Understanding Pack Expansions

The expansion of the function parameter pack in `print` just expanded the pack into its constituent parts. More complicated patterns are also possible when we expand a function parameter pack. For example, we might write a second variadic function that calls `debug_rep` (§ 16.3, p. 695) on each of its arguments and then calls `print` to print the resulting `string`s:

Click here to view code image

```
// call debug_rep on each argument in the call to print
template <typename... Args>
ostream &errorMsg(ostream &os, const Args&... rest)
{
    // print(os, debug_rep(a1), debug_rep(a2), ..., debug_rep(an)
    return print(os, debug_rep(rest)...);
}
```

The call to `print` uses the pattern `debug_rep(rest)`. That pattern says that we want to call `debug_rep` on each element in the function parameter pack `rest`. The resulting expanded pack will be a comma-separated list of calls to `debug_rep`. That is, a call such as

Click here to view code image

```
errorMsg(cerr,   fcnName,   code.num(),   otherData,   "other",
item);
```

will execute as if we had written

Click here to view code image

```
print(cerr, debug_rep(fcnName), debug_rep(code.num()),
            debug_rep(otherData), debug_rep("otherData"),
            debug_rep(item));
```

In contrast, the following pattern would fail to compile:

Click here to view code image

```
// passes the pack to debug_rep; print(os, debug_rep(a1, a2, ..., an))
print(os, debug_rep(rest...)); // error: no matching function to call
```

The problem here is that we expanded `rest` in the call to `debug_rep`. This call would execute as if we had written

Click here to view code image

```
print(cerr, debug_rep(fcnName, code.num(),
                      otherData, "otherData", item));
```

In this expansion, we attempted to call `debug_rep` with a list of five arguments. There is no version of `debug_rep` that matches this call. The `debug_rep` function is not variadic and there is no version of `debug_rep` that has five parameters.

Note

The pattern in an expansion applies separately to each element in the pack.

Exercises Section 16.4.2

Exercise 16.56: Write and test a variadic version of `errorMsg`.

Exercise 16.57: Compare your variadic version of `errorMsg` to the `error_msg` function in § 6.2.6 (p. 220). What are the advantages and disadvantages of each approach?

16.4.3. Forwarding Parameter Packs

Under the new standard, we can use variadic templates together with `forward` to write functions that pass their arguments unchanged to some other function. To illustrate such functions, we'll add an `emplace_back` member to our `StrVec` class (§ 13.5, p. 526). The `emplace_back` member of the library containers is a variadic member template (§ 16.1.4, p. 673) that uses its arguments to construct an element directly in space managed by the container.

Our version of `emplace_back` for `StrVec` will also have to be variadic, because `string` has a number of constructors that differ in terms of their parameters. Because we'd like to be able to use the `string` move constructor, we'll also need to preserve all the type information about the arguments passed to `emplace_back`.

As we've seen, preserving type information is a two-step process. First, to preserve type information in the arguments, we must define `emplace_back`'s function parameters as rvalue references to a template type parameter (§ 16.2.7, p. 693):

Click here to view code image

```
class StrVec {
public:
    template <class... Args> void emplace_back(Args&&...);
    //  remaining members as in  § 13.5 (p. 526)
};
```

The pattern in the expansion of the template parameter pack, `&&`, means that each function parameter will be an rvalue reference to its corresponding argument.

Second, we must use `forward` to preserve the arguments' original types when `emplace_back` passes those arguments to `construct` (§ 16.2.7, p. 694):

Click here to view code image

```
template <class... Args>
inline
void StrVec::emplace_back(Args&&... args)
{
    chk_n_alloc(); //  reallocates the  StrVec  if necessary
    alloc.construct(first_free++,
std::forward<Args>(args)...);
}
```

The body of `emplace_back` calls `chk_n_alloc` (§ 13.5, p. 526) to ensure that there is enough room for an element and calls `construct` to create an element in the `first_free` spot. The expansion in the call to `construct`:

```
std::forward<Args>(args)...
```

expands both the template parameter pack, `Args`, and the function parameter pack, `args`. This pattern generates elements with the form

```
std::forward<Ti>(ti)
```

where T_i represents the type of the *i*th element in the template parameter pack and t_i represents the *i*th element in the function parameter pack. For example, assuming `svec` is a `StrVec`, if we call

Click here to view code image

```
svec.emplace_back(10, 'c'); // adds cccccccccc as a new last element
```

the pattern in the call to `construct` will expand to

Click here to view code image

```
std::forward<int>(10), std::forward<char>(c)
```

By using `forward` in this call, we guarantee that if `emplace_back` is called with an rvalue, then `construct` will also get an rvalue. For example, in this call:

Click here to view code image

```
svec.emplace_back(s1 + s2); // uses the move constructor
```

the argument to `emplace_back` is an rvalue, which is passed to `construct` as

Click here to view code image

```
std::forward<string>(string("the end"))
```

The result type from `forward<string>` is `string&&`, so `construct` will be called with an rvalue reference. The `construct` function will, in turn, forward this argument to the `string` move constructor to build this element.

Advice: Forwarding and Variadic Templates

Variadic functions often forward their parameters to other functions. Such functions typically have a form similar to our `emplace_back` function:

Click here to view code image

```
// fun has zero or more parameters each of which is
// an rvalue reference to a template parameter type
template<typename... Args>
void fun(Args&&... args) // expands Args as a list of rvalue references
{
    // the argument to work expands both Args and args
    work(std::forward<Args>(args)...);
}
```

Here we want to forward all of `fun`’s arguments to another function named `work` that presumably does the real work of the function. Like our call to `construct` inside `emplace_back`, the expansion in the call to `work` expands both the template parameter pack and the function parameter pack.

Because the parameters to fun are rvalue references, we can pass arguments of any type to fun; because we use std::forward to pass those arguments, all type information about those arguments will be preserved in the call to work.

Exercises Section 16.4.3

Exercise 16.58: Write the emplace_back function for your StrVec class and for the Vec class that you wrote for the exercises in § 16.1.2 (p. 668).

Exercise 16.59: Assuming s is a string, explain svec.emplace_back(s).

Exercise 16.60: Explain how make_shared (§ 12.1.1, p. 451) works.

Exercise 16.61: Define your own version of make_shared.

16.5. Template Specializations

It is not always possible to write a single template that is best suited for every possible template argument with which the template might be instantiated. In some cases, the general template definition is simply wrong for a type: The general definition might not compile or might do the wrong thing. At other times, we may be able to take advantage of some specific knowledge to write more efficient code than would be instantiated from the template. When we can't (or don't want to) use the template version, we can define a specialized version of the class or function template.

Our compare function is a good example of a function template for which the general definition is not appropriate for a particular type, namely, character pointers. We'd like compare to compare character pointers by calling strcmp rather than by comparing the pointer values. Indeed, we have already overloaded the compare function to handle character string literals (§ 16.1.1, p. 654):

Click here to view code image

```
// first version; can compare any two types
template <typename T> int compare(const T&, const T&);
// second version to handle string literals
template<size_t N, size_t M>
int compare(const char (&)[N], const char (&)[M]);
```

However, the version of compare that has two nontype template parameters will be called only when we pass a string literal or an array. If we call compare with character pointers, the first version of the template will be called:

Click here to view code image

```
const char *p1 = "hi", *p2 = "mom";
compare(p1, p2);       // calls the first template
compare("hi", "mom"); // calls the template with two nontype parameters
```

There is no way to convert a pointer to a reference to an array, so the second version of `compare` is not viable when we pass `p1` and `p2` as arguments.

To handle character pointers (as opposed to arrays), we can define a **template specialization** of the first version of `compare`. A specialization is a separate definition of the template in which one or more template parameters are specified to have particular types.

Defining a Function Template Specialization

When we specialize a function template, we must supply arguments for every template parameter in the original template. To indicate that we are specializing a template, we use the keyword `template` followed by an empty pair of angle brackets (`< >`). The empty brackets indicate that arguments will be supplied for all the template parameters of the original template:

Click here to view code image

```
// special version of compare to handle pointers to character arrays
template <>
int compare(const char* const &p1, const char* const &p2)
{
    return strcmp(p1, p2);
}
```

The hard part in understanding this specialization is the function parameter types. When we define a specialization, the function parameter type(s) must match the corresponding types in a previously declared template. Here we are specializing:

Click here to view code image

```
template <typename T> int compare(const T&, const T&);
```

in which the function parameters are references to a `const` type. As with type aliases, the interaction between template parameter types, pointers, and `const` can be surprising (§ 2.5.1, p. 68).

We want to define a specialization of this function with `T` as `const char*`. Our function requires a reference to the `const` version of this type. The `const` version of a pointer type is a constant pointer as distinct from a pointer to `const` (§ 2.4.2, p. 63). The type we need to use in our specialization is `const char* const &`, which is a reference to a `const` pointer to `const char`.

Function Overloading versus Template Specializations

When we define a function template specialization, we are essentially taking over the job of the compiler. That is, we are supplying the definition to use for a specific instantiation of the original template. It is important to realize that a specialization is an instantiation; it is not an overloaded instance of the function name.

Note

Specializations instantiate a template; they do not overload it. As a result, specializations do not affect function matching.

Whether we define a particular function as a specialization or as an independent, nontemplate function can impact function matching. For example, we have defined two versions of our `compare` function template, one that takes references to array parameters and the other that takes `const T&`. The fact that we also have a specialization for character pointers has no impact on function matching. When we call `compare` on a string literal:

```
compare("hi", "mom")
```

both function templates are viable and provide an equally good (i.e., exact) match to the call. However, the version with character array parameters is more specialized (§ 16.3, p. 695) and is chosen for this call.

Had we defined the version of `compare` that takes character pointers as a plain nontemplate function (rather than as a specialization of the template), this call would resolve differently. In this case, there would be three viable functions: the two templates and the nontemplate character-pointer version. All three are also equally good matches for this call. As we've seen, when a nontemplate provides an equally good match as a function template, the nontemplate is selected (§ 16.3, p. 695)

Key Concept: Ordinary Scope Rules Apply to Specializations

In order to specialize a template, a declaration for the original template must be in scope. Moreover, a declaration for a specialization must be in scope before any code uses that instantiation of the template.

With ordinary classes and functions, missing declarations are (usually) easy to find—the compiler won't be able to process our code. However, if a specialization declaration is missing, the compiler will usually generate code using the original template. Because the compiler can often instantiate the original template when a specialization is missing, errors in declaration order between a template and its specializations are easy to make but hard to find.

It is an error for a program to use a specialization and an instantiation of the original template with the same set of template arguments. However, it is an error that the compiler is unlikely to detect.

Best Practices

Templates and their specializations should be declared in the same header file. Declarations for all the templates with a given name should appear first, followed by any specializations of those templates.

Class Template Specializations

In addition to specializing function templates, we can also specialize class templates. As an example, we'll define a specialization of the library `hash` template that we can use to store `Sales_data` objects in an unordered container. By default, the unordered containers use `hash<key_type>` (§ 11.4, p. 444) to organize their elements. To use this default with our own data type, we must define a specialization of the `hash` template. A specialized `hash` class must define

- An overloaded call operator (§ 14.8, p. 571) that returns a `size_t` and takes an object of the container's key type
- Two type members, `result_type` and `argument_type`, which are the return and argument types, respectively, of the call operator
- The default constructor and a copy-assignment operator (which can be implicitly defined (§ 13.1.2, p. 500))

The only complication in defining this `hash` specialization is that when we specialize a template, we must do so in the same namespace in which the original template is defined. We'll have more to say about namespaces in § 18.2 (p. 785). For now, what we need to know is that we can add members to a namespace. To do so, we must first open the namespace:

Click here to view code image

```
// open the std namespace so we can specialize std::hash
namespace std {
}  // close the std namespace; note: no semicolon after the close curly
```

Any definitions that appear between the open and close curlies will be part of the `std` namespace.

The following defines a specialization of `hash` for `Sales_data`:

Click here to view code image

```
//
```

```
      open the  std  namespace so we can specialize  std::hash
   namespace std {
   template <>              //  we're defining a specialization with
   struct hash<Sales_data> //  the template parameter of  Sales_data
   {
       //  the type used to hash an unordered container must define these types
       typedef size_t result_type;
        typedef  Sales_data  argument_type;  //  by  default,  this  type  needs
   ==
       size_t operator()(const Sales_data& s) const;
       //  our class uses synthesized copy control and default constructor
   };
   size_t
   hash<Sales_data>::operator()(const Sales_data& s) const
   {
       return hash<string>()(s.bookNo) ^
              hash<unsigned>()(s.units_sold) ^
              hash<double>()(s.revenue);
   }
   } //  close the  std  namespace; note: no semicolon after the close curly
```

Our `hash<Sales_data>` definition starts with `template<>`, which indicates that we are defining a fully specialized template. The template we're specializing is named `hash` and the specialized version is `hash<Sales_data>`. The members of the class follow directly from the requirements for specializing `hash`.

As with any other class, we can define the members of a specialization inside the class or out of it, as we did here. The overloaded call operator must define a hashing function over the values of the given type. This function is required to return the same result every time it is called for a given value. A good hash function will (almost always) yield different results for objects that are not equal.

Here, we delegate the complexity of defining a good hash function to the library. The library defines specializations of the `hash` class for the built-in types and for many of the library types. We use an (unnamed) `hash<string>` object to generate a hash code for `bookNo`, an object of type `hash<unsigned>` to generate a hash from `units_sold`, and an object of type `hash<double>` to generate a hash from `revenue`. We exclusive OR (§ 4.8, p. 154) these results to form an overall hash code for the given `Sales_data` object.

It is worth noting that we defined our `hash` function to hash all three data members so that our `hash` function will be compatible with our definition of `operator==` for `Sales_data` (§ 14.3.1, p. 561). By default, the unordered containers use the specialization of `hash` that corresponds to the `key_type` along with the equality operator on the key type.

Assuming our specialization is in scope, it will be used automatically when we use `Sales_data` as a key to one of these containers:

Click here to view code image

```
// uses hash<Sales_data> and Sales_data operator==from § 14.3.1 (p. 561)
unordered_multiset<Sales_data> SDset;
```

Because `hash<Sales_data>` uses the private members of `Sales_data`, we must make this class a friend of `Sales_data`:

Click here to view code image

```
template <class T> class std::hash;  // needed for the friend declaration
class Sales_data {
friend class std::hash<Sales_data>;
    // other members as before
};
```

Here we say that the specific instantiation of `hash<Sales_data>` is a friend. Because that instantiation is defined in the `std` namespace, we must remember to that this `hash` type is defined in the `std` namespace. Hence, our `friend` declaration refers to `std::hash`.

Note

To enable users of `Sales_data` to use the specialization of `hash`, we should define this specialization in the `Sales_data` header.

Class-Template Partial Specializations

Differently from function templates, a class template specialization does not have to supply an argument for every template parameter. We can specify some, but not all, of the template parameters or some, but not all, aspects of the parameters. A class template **partial specialization** is itself a template. Users must supply arguments for those template parameters that are not fixed by the specialization.

Note

We can partially specialize only a class template. We cannot partially specialize a function template.

In § 16.2.3 (p. 684) we introduced the library `remove_reference` type. That template works through a series of specializations:

Click here to view code image

```
// original, most general template
```

```
template <class T> struct remove_reference {
    typedef T type;
};
// partial specializations that will be used for lvalue and rvalue references
template <class T> struct remove_reference<T&>    // lvalue references
    { typedef T type; };
template <class T> struct remove_reference<T&&>   // rvalue references
    { typedef T type; };
```

The first template defines the most general version. It can be instantiated with any type; it uses its template argument as the type for its member named `type`. The next two classes are partial specializations of this original template.

Because a partial specialization is a template, we start, as usual, by defining the template parameters. Like any other specialization, a partial specialization has the same name as the template it specializes. The specialization's template parameter list includes an entry for each template parameter whose type is not completely fixed by this partial specialization. After the class name, we specify arguments for the template parameters we are specializing. These arguments are listed inside angle brackets following the template name. The arguments correspond positionally to the parameters in the original template.

The template parameter list of a partial specialization is a subset of, or a specialization of, the parameter list of the original template. In this case, the specializations have the same number of parameters as the original template. However, the parameter's type in the specializations differ from the original template. The specializations will be used for lvalue and rvalue reference types, respectively:

Click here to view code image

```
int i;
// decltype(42) is int, uses the original template
remove_reference<decltype(42)>::type a;
// decltype(i) is int&, uses first (T&) partial specialization
remove_reference<decltype(i)>::type b;
// decltype(std::move(i)) is int&&, uses second (i.e., T&&) partial specialization
remove_reference<decltype(std::move(i))>::type c;
```

All three variables, `a`, `b`, and `c`, have type `int`.

Specializing Members but Not the Class

Rather than specializing the whole template, we can specialize just specific member function(s). For example, if `Foo` is a template class with a member `Bar`, we can specialize just that member:

Click here to view code image

```
template <typename T> struct Foo {
    Foo(const T &t = T()): mem(t) { }
    void Bar() { /* ... */ }
    T mem;
    // other members of Foo
};
template<>              // we're specializing a template
void Foo<int>::Bar() // we're specializing the Bar member of Foo<int>
{
     // do whatever specialized processing that applies to ints
}
```

Here we are specializing just one member of the Foo<int> class. The other members of Foo<int> will be supplied by the Foo template:

Click here to view code image

```
Foo<string> fs;  // instantiates Foo<string>::Foo()
fs.Bar();        // instantiates Foo<string>::Bar()
Foo<int> fi;     // instantiates Foo<int>::Foo()
fi.Bar();        // uses our specialization of Foo<int>::Bar()
```

When we use Foo with any type other than int, members are instantiated as usual. When we use Foo with int, members other than Bar are instantiated as usual. If we use the Bar member of Foo<int>, then we get our specialized definition.

Exercises Section 16.5

Exercise 16.62: Define your own version of hash<Sales_data> and define an unordered_multiset of Sales_data objects. Put several transactions into the container and print its contents.

Exercise 16.63: Define a function template to count the number of occurrences of a given value in a vector. Test your program by passing it a vector of doubles, a vector of ints, and a vector of strings.

Exercise 16.64: Write a specialized version of the template from the previous exercise to handle vector<const char*> and a program that uses this specialization.

Exercise 16.65: In § 16.3 (p. 698) we defined overloaded two versions of debug_rep one had a const char* and the other a char* parameter. Rewrite these functions as specializations.

Exercise 16.66: What are the advantages and disadvantages of overloading these debug_rep functions as compared to defining specializations?

Exercise 16.67: Would defining these specializations affect function matching for debug_rep? If so, how? If not, why not?

Chapter Summary

Templates are a distinctive feature of C++ and are fundamental to the library. A template is a blueprint that the compiler uses to generate specific class types or functions. This process is called instantiation. We write the template once, and the compiler instantiates the template for the type(s) or value(s) with which we use the template.

We can define both function templates and class templates. The library algorithms are function templates and the library containers are class templates.

An explicit template argument lets us fix the type or value of one or more template parameters. Normal conversions are applied to parameters that have an explicit template argument.

A template specialization is a user-provided instantiation of a template that binds one or more template parameters to specified types or values. Specializations are useful when there are types that we cannot use (or do not want to use) with the template definition.

A major part of the latest release of the C++ standard is variadic templates. A variadic template can take a varying number and types of parameters. Variadic templates let us write functions, such as the container `emplace` members and the library `make_shared` function, that pass arguments to an object's constructor.

Defined Terms

class template Definition from which specific classes can be instantiated. Class templates are defined using the `template` keyword followed by a comma-separated list of one or more template parameters enclosed in < and > brackets, followed by a class definition.

default template arguments A type or a value that a template uses if the user does not supply a corresponding template argument.

explicit instantiation A declaration that supplies explicit arguments for all the template parameters. Used to guide the instantiation process. If the declaration is `extern`, the template will not be instantiated; otherwise, the template is instantiated with the specified arguments. There must be a non`extern` explicit instantiation somewhere in the program for every `extern` template declaration.

explicit template argument Template argument supplied by the user in a call to a function or when defining a template class type. Explicit template arguments are supplied inside angle brackets immediately following the template's name.

function parameter pack Parameter pack that represents zero or more function parameters.

function template Definition from which specific functions can be instantiated. A function template is defined using the `template` keyword followed by a comma-separated list of one or more template parameters enclosed in < and > brackets, followed by a function definition.

instantiate Compiler process whereby the actual template argument(s) are used to generate a specific instance of the template in which the parameter(s) are replaced by the corresponding argument(s). Functions are instantiated automatically based on the arguments used in a call. We must supply explicit template arguments whenever we use a class template.

instantiation Class or function generated by the compiler from a template.

member template Member function that is a template. A member template may not be virtual.

nontype parameter A template parameter that represents a value. Template arguments for nontype template parameters must be constant expressions.

pack expansion Process by which a parameter pack is replaced by the corresponding list of its elements.

parameter pack Template or function parameter that represents zero or more parameters.

partial specialization Version of a class template in which some some but not all of the template parameters are specified or in which one or more parameters are not completely specified.

pattern Defines the form of each element in an expanded parameter pack.

template argument Type or value used to instantiate a template parameter.

template argument deduction Process by which the compiler determines which function template to instantiate. The compiler examines the types of the arguments that were specified using a template parameter. It automatically instantiates a version of the function with those types or values bound to the template parameters.

template parameter Name specifed in the template parameter list that may be used inside the definition of a template. Template parameters can be type or nontype parameters. To use a class template, we must supply explicit arguments for each template parameter. The compiler uses those types or values to instantiate a version of the class in which uses of the parameter(s) are replaced by the actual argument(s). When a function template is used, the compiler deduces the template arguments from the arguments in the call and instantiates a specific function using the deduced template arguments.

template parameter list List of parameters, separated by commas, to be used in the definition or declaration of a template. Each parameter may be a type or nontype parameter.

template parameter pack Parameter pack that represents zero or more template parameters.

template specialization Redefinition of a class template, a member of a class template, or a function template, in which some (or all) of the template parameters are specified. A template specialization may not appear until after the base template that it specializes has been declared. A template specialization must appear before any use of the template with the specialized arguments. Each template parameter in a function template must be completely specialized.

type parameter Name used in a template parameter list to represent a type. Type parameters are specified following the keyword `typename` or `class`.

type transformation Class templates defined by the library that transform their given template type parameter to a related type.

variadic template Template that takes a varying number of template arguments. A template parameter pack is specified using an elipsis (e.g., `class`. . ., `typename`. . ., or *type-name*. . .).

Part IV: Advanced Topics

Contents

Part IV covers additional features that, although useful in the right context, are not needed by every C++ programmer. These features divide into two clusters: those that are useful for large-scale problems and those that are applicable to specialized problems rather than general ones. Features for specialized problems occur both in the language, the topic of Chapter 19, and in the library, Chapter 17.

In Chapter 17 we cover four special-purpose library facilities: the `bitset` class and three new library facilities: `tuples`, regular expressions, and random numbers. We'll also look at some of the less commonly used parts of the IO library.

Chapter 18 covers exception handling, namespaces, and multiple inheritance. These features tend to be most useful in the context of large-scale problems.

Even programs simple enough to be written by a single author can benefit from