

```

void f()
{
    int *ip = new int(42);      // dynamically allocate a new object
    // code that throws an exception that is not caught inside f
    delete ip;                // free the memory before exiting
}

```

If an exception happens between the `new` and the `delete`, and is not caught inside `f`, then this memory can never be freed. There is no pointer to this memory outside the function `f`. Thus, there is no way to free this memory.



## Smart Pointers and Dumb Classes

Many C++ classes, including all the library classes, define destructors (§ 12.1.1, p. 452) that take care of cleaning up the resources used by that object. However, not all classes are so well behaved. In particular, classes that are designed to be used by both C and C++ generally require the user to specifically free any resources that are used.

Classes that allocate resources—and that do not define destructors to free those resources—can be subject to the same kind of errors that arise when we use dynamic memory. It is easy to forget to release the resource. Similarly, if an exception happens between when the resource is allocated and when it is freed, the program will leak that resource.

We can often use the same kinds of techniques we use to manage dynamic memory to manage classes that do not have well-behaved destructors. For example, imagine we're using a network library that is used by both C and C++. Programs that use this library might contain code such as

```

struct destination; // represents what we are connecting to
struct connection; // information needed to use the connection
connection connect(destination*); // open the connection
void disconnect(connection); // close the given connection
void f(destination &d /* other parameters */)
{
    // get a connection; must remember to close it when done
    connection c = connect(&d);
    // use the connection
    // if we forget to call disconnect before exiting f, there will be no way to close c
}

```

If `connection` had a destructor, that destructor would automatically close the connection when `f` completes. However, `connection` does not have a destructor. This problem is nearly identical to our previous program that used a `shared_ptr` to avoid memory leaks. It turns out that we can also use a `shared_ptr` to ensure that the `connection` is properly closed.



## Using Our Own Deletion Code

By default, `shared_ptrs` assume that they point to dynamic memory. Hence, by default, when a `shared_ptr` is destroyed, it executes `delete` on the pointer it

holds. To use a `shared_ptr` to manage a connection, we must first define a function to use in place of `delete`. It must be possible to call this `deleter` function with the pointer stored inside the `shared_ptr`. In this case, our deleter must take a single argument of type `connection*`:

```
void end_connection(connection *p) { disconnect(*p); }
```

When we create a `shared_ptr`, we can pass an optional argument that points to a deleter function (§ 6.7, p. 247):

```
void f(destination &d /* other parameters */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

When `p` is destroyed, it won't execute `delete` on its stored pointer. Instead, `p` will call `end_connection` on that pointer. In turn, `end_connection` will call `disconnect`, thus ensuring that the connection is closed. If `f` exits normally, then `p` will be destroyed as part of the return. Moreover, `p` will also be destroyed, and the connection will be closed, if an exception occurs.

#### CAUTION: SMART POINTER PITFALLS

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they are used properly. To use smart pointers correctly, we must adhere to a set of conventions:

- Don't use the same built-in pointer value to initialize (or `reset`) more than one smart pointer.
- Don't `delete` the pointer returned from `get()`.
- Don't use `get()` to initialize or `reset` another smart pointer.
- If you use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.
- If you use a smart pointer to manage a resource other than memory allocated by `new`, remember to pass a deleter (§ 12.1.4, p. 468, and § 12.1.5, p. 471).

#### EXERCISES SECTION 12.1.4

**Exercise 12.14:** Write your own version of a function that uses a `shared_ptr` to manage a connection.

**Exercise 12.15:** Rewrite the first exercise to use a lambda (§ 10.3.2, p. 388) in place of the `end_connection` function.

### 12.1.5 unique\_ptr

A `unique_ptr` “owns” the object to which it points. Unlike `shared_ptr`, only one `unique_ptr` at a time can point to a given object. The object to which a `unique_ptr` points is destroyed when the `unique_ptr` is destroyed. Table 12.4 lists the operations specific to `unique_ptr`s. The operations common to both were covered in Table 12.1 (p. 452).

Unlike `shared_ptr`, there is no library function comparable to `make_shared` that returns a `unique_ptr`. Instead, when we define a `unique_ptr`, we bind it to a pointer returned by `new`. As with `shared_ptr`s, we must use the direct form of initialization:

```
unique_ptr<double> p1; // unique_ptr that can point at a double
unique_ptr<int> p2(new int(42)); // p2 points to int with value 42
```

Because a `unique_ptr` owns the object to which it points, `unique_ptr` does not support ordinary copy or assignment:

```
unique_ptr<string> p1(new string("Stegosaurus"));
unique_ptr<string> p2(p1); // error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p2; // error: no assign for unique_ptr
```

**Table 12.4: unique\_ptr Operations (See Also Table 12.1 (p. 452))**

|  |   |
|--|---|
| <code>unique_ptr&lt;T&gt; u1</code>      | Null <code>unique_ptr</code> s that can point to objects of type T. <code>u1</code> will use <code>delete</code> to free its pointer;         |
| <code>unique_ptr&lt;T, D&gt; u2</code>   | <code>u2</code> will use a callable object of type D to free its pointer.   |
| <code>unique_ptr&lt;T, D&gt; u(d)</code> | Null <code>unique_ptr</code> that point to objects of type T that uses d, which must be an object of type D in place of <code>delete</code> . |
| <code>u = nullptr</code>                 | Deletes the object to which <code>u</code> points; makes <code>u</code> null.   |
| <code>u.release()</code>                 | Relinquishes control of the pointer <code>u</code> had held; returns the pointer <code>u</code> had held and makes <code>u</code> null.       |
| <code>u.reset()</code>                   | Deletes the object to which <code>u</code> points;  |
| <code>u.reset(q)</code>                  | If the built-in pointer <code>q</code> is supplied, makes <code>u</code> point to that object.  |
| <code>u.reset(nullptr)</code>            | Otherwise makes <code>u</code> null.  |

Although we can't copy or assign a `unique_ptr`, we can transfer ownership from one (nonconst) `unique_ptr` to another by calling `release` or `reset`:

```
// transfers ownership from p1 (which points to the string Stegosaurus) to p2
unique_ptr<string> p2(p1.release()); // release makes p1 null
unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release()); // reset deletes the memory to which p2 had pointed
```

The `release` member returns the pointer currently stored in the `unique_ptr` and makes that `unique_ptr` null. Thus, `p2` is initialized from the pointer value that had been stored in `p1` and `p1` becomes null.

The `reset` member takes an optional pointer and repositions the `unique_ptr` to point to the given pointer. If the `unique_ptr` is not null, then the object to which the `unique_ptr` had pointed is deleted. The call to `reset` on `p2`, therefore, frees the memory used by the string initialized from "Stegosaurus", transfers `p3`'s pointer to `p2`, and makes `p3` null.

Calling `release` breaks the connection between a `unique_ptr` and the object it had been managing. Often the pointer returned by `release` is used to initialize or assign another smart pointer. In that case, responsibility for managing the memory is simply transferred from one smart pointer to another. However, if we do not use another smart pointer to hold the pointer returned from `release`, our program takes over responsibility for freeing that resource:

```
p2.release(); // WRONG: p2 won't free the memory and we've lost the pointer  
auto p = p2.release(); // ok, but we must remember to delete(p)
```

## Passing and Returning `unique_ptr`s

There is one exception to the rule that we cannot copy a `unique_ptr`: We can copy or assign a `unique_ptr` that is about to be destroyed. The most common example is when we return a `unique_ptr` from a function:

```
unique_ptr<int> clone(int p) {  
    // ok: explicitly create a unique_ptr<int> from int*  
    return unique_ptr<int>(new int(p));  
}
```

Alternatively, we can also return a copy of a local object:

```
unique_ptr<int> clone(int p) {  
    unique_ptr<int> ret(new int (p));  
    // ...  
    return ret;  
}
```

In both cases, the compiler knows that the object being returned is about to be destroyed. In such cases, the compiler does a special kind of "copy" which we'll discuss in § 13.6.2 (p. 534).

### BACKWARD COMPATIBILITY: `AUTO_PTR`

Earlier versions of the library included a class named `auto_ptr` that had some, but not all, of the properties of `unique_ptr`. In particular, it was not possible to store an `auto_ptr` in a container, nor could we return one from a function.

Although `auto_ptr` is still part of the standard library, programs should use `unique_ptr` instead.

## Passing a Deleter to `unique_ptr`

Like `shared_ptr`, by default, `unique_ptr` uses `delete` to free the object to which a `unique_ptr` points. As with `shared_ptr`, we can override the default

deleter in a `unique_ptr` (§ 12.1.4, p. 468). However, for reasons we'll describe in § 16.1.6 (p. 676), the way `unique_ptr` manages its deleter is differs from the way `shared_ptr` does.

Overriding the deleter in a `unique_ptr` affects the `unique_ptr` type as well as how we construct (or `reset`) objects of that type. Similar to overriding the comparison operation of an associative container (§ 11.2.2, p. 425), we must supply the deleter type inside the angle brackets along with the type to which the `unique_ptr` can point. We supply a callable object of the specified type when we create or `reset` an object of this type:

```
// p points to an object of type objT and uses an object of type delT to free that object
// it will call an object named fcn of type delT
unique_ptr<objT, delT> p (new objT, fcn);
```

As a somewhat more concrete example, we'll rewrite our connection program to use a `unique_ptr` in place of a `shared_ptr` as follows:

```
void f(destination &d /* other needed parameters */)
{
    connection c = connect (&d); // open the connection
    // when p is destroyed, the connection will be closed
    unique_ptr<connection, decltype(end_connection)*>
        p(&c, end_connection);
    // use the connection
    // when f exits, even if by an exception, the connection will be properly closed
}
```

Here we use `decltype` (§ 2.5.3, p. 70) to specify the function pointer type. Because `decltype(end_connection)` returns a function type, we must remember to add a `*` to indicate that we're using a pointer to that type (§ 6.7, p. 250).

## EXERCISES SECTION 12.1.5

**Exercise 12.16:** Compilers don't always give easy-to-understand error messages if we attempt to copy or assign a `unique_ptr`. Write a program that contains these errors to see how your compiler diagnoses them.

**Exercise 12.17:** Which of the following `unique_ptr` declarations are illegal or likely to result in subsequent program error? Explain what the problem is with each one.

```
int ix = 1024, *pi = &ix, *pi2 = new int(2048);
typedef unique_ptr<int> IntP;
(a) IntP p0(ix);           (b) IntP p1(pi);
(c) IntP p2(pi2);          (d) IntP p3(&ix);
(e) IntP p4(new int(2048)); (f) IntP p5(p2.get());
```

**Exercise 12.18:** Why doesn't `shared_ptr` have a `release` member?

### 12.1.6 `weak_ptr`



C++  
11

A `weak_ptr` (Table 12.5) is a smart pointer that does not control the lifetime of the object to which it points. Instead, a `weak_ptr` points to an object that is managed by a `shared_ptr`. Binding a `weak_ptr` to a `shared_ptr` does not change the reference count of that `shared_ptr`. Once the last `shared_ptr` pointing to the object goes away, the object itself will be deleted. That object will be deleted even if there are `weak_ptr`s pointing to it—hence the name `weak_ptr`, which captures the idea that a `weak_ptr` shares its object “weakly.”

When we create a `weak_ptr`, we initialize it from a `shared_ptr`:

```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p); // wp weakly shares with p; use count in p is unchanged
```

Here both `wp` and `p` point to the same object. Because the sharing is weak, creating `wp` doesn’t change the reference count of `p`; it is possible that the object to which `wp` points might be deleted.

Because the object might no longer exist, we cannot use a `weak_ptr` to access its object directly. To access that object, we must call `lock`. The `lock` function checks whether the object to which the `weak_ptr` points still exists. If so, `lock` returns a `shared_ptr` to the shared object. As with any other `shared_ptr`, we are guaranteed that the underlying object to which that `shared_ptr` points continues to exist at least as long as that `shared_ptr` exists. For example:

```
if (shared_ptr<int> np = wp.lock()) { // true if np is not null
    // inside the if, np shares its object with p
}
```

Here we enter the body of the `if` only if the call to `lock` succeeds. Inside the `if`, it is safe to use `np` to access that object.

**Table 12.5: `weak_ptr`s**

|                                      |   |
|--------------------------------------|---|
| <code>weak_ptr&lt;T&gt; w</code>     | Null <code>weak_ptr</code> that can point at objects of type <code>T</code> .   |
| <code>weak_ptr&lt;T&gt; w(sp)</code> | <code>weak_ptr</code> that points to the same object as the <code>shared_ptr</code> <code>sp</code> . <code>T</code> must be convertible to the type to which <code>sp</code> points. |
| <code>w = p</code>                   | <code>p</code> can be a <code>shared_ptr</code> or a <code>weak_ptr</code> . After the assignment <code>w</code> shares ownership with <code>p</code> .                               |
| <code>w.reset()</code>               | Makes <code>w</code> null.  |
| <code>w.use_count()</code>           | The number of <code>shared_ptr</code> s that share ownership with <code>w</code> .  |
| <code>w.expired()</code>             | Returns <code>true</code> if <code>w.use_count()</code> is zero, <code>false</code> otherwise.  |
| <code>w.lock()</code>                | If <code>expired</code> is <code>true</code> , returns a null <code>shared_ptr</code> ; otherwise returns a <code>shared_ptr</code> to the object to which <code>w</code> points.     |

### Checked Pointer Class

As an illustration of when a `weak_ptr` is useful, we’ll define a companion pointer class for our `StrBlob` class. Our pointer class, which we’ll name `StrBlobPtr`,

will store a `weak_ptr` to the data member of the `StrBlob` from which it was initialized. By using a `weak_ptr`, we don't affect the lifetime of the vector to which a given `StrBlob` points. However, we can prevent the user from attempting to access a vector that no longer exists.

`StrBlobPtr` will have two data members: `wptr`, which is either null or points to a vector in a `StrBlob`; and `curr`, which is the index of the element that this object currently denotes. Like its companion `StrBlob` class, our pointer class has a `check` member to verify that it is safe to dereference the `StrBlobPtr`:

```
// StrBlobPtr throws an exception on attempts to access a nonexistent element
class StrBlobPtr {
public:
    StrBlobPtr() : curr(0) { }
    StrBlobPtr(StrBlob &a, size_t sz = 0) :
        wptr(a.data), curr(sz) { }
    std::string& deref() const;
    StrBlobPtr& incr();           // prefix version
private:
    // check returns a shared_ptr to the vector if the check succeeds
    std::shared_ptr<std::vector<std::string>>
        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<std::string>> wptr;
    std::size_t curr;             // current position within the array
};
```

The default constructor generates a null `StrBlobPtr`. Its constructor initializer list (§ 7.1.4, p. 265) explicitly initializes `curr` to zero and implicitly initializes `wptr` as a null `weak_ptr`. The second constructor takes a reference to `StrBlob` and an optional index value. This constructor initializes `wptr` to point to the vector in the `shared_ptr` of the given `StrBlob` object and initializes `curr` to the value of `sz`. We use a default argument (§ 6.5.1, p. 236) to initialize `curr` to denote the first element by default. As we'll see, the `sz` parameter will be used by the end member of `StrBlob`.

It is worth noting that we cannot bind a `StrBlobPtr` to a `const StrBlob` object. This restriction follows from the fact that the constructor takes a reference to a nonconst object of type `StrBlob`.

The `check` member of `StrBlobPtr` differs from the one in `StrBlob` because it must check whether the vector to which it points is still around:

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i, const std::string &msg) const
{
    auto ret = wptr.lock();    // is the vector still around?
    if (!ret)
        throw std::runtime_error("unbound StrBlobPtr");
    if (i >= ret->size())
        throw std::out_of_range(msg);
    return ret; // otherwise, return a shared_ptr to the vector
}
```

Because a `weak_ptr` does not participate in the reference count of its corresponding `shared_ptr`, the vector to which this `StrBlobPtr` points might have been deleted. If the vector is gone, `lock` will return a null pointer. In this case, any reference to the vector will fail, so we throw an exception. Otherwise, `check` verifies its given index. If that value is okay, `check` returns the `shared_ptr` it obtained from `lock`.

## Pointer Operations

We'll learn how to define our own operators in Chapter 14. For now, we've defined functions named `deref` and `incr` to dereference and increment the `StrBlobPtr`, respectively. The `deref` member calls `check` to verify that it is safe to use the vector and that `curr` is in range:

```
std::string& StrBlobPtr::deref() const
{
    auto p = check(curr, "dereference past end");
    return (*p)[curr]; // (*p) is the vector to which this object points
}
```

If `check` succeeds, `p` is a `shared_ptr` to the vector to which this `StrBlobPtr` points. The expression `(*p)[curr]` dereferences that `shared_ptr` to get the vector and uses the subscript operator to fetch and return the element at `curr`.

The `incr` member also calls `check`:

```
// prefix: return a reference to the incremented object
StrBlobPtr& StrBlobPtr::incr()
{
    // 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;
}
```

We'll also give our `StrBlob` class `begin` and `end` operations. These members will return `StrBlobPtr`s pointing to the first or one past the last element in the `StrBlob` itself. In addition, because `StrBlobPtr` accesses the `data` member of `StrBlob`, we must also make `StrBlobPtr` a friend of `StrBlob` (§ 7.3.4, p. 279):

```
class StrBlob {
    friend class StrBlobPtr;
    // other members as in § 12.1.1 (p. 456)
    StrBlobPtr begin(); // return StrBlobPtr to the first element
    StrBlobPtr end(); // and one past the last element
};
// these members can't be defined until StrStrBlob and StrStrBlobPtr are defined
StrBlobPtr StrBlob::begin() { return StrBlobPtr(*this); }
StrBlobPtr StrBlob::end()
{ return StrBlobPtr(*this, data->size()); }
```

## EXERCISES SECTION 12.1.6

**Exercise 12.19:** Define your own version of `StrBlobPtr` and update your `StrBlob` class with the appropriate friend declaration and begin and end members.

**Exercise 12.20:** Write a program that reads an input file a line at a time into a `StrBlob` and uses a `StrBlobPtr` to print each element in that `StrBlob`.

**Exercise 12.21:** We could have written `StrBlobPtr`'s `deref` member as follows:

```
std::string& deref() const
{ return (*check(curr, "dereference past end"))[curr]; }
```

Which version do you think is better and why?

**Exercise 12.22:** What changes would need to be made to `StrBlobPtr` to create a class that can be used with a `const StrBlob`? Define a class named `ConstStrBlobPtr` that can point to a `const StrBlob`.



## 12.2 Dynamic Arrays

The `new` and `delete` operators allocate objects one at a time. Some applications, need the ability to allocate storage for many objects at once. For example, `vectors` and `strings` store their elements in contiguous memory and must allocate several elements at once whenever the container has to be reallocated (§ 9.4, p. 355).

To support such usage, the language and library provide two ways to allocate an array of objects at once. The language defines a second kind of new expression that allocates and initializes an array of objects. The library includes a template class named `allocator` that lets us separate allocation from initialization. For reasons we'll explain in § 12.2.2 (p. 481), using an `allocator` generally provides better performance and more flexible memory management.

Many, perhaps even most, applications have no direct need for dynamic arrays. When an application needs a varying number of objects, it is almost always easier, faster, and safer to do as we did with `StrBlob`: use a `vector` (or other library container). For reasons we'll explain in § 13.6 (p. 531), the advantages of using a library container are even more pronounced under the new standard. Libraries that support the new standard tend to be dramatically faster than previous releases.



Most applications should use a library container rather than dynamically allocated arrays. Using a container is easier, less likely to contain memory-management bugs, and is likely to give better performance.

As we've seen, classes that use the containers can use the default versions of the operations for copy, assignment, and destruction (§ 7.1.5, p. 267). Classes that allocate dynamic arrays must define their own versions of these operations to manage the associated memory when objects are copied, assigned, and destroyed.



Do not allocate dynamic arrays in code inside classes until you have read Chapter 13.

## 12.2.1 new and Arrays



We ask `new` to allocate an array of objects by specifying the number of objects to allocate in a pair of square brackets after a type name. In this case, `new` allocates the requested number of objects and (assuming the allocation succeeds) returns a pointer to the first one:

```
// call get_size to determine how many ints to allocate
int *pia = new int[get_size()]; // pia points to the first of these ints
```

The size inside the brackets must have integral type but need not be a constant.

We can also allocate an array by using a type alias (§ 2.5.1, p. 67) to represent an array type. In this case, we omit the brackets:

```
typedef int arrT[42]; // arrT names the type array of 42 ints
int *p = new arrT; // allocates an array of 42 ints; p points to the first one
```

Here, `new` allocates an array of `ints` and returns a pointer to the first one. Even though there are no brackets in our code, the compiler executes this expression using `new []`. That is, the compiler executes this expression as if we had written

```
int *p = new int[42];
```

### Allocating an Array Yields a Pointer to the Element Type

Although it is common to refer to memory allocated by `new T []` as a “dynamic array,” this usage is somewhat misleading. When we use `new` to allocate an array, we do not get an object with an array type. Instead, we get a pointer to the element type of the array. Even if we use a type alias to define an array type, `new` does not allocate an object of array type. In this case, the fact that we’re allocating an array is not even visible; there is no `[num]`. Even so, `new` returns a pointer to the element type.

Because the allocated memory does not have an array type, we cannot call `begin` or `end` (§ 3.5.3, p. 118) on a dynamic array. These functions use the array dimension (which is part of an array’s type) to return pointers to the first and one past the last elements, respectively. For the same reasons, we also cannot use a range `for` to process the elements in a (so-called) dynamic array.

C++  
11



It is important to remember that what we call a dynamic array does not have an array type.

### Initializing an Array of Dynamically Allocated Objects

By default, objects allocated by `new`—whether allocated as a single object or in an array—are default initialized. We can value initialize (§ 3.3.1, p. 98) the elements in an array by following the size with an empty pair of parentheses.

```
int *pia = new int[10]; // block of ten uninitialized ints
int *pia2 = new int[10](); // block of ten ints value initialized to 0
string *psa = new string[10]; // block of ten empty strings
string *psa2 = new string[10](); // block of ten empty strings
```

**C++ 11**

Under the new standard, we can also provide a braced list of element initializers:

```
// block of ten ints each initialized from the corresponding initializer
int *pia3 = new int[10]{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
// block of ten strings; the first four are initialized from the given initializers
// remaining elements are value initialized
string *psa3 = new string[10]{"a", "an", "the", string(3, 'x')};
```

As when we list initialize an object of built-in array type (§ 3.5.1, p. 114), the initializers are used to initialize the first elements in the array. If there are fewer initializers than elements, the remaining elements are value initialized. If there are more initializers than the given size, then the `new` expression fails and no storage is allocated. In this case, `new` throws an exception of type `bad_array_new_length`. Like `bad_alloc`, this type is defined in the `new` header.

**C++ 11**

Although we can use empty parentheses to value initialize the elements of an array, we cannot supply an element initializer inside the parentheses. The fact that we cannot supply an initial value inside the parentheses means that we cannot use `auto` to allocate an array (§ 12.1.2, p. 459).

## It Is Legal to Dynamically Allocate an Empty Array

We can use an arbitrary expression to determine the number of objects to allocate:

```
size_t n = get_size(); // get_size returns the number of elements needed
int* p = new int[n]; // allocate an array to hold the elements
for (int* q = p; q != p + n; ++q)
    /* process the array */;
```

An interesting question arises: What happens if `get_size` returns 0? The answer is that our code works fine. Calling `new[n]` with `n` equal to 0 is legal even though we cannot create an array variable of size 0:

```
char arr[0]; // error: cannot define a zero-length array
char *cp = new char[0]; // ok: but cp can't be dereferenced
```

When we use `new` to allocate an array of size zero, `new` returns a valid, nonzero pointer. That pointer is guaranteed to be distinct from any other pointer returned by `new`. This pointer acts as the off-the-end pointer (§ 3.5.3, p. 119) for a zero-element array. We can use this pointer in ways that we use an off-the-end iterator. The pointer can be compared as in the loop above. We can add zero to (or subtract zero from) such a pointer and can subtract the pointer from itself, yielding zero. The pointer cannot be dereferenced—after all, it points to no element.

In our hypothetical loop, if `get_size` returns 0, then `n` is also 0. The call to `new` will allocate zero objects. The condition in the `for` will fail (`p` is equal to `q + n` because `n` is 0). Thus, the loop body is not executed.

## Freeing Dynamic Arrays

To free a dynamic array, we use a special form of `delete` that includes an empty pair of square brackets:

```
delete p;      // p must point to a dynamically allocated object or be null
delete [] pa; // pa must point to a dynamically allocated array or be null
```

The second statement destroys the elements in the array to which `pa` points and frees the corresponding memory. Elements in an array are destroyed in reverse order. That is, the last element is destroyed first, then the second to last, and so on.

When we delete a pointer to an array, the empty bracket pair is essential: It indicates to the compiler that the pointer addresses the first element of an array of objects. If we omit the brackets when we delete a pointer to an array (or provide them when we delete a pointer to an object), the behavior is undefined.

Recall that when we use a type alias that defines an array type, we can allocate an array without using `[]` with `new`. Even so, we must use brackets when we delete a pointer to that array:

```
typedef int arrT[42]; // arrT names the type array of 42 ints
int *p = new arrT;    // allocates an array of 42 ints; p points to the first one
delete [] p;          // brackets are necessary because we allocated an array
```

Despite appearances, `p` points to the first element of an array of objects, not to a single object of type `arrT`. Thus, we must use `[]` when we delete `p`.



The compiler is unlikely to warn us if we forget the brackets when we delete a pointer to an array or if we use them when we delete a pointer to an object. Instead, our program is apt to misbehave without warning during execution.

## Smart Pointers and Dynamic Arrays

The library provides a version of `unique_ptr` that can manage arrays allocated by `new`. To use a `unique_ptr` to manage a dynamic array, we must include a pair of empty brackets after the object type:

```
// up points to an array of ten uninitialized ints
unique_ptr<int []> up(new int [10]);
up.release(); // automatically uses delete [] to destroy its pointer
```

The brackets in the type specifier (`<int []>`) say that `up` points not to an `int` but to an array of `ints`. Because `up` points to an array, when `up` destroys the pointer it manages, it will automatically use `delete []`.

`unique_ptr`s that point to arrays provide slightly different operations than those we used in § 12.1.5 (p. 470). These operations are described in Table 12.6 (overleaf). When a `unique_ptr` points to an array, we cannot use the dot and arrow member access operators. After all, the `unique_ptr` points to an array, not an object so these operators would be meaningless. On the other hand, when a `unique_ptr` points to an array, we can use the subscript operator to access the elements in the array:

```
for (size_t i = 0; i != 10; ++i)
    up[i] = i; // assign a new value to each of the elements
```

**Table 12.6: unique\_ptrs to Arrays**

**Member access operators (dot and arrow) are not supported for unique\_ptrs to arrays.**  
**Other unique\_ptr operations unchanged.**

|   |   |
|---|---|
| <code>unique_ptr&lt;T[]&gt; u</code>    | u can point to a dynamically allocated array of type T.   |
| <code>unique_ptr&lt;T[]&gt; u(p)</code> | u points to the dynamically allocated array to which the built-in pointer p points. p must be convertible to T* (§ 4.11.2, p. 161). |
| <code>u[i]</code>                       | Returns the object at position i in the array that u owns.<br><b>u must point to an array.</b>                                      |

Unlike `unique_ptr`, `shared_ptr`s provide no direct support for managing a dynamic array. If we want to use a `shared_ptr` to manage a dynamic array, we must provide our own deleter:

```
// to use a shared_ptr we must supply a deleter
shared_ptr<int> sp(new int[10], [](int *p) { delete[] p; });
sp.reset(); // uses the lambda we supplied that uses delete[] to free the array
```

Here we pass a lambda (§ 10.3.2, p. 388) that uses `delete[]` as the deleter.

Had we neglected to supply a deleter, this code would be undefined. By default, `shared_ptr` uses `delete` to destroy the object to which it points. If that object is a dynamic array, using `delete` has the same kinds of problems that arise if we forget to use `[]` when we delete a pointer to a dynamic array (§ 12.2.1, p. 479).

The fact that `shared_ptr` does not directly support managing arrays affects how we access the elements in the array:

```
// shared_ptrs don't have subscript operator and don't support pointer arithmetic
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i; // use get to get a built-in pointer
```

There is no subscript operator for `shared_ptr`s, and the smart pointer types do not support pointer arithmetic. As a result, to access the elements in the array, we must use `get` to obtain a built-in pointer, which we can then use in normal ways.

## EXERCISES SECTION 12.2.1

**Exercise 12.23:** Write a program to concatenate two string literals, putting the result in a dynamically allocated array of `char`. Write a program to concatenate two library strings that have the same value as the literals used in the first program.

**Exercise 12.24:** Write a program that reads a string from the standard input into a dynamically allocated character array. Describe how your program handles varying size inputs. Test your program by giving it a string of data that is longer than the array size you've allocated.

**Exercise 12.25:** Given the following new expression, how would you `delete` pa?

```
int *pa = new int[10];
```



## 12.2.2 The allocator Class

An aspect of new that limits its flexibility is that new combines allocating memory with constructing object(s) in that memory. Similarly, delete combines destruction with deallocation. Combining initialization with allocation is usually what we want when we allocate a single object. In that case, we almost certainly know the value the object should have.

When we allocate a block of memory, we often plan to construct objects in that memory as needed. In this case, we'd like to decouple memory allocation from object construction. Decoupling construction from allocation means that we can allocate memory in large chunks and pay the overhead of constructing the objects only when we actually need to create them.

In general, coupling allocation and construction can be wasteful. For example:

```
string *const p = new string[n]; // construct n empty strings
string s;
string *q = p; // q points to the first string
while (cin >> s && q != p + n)
    *q++ = s; // assign a new value to *q
const size_t size = q - p; // remember how many strings we read
// use the array
delete[] p; // p points to an array; must remember to use delete[]
```

This new expression allocates and initializes n strings. However, we might not need n strings; a smaller number might suffice. As a result, we may have created objects that are never used. Moreover, for those objects we do use, we immediately assign new values over the previously initialized strings. The elements that are used are written twice: first when the elements are default initialized, and subsequently when we assign to them.

More importantly, classes that do not have default constructors cannot be dynamically allocated as an array.

## The allocator Class

The library **allocator** class, which is defined in the `memory` header, lets us separate allocation from construction. It provides type-aware allocation of raw, unconstructed, memory. Table 12.7 (overleaf) outlines the operations that allocator supports. In this section, we'll describe the allocator operations. In § 13.5 (p. 524), we'll see an example of how this class is typically used.

Like `vector`, `allocator` is a template (§ 3.3, p. 96). To define an allocator we must specify the type of objects that a particular allocator can allocate. When an allocator object allocates memory, it allocates memory that is appropriately sized and aligned to hold objects of the given type:

```
allocator<string> alloc; // object that can allocate strings
auto const p = alloc.allocate(n); // allocate n unconstructed strings
```

This call to `allocate` allocates memory for n strings.

**Table 12.7: Standard allocator Class and Customized Algorithms**

|                                   |   |
|-----------------------------------|---|
| <code>allocator&lt;T&gt; a</code> | Defines an allocator object named <code>a</code> that can allocate memory for objects of type <code>T</code> .  |
| <code>a.allocate(n)</code>        | Allocates raw, unconstructed memory to hold <code>n</code> objects of type <code>T</code> .   |
| <code>a.deallocate(p, n)</code>   | Deallocates memory that held <code>n</code> objects of type <code>T</code> starting at the address in the <code>T*</code> pointer <code>p</code> ; <code>p</code> must be a pointer previously returned by <code>allocate</code> , and <code>n</code> must be the size requested when <code>p</code> was created. The user must run <code>destroy</code> on any objects that were constructed in this memory before calling <code>deallocate</code> . |
| <code>a.construct(p, args)</code> | <code>p</code> must be a pointer to type <code>T</code> that points to raw memory; <code>args</code> are passed to a constructor for type <code>T</code> , which is used to construct an object in the memory pointed to by <code>p</code> .  |
| <code>a.destroy(p)</code>         | Runs the destructor (§ 12.1.1, p. 452) on the object pointed to by the <code>T*</code> pointer <code>p</code> .   |

### allocators Allocate Unconstructed Memory

The memory an allocator allocates is *unconstructed*. We use this memory by constructing objects in that memory. In the new library the `construct` member takes a pointer and zero or more additional arguments; it constructs an element at the given location. The additional arguments are used to initialize the object being constructed. Like the arguments to `make_shared` (§ 12.1.1, p. 451), these additional arguments must be valid initializers for an object of the type being constructed. In particular, if the `,` object is a class type, these arguments must match a constructor for that class:

```
auto q = p; // q will point to one past the last constructed element
alloc.construct(q++); // *q is the empty string
alloc.construct(q++, 10, 'c'); // *q is cccccccccc
alloc.construct(q++, "hi"); // *q is hi!
```

In earlier versions of the library, `construct` took only two arguments: the pointer at which to construct an object and a value of the element type. As a result, we could only copy an element into unconstructed space, we could not use any other constructor for the element type.

It is an error to use raw memory in which an object has not been constructed:

```
cout << *p << endl; // ok: uses the string output operator
cout << *q << endl; // disaster: q points to unconstructed memory!
```



We must construct objects in order to use memory returned by `allocate`. Using unconstructed memory in other ways is undefined.

When we're finished using the objects, we must destroy the elements we constructed, which we do by calling `destroy` on each constructed element. The `destroy` function takes a pointer and runs the destructor (§ 12.1.1, p. 452) on the pointed-to object:

```
while (q != p)
    alloc.destroy(--q);           // free the strings we actually allocated
```

At the beginning of our loop, `q` points one past the last constructed element. We decrement `q` before calling `destroy`. Thus, on the first call to `destroy`, `q` points to the last constructed element. We destroy the first element in the last iteration, after which `q` will equal `p` and the loop ends.



We may destroy only elements that are actually constructed.

**WARNING**

Once the elements have been destroyed, we can either reuse the memory to hold other strings or return the memory to the system. We free the memory by calling `deallocate`:

```
alloc.deallocate(p, n);
```

The pointer we pass to `deallocate` cannot be null; it must point to memory allocated by `allocate`. Moreover, the size argument passed to `deallocate` must be the same size as used in the call to `allocate` that obtained the memory to which the pointer points.

## Algorithms to Copy and Fill Uninitialized Memory

As a companion to the `allocator` class, the library also defines two algorithms that can construct objects in uninitialized memory. These functions, described in Table 12.8, are defined in the `memory` header.

**Table 12.8: allocator Algorithms**

**These functions construct elements in the destination, rather than assigning to them.**

`uninitialized_copy(b, e, b2)`

Copies elements from the input range denoted by iterators `b` and `e` into unconstructed, raw memory denoted by the iterator `b2`. The memory denoted by `b2` must be large enough to hold a copy of the elements in the input range.

`uninitialized_copy_n(b, n, b2)`

Copies `n` elements starting from the one denoted by the iterator `b` into raw memory starting at `b2`.

`uninitialized_fill(b, e, t)`

Constructs objects in the range of raw memory denoted by iterators `b` and `e` as a copy of `t`.

`uninitialized_fill_n(b, n, t)`

Constructs an unsigned number `n` objects starting at `b`. `b` must denote unconstructed, raw memory large enough to hold the given number of objects.

As an example, assume we have a `vector` of `ints` that we want to copy into dynamic memory. We'll allocate memory for twice as many `ints` as are in the `vector`. We'll construct the first half of the newly allocated memory by copying elements from the original `vector`. We'll construct elements in the second half by filling them with a given value:

```
// allocate twice as many elements as vi holds
auto p = alloc.allocate(vi.size() * 2);
// construct elements starting at p as copies of elements in vi
auto q = uninitialized_copy(vi.begin(), vi.end(), p);
// initialize the remaining elements to 42
uninitialized_fill_n(q, vi.size(), 42);
```

Like the `copy` algorithm (§ 10.2.2, p. 382), `uninitialized_copy` takes three iterators. The first two denote an input sequence and the third denotes the destination into which those elements will be copied. The destination iterator passed to `uninitialized_copy` must denote unconstructed memory. Unlike `copy`, `uninitialized_copy` constructs elements in its destination.

Like `copy`, `uninitialized_copy` returns its (incremented) destination iterator. Thus, a call to `uninitialized_copy` returns a pointer positioned one element past the last constructed element. In this example, we store that pointer in `q`, which we pass to `uninitialized_fill_n`. This function, like `fill_n` (§ 10.2.2, p. 380), takes a pointer to a destination, a count, and a value. It will construct the given number of objects from the given value at locations starting at the given destination.

### EXERCISES SECTION 12.2.2

**Exercise 12.26:** Rewrite the program on page 481 using an allocator.



## 12.3 Using the Library: A Text-Query Program

To conclude our discussion of the library, we'll implement a simple text-query program. Our program will let a user search a given file for words that might occur in it. The result of a query will be the number of times the word occurs and a list of lines on which that word appears. If a word occurs more than once on the same line, we'll display that line only once. Lines will be displayed in ascending order—that is, line 7 should be displayed before line 9, and so on.

For example, we might read the file that contains the input for this chapter and look for the word `element`. The first few lines of the output would be

```
element occurs 112 times
(line 36) A set element contains only a key;
(line 158) operator creates a new element
(line 160) Regardless of whether the element
(line 168) When we fetch an element from a map, we
(line 214) If the element is not found, find returns
```

followed by the remaining 100 or so lines in which the word `element` occurs.

### 12.3.1 Design of the Query Program



A good way to start the design of a program is to list the program's operations. Knowing what operations we need can help us see what data structures we'll need. Starting from requirements, the tasks our program must do include the following:

- When it reads the input, the program must remember the line(s) in which each word appears. Hence, the program will need to read the input a line at a time and break up the lines from the input file into its separate words
- When it generates output,
  - The program must be able to fetch the line numbers associated with a given word
  - The line numbers must appear in ascending order with no duplicates
  - The program must be able to print the text appearing in the input file at a given line number.

These requirements can be met quite neatly by using various library facilities:

- We'll use a `vector<string>` to store a copy of the entire input file. Each line in the input file will be an element in this `vector`. When we want to print a line, we can fetch the line using its line number as the index.
- We'll use an `istringstream` (§ 8.3, p. 321) to break each line into words.
- We'll use a `set` to hold the line numbers on which each word in the input appears. Using a `set` guarantees that each line will appear only once and that the line numbers will be stored in ascending order.
- We'll use a `map` to associate each word with the `set` of line numbers on which the word appears. Using a `map` will let us fetch the `set` for any given word.

For reasons we'll explain shortly, our solution will also use `shared_ptr`s.

## Data Structures

Although we could write our program using `vector`, `set`, and `map` directly, it will be more useful if we define a more abstract solution. We'll start by designing a class to hold the input file in a way that makes querying the file easy. This class, which we'll name `TextQuery`, will hold a `vector` and a `map`. The `vector` will hold the text of the input file; the `map` will associate each word in that file to the `set` of line numbers on which that word appears. This class will have a constructor that reads a given input file and an operation to perform the queries.

The work of the query operation is pretty simple: It will look inside its `map` to see whether the given word is present. The hard part in designing this function is deciding what the query function should return. Once we know that a word was found, we need to know how often it occurred, the line numbers on which it occurred, and the corresponding text for each of those line numbers.

The easiest way to return all those data is to define a second class, which we'll name `QueryResult`, to hold the results of a query. This class will have a `print` function to print the results in a `QueryResult`.

## Sharing Data between Classes

Our `QueryResult` class is intended to represent the results of a query. Those results include the set of line numbers associated with the given word and the corresponding lines of text from the input file. These data are stored in objects of type `TextQuery`.

Because the data that a `QueryResult` needs are stored in a `TextQuery` object, we have to decide how to access them. We could copy the set of line numbers, but that might be an expensive operation. Moreover, we certainly wouldn't want to copy the vector, because that would entail copying the entire file in order to print (what will usually be) a small subset of the file.

We could avoid making copies by returning iterators (or pointers) into the `TextQuery` object. However, this approach opens up a pitfall: What happens if the `TextQuery` object is destroyed before a corresponding `QueryResult`? In that case, the `QueryResult` would refer to data in an object that no longer exists.

This last observation about synchronizing the lifetime of a `QueryResult` with the `TextQuery` object whose results it represents suggests a solution to our design problem. Given that these two classes conceptually “share” data, we'll use `shared_ptr` (§ 12.1.1, p. 450) to reflect that sharing in our data structures.

## Using the `TextQuery` Class

When we design a class, it can be helpful to write programs using the class before actually implementing the members. That way, we can see whether the class has the operations we need. For example, the following program uses our proposed `TextQuery` and `QueryResult` classes. This function takes an `ifstream` that points to the file we want to process, and interacts with a user, printing the results for the given words:

```
void runQueries(ifstream &infile)
{
    // infile is an ifstream that is the file we want to query
    TextQuery tq(infile); // store the file and build the query map
    // iterate with the user: prompt for a word to find and print results
    while (true) {
        cout << "enter word to look for, or q to quit: ";
        string s;
        // stop if we hit end-of-file on the input or if a 'q' is entered
        if (!(cin >> s) || s == "q") break;
        // run the query and print the results
        print(cout, tq.query(s)) << endl;
    }
}
```

We start by initializing a `TextQuery` object named `tq` from a given `ifstream`. The `TextQuery` constructor reads that file into its vector and builds the map that associates the words in the input with the line numbers on which they appear.

The `while` loop iterates (indefinitely) with the user asking for a word to query and printing the related results. The loop condition tests the literal `true` (§ 2.1.3, p. 41), so it always succeeds. We exit the loop through the `break` (§ 5.5.1, p. 190)

after the first `if`. That `if` checks that the read succeeded. If so, it also checks whether the user entered a `q` to quit. Once we have a word to look for, we ask `tq` to find that word and then call `print` to print the results of the search.

### EXERCISES SECTION 12.3.1

**Exercise 12.27:** The `TextQuery` and `QueryResult` classes use only capabilities that we have already covered. Without looking ahead, write your own versions of these classes.

**Exercise 12.28:** Write a program to implement text queries without defining classes to manage the data. Your program should take a file and interact with a user to query for words in that file. Use `vector`, `map`, and `set` containers to hold the data for the file and to generate the results for the queries.

**Exercise 12.29:** We could have written the loop to manage the interaction with the user as a `do while` (§ 5.4.4, p. 189) loop. Rewrite the loop to use a `do while`. Explain which version you prefer and why.

### 12.3.2 Defining the Query Program Classes



We'll start by defining our `TextQuery` class. The user will create objects of this class by supplying an `istream` from which to read the input file. This class also provides the `query` operation that will take a `string` and return a `QueryResult` representing the lines on which that `string` appears.

The data members of the class have to take into account the intended sharing with `QueryResult` objects. The `QueryResult` class will share the `vector` representing the input file and the sets that hold the line numbers associated with each word in the input. Hence, our class has two data members: a `shared_ptr` to a dynamically allocated `vector` that holds the input file, and a `map` from `string` to `shared_ptr<set>`. The `map` associates each word in the file with a dynamically allocated `set` that holds the line numbers on which that word appears.

To make our code a bit easier to read, we'll also define a type member (§ 7.3.1, p. 271) to refer to line numbers, which are indices into a `vector` of `strings`:

```
class QueryResult; // declaration needed for return type in the query function
class TextQuery {
public:
    using line_no = std::vector<std::string>::size_type;
    TextQuery(std::ifstream&);
    QueryResult query(const std::string&) const;
private:
    std::shared_ptr<std::vector<std::string>> file; // input file
    // map of each word to the set of the lines in which that word appears
    std::map<std::string,
            std::shared_ptr<std::set<line_no>>> wm;
};
```

The hardest part about this class is untangling the class names. As usual, for code that will go in a header file, we use `std::` when we use a library name (§ 3.1, p. 83). In this case, the repeated use of `std::` makes the code a bit hard to read at first. For example,

```
std::map<std::string, std::shared_ptr<std::set<line_no>>> wm;
```

is easier to understand when rewritten as

```
map<string, shared_ptr<set<line_no>>> wm;
```

## The TextQuery Constructor

The `TextQuery` constructor takes an `ifstream`, which it reads a line at a time:

```
// read the input file and build the map of lines to line numbers
TextQuery::TextQuery(ifstream &is): file(new vector<string>)
{
    string text;
    while (getline(is, text)) {           // for each line in the file
        file->push_back(text);          // remember this line of text
        int n = file->size() - 1;        // the current line number
        istringstream line(text);         // separate the line into words
        string word;
        while (line >> word) {          // for each word in that line
            // if word isn't already in wm, subscripting adds a new entry
            auto &lines = wm[word]; // lines is a shared_ptr
            if (!lines) // that pointer is null the first time we see word
                lines.reset(new set<line_no>); // allocate a new set
            lines->insert(n);           // insert this line number
        }
    }
}
```

The constructor initializer allocates a new `vector` to hold the text from the input file. We use `getline` to read the file a line at a time and push each line onto the `vector`. Because `file` is a `shared_ptr`, we use the `->` operator to dereference `file` to fetch the `push_back` member of the `vector` to which `file` points.

Next we use an `istringstream` (§ 8.3, p. 321) to process each word in the line we just read. The inner `while` uses the `istringstream` input operator to read each word from the current line into `word`. Inside the `while`, we use the `map` subscript operator to fetch the `shared_ptr<set>` associated with `word` and bind `lines` to that pointer. Note that `lines` is a reference, so changes made to `lines` will be made to the element in `wm`.

If `word` wasn't in the `map`, the subscript operator adds `word` to `wm` (§ 11.3.4, p. 435). The element associated with `word` is value initialized, which means that `lines` will be a null pointer if the subscript operator added `word` to `wm`. If `lines` is null, we allocate a new `set` and call `reset` to update the `shared_ptr` to which `lines` refers to point to this newly allocated `set`.

Regardless of whether we created a new `set`, we call `insert` to add the current line number. Because `lines` is a reference, the call to `insert` adds an element

to the set in `wm`. If a given word occurs more than once in the same line, the call to `insert` does nothing.

## The `QueryResult` Class

The `QueryResult` class has three data members: a `string` that is the word whose results it represents; a `shared_ptr` to the vector containing the input file; and a `shared_ptr` to the set of line numbers on which this word appears. Its only member function is a constructor that initializes these three members:

```
class QueryResult {
    friend std::ostream& print(std::ostream&, const QueryResult&);
public:
    QueryResult(std::string s,
                std::shared_ptr<std::set<line_no>> p,
                std::shared_ptr<std::vector<std::string>> f) :
        sought(s), lines(p), file(f) { }
private:
    std::string sought; // word this query represents
    std::shared_ptr<std::set<line_no>> lines; // lines it's on
    std::shared_ptr<std::vector<std::string>> file; // input file
};
```

The constructor's only job is to store its arguments in the corresponding data members, which it does in the constructor initializer list (§ 7.1.4, p. 265).

## The `query` Function

The `query` function takes a `string`, which it uses to locate the corresponding set of line numbers in the map. If the `string` is found, the `query` function constructs a `QueryResult` from the given `string`, the `TextQuery` `file` member, and the `set` that was fetched from `wm`.

The only question is: What should we return if the given `string` is not found? In this case, there is no `set` to return. We'll solve this problem by defining a local `static` object that is a `shared_ptr` to an empty set of line numbers. When the word is not found, we'll return a copy of this `shared_ptr`:

```
QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}
```

## Printing the Results

The print function prints its given `QueryResult` object on its given stream:

```
ostream &print(ostream & os, const QueryResult &qr)
{
    // if the word was found, print the count and all occurrences
    os << qr.sought << " occurs " << qr.lines->size() << " "
        << make_plural(qr.lines->size(), "time", "s") << endl;
    // print each line in which the word appeared
    for (auto num : *qr.lines) // for every element in the set
        // don't confound the user with text lines starting at 0
        os << "\t(line " << num + 1 << ") "
            << *(qr.file->begin() + num) << endl;
    return os;
}
```

We use the `size` of the set to which the `qr.lines` points to report how many matches were found. Because that set is in a `shared_ptr`, we have to remember to dereference `lines`. We call `make_plural` (§ 6.3.2, p. 224) to print `time` or `times`, depending on whether that size is equal to 1.

In the `for` we iterate through the set to which `lines` points. The body of the `for` prints the line number, adjusted to use human-friendly counting. The numbers in the set are indices of elements in the vector, which are numbered from zero. However, most users think of the first line as line number 1, so we systematically add 1 to the line numbers to convert to this more common notation.

We use the line number to fetch a line from the vector to which `file` points. Recall that when we add a number to an iterator, we get the element that many elements further into the vector (§ 3.4.2, p. 111). Thus, `file->begin() + num` is the `num`th element after the start of the vector to which `file` points.

Note that this function correctly handles the case that the word is not found. In this case, the set will be empty. The first output statement will note that the word occurred 0 times. Because `*res.lines` is empty, the `for` loop won't be executed.

### EXERCISES SECTION 12.3.2

**Exercise 12.30:** Define your own versions of the `TextQuery` and `QueryResult` classes and execute the `runQueries` function from § 12.3.1 (p. 486).

**Exercise 12.31:** What difference(s) would it make if we used a `vector` instead of a `set` to hold the line numbers? Which approach is better? Why?

**Exercise 12.32:** Rewrite the `TextQuery` and `QueryResult` classes to use a `StrBlob` instead of a `vector<string>` to hold the input file.

**Exercise 12.33:** In Chapter 15 we'll extend our query system and will need some additional members in the `QueryResult` class. Add members named `begin` and `end` that return iterators into the set of line numbers returned by a given query, and a member named `get_file` that returns a `shared_ptr` to the file in the `QueryResult` object.

## CHAPTER SUMMARY

---

In C++, memory is allocated through new expressions and freed through delete expressions. The library also defines an allocator class for allocating blocks of dynamic memory.

Programs that allocate dynamic memory are responsible for freeing the memory they allocate. Properly freeing dynamic memory is a rich source of bugs: Either the memory is never freed, or it is freed while there are still pointers referring to the memory. The new library defines smart pointers—shared\_ptr, unique\_ptr, and weak\_ptr—that make managing dynamic memory much safer. A smart pointer automatically frees the memory once there are no other users of that memory. When possible, modern C++ programs ought to use smart pointers.

## DEFINED TERMS

---

**allocator** Library class that allocates unconstructed memory.

**dangling pointer** A pointer that refers to memory that once had an object but no longer does. Program errors due to dangling pointers are notoriously difficult to debug.

**delete** Frees memory allocated by new. delete p frees the object and delete [] p frees the array to which p points. p may be null or point to memory allocated by new.

**deleter** Function passed to a smart pointer to use in place of delete when destroying the object to which the pointer is bound.

**destructor** Special member function that cleans up an object when the object goes out of scope or is deleted.

**dynamically allocated** Object that is allocated on the free store. Objects allocated on the free store exist until they are explicitly deleted or the program terminates.

**free store** Memory pool available to a program to hold dynamically allocated objects.

**heap** Synonym for free store.

**new** Allocates memory from the free store. new T allocates and constructs an object of type T and returns a pointer to that object; if T is an array type, new returns a pointer to the first element in the array. Similarly,

new [n] T allocates n objects of type T and returns a pointer to the first element in the array. By default, the allocated object is default initialized. We may also provide optional initializers.

**placement new** Form of new that takes additional arguments passed in parentheses following the keyword new; for example, new (nothrow) int tells new that it should not throw an exception.

**reference count** Counter that tracks how many users share a common object. Used by smart pointers to know when it is safe to delete memory to which the pointers point.

**shared\_ptr** Smart pointer that provides shared ownership: The object is deleted when the last shared\_ptr pointing to that object is destroyed.

**smart pointer** Library type that acts like a pointer but can be checked to see whether it is safe to use. The type takes care of deleting memory when appropriate.

**unique\_ptr** Smart pointer that provides single ownership: The object is deleted when the unique\_ptr pointing to that object is destroyed. unique\_ptrs cannot be directly copied or assigned.

**weak\_ptr** Smart pointer that points to an object managed by a shared\_ptr. The shared\_ptr does not count weak\_ptrs when deciding whether to delete its object.

*This page intentionally left blank*

# P A R T

# III

## TOOLS FOR CLASS AUTHORS

### CONTENTS

---

|  |     |
|--|-----|
| Chapter 13 Copy Control . . . . .                          | 495 |
| Chapter 14 Overloaded Operations and Conversions . . . . . | 551 |
| Chapter 15 Object-Oriented Programming . . . . .           | 591 |
| Chapter 16 Templates and Generic Programming . . . . .     | 651 |

Classes are the central concept in C++. Chapter 7 began our detailed coverage of how classes are defined. That chapter covered topics fundamental to any use of classes: class scope, data hiding, and constructors. It also introduced various important class features: member functions, the implicit `this` pointer, friends, and `const`, `static`, and `mutable` members. In this part, we'll extend our coverage of classes by looking at copy control, overloaded operators, inheritance, and templates.

As we've seen, in C++ classes define constructors to control what happens when objects of the class type are initialized. Classes also control what happens when objects are copied, assigned, moved, and destroyed. In this respect, C++ differs from other languages, many of which do not give class designers the ability to control these operations. Chapter 13 covers these topics. This chapter also covers two important concepts introduced by the new standard: rvalue references and move operations.

Chapter 14 looks at operator overloading, which allows operands of class types to be used with the built-in operators. Operator overloading is one of the ways whereby C++ lets us create new types that are as intuitive to use as are the built-in types.

Among the operators that a class can overload is the function call operator. We can "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-independent 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.

# C H A P T E R

# 13

## C O P Y C O N T R O L

### CONTENTS

---

|   |     |
|---|-----|
| Section 13.1 Copy, Assign, and Destroy . . . . .  | 496 |
| Section 13.2 Copy Control and Resource Management | 510 |
| Section 13.3 Swap . . . . .                       | 516 |
| Section 13.4 A Copy-Control Example . . . . .     | 519 |
| Section 13.5 Classes That Manage Dynamic Memory . | 524 |
| Section 13.6 Moving Objects . . . . .             | 531 |
| Chapter Summary . . . . .                         | 549 |
| Defined Terms . . . . .                           | 549 |

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:

```
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 `nonconst`. 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 `nonstatic` 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:

```
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):

```
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):

```
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

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

into

```
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:

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

**Exercise 13.3:** What happens when we copy a `StrBlob`? What about `StrBlobPtrs`?

**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:

```
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.

```
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:

```
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:

```
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.



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:

```
// 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.



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:

```
{
    // 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.



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:

```
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?

```
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:

```
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:

```
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:

```
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`:

```
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.



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:

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

what output does the following code produce?

```
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

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):

```
C++ 11
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.



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



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:

```
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
};
```

C++  
11

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:

```
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
```



**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.



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`:

```
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.



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

```
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`:

```
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

```
// 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.



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_ptr`s 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:

```
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:

```

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:

```

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:

```
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:

```
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 value-like `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:

```
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:

```
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`.



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:

```
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:

```
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:

```
// 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 deletes 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 way 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.



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 `HasPtrs`. 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

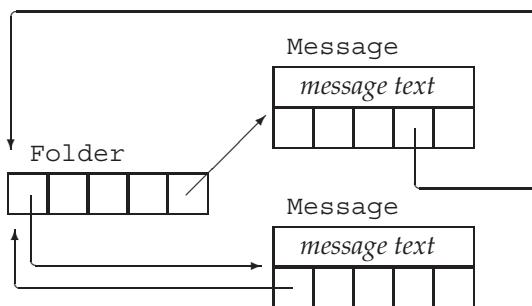
Although copy control is most often needed for classes that allocate resources, resource management is not the only reason why a class might need to define these members. Some classes have bookkeeping or other actions that the copy-control members must perform.

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 needed 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:

```
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 `Folders` 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:

```
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 `Folders` 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:

```

// 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:

```

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:

```

void Message::remove_from_Folders()
{
    for (auto f : folders) // for each pointer in folders
        f->remMsg(this); // remove this Message from that Folder
    folders.clear(); // no Folder points to this Message
}

```

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:

```
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:

```
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`:

```

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 `strings`. 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.

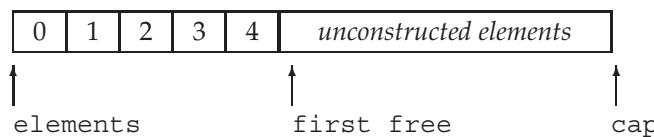
We'll use a similar strategy in our `StrVec` class. We'll use an allocator to obtain raw memory (§ 12.2.2, p. 481). Because the memory an allocator allocates is unconstructed, we'll use the allocator's `construct` member to create objects in that space when we need to add an element. Similarly, when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

```
// 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:
    static std::allocator<std::string> alloc; // allocates the elements
    void chk_n_alloc() // used by functions that add elements to a StrVec
        { 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
};

// alloc must be defined in the StrVec implementation file
allocator<string> StrVec::alloc;
```

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. When `chk_n_alloc` returns, `push_back` knows that there is room for the new element. It asks its `alloc` member to construct a new last element:

```
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`, 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:

```
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 must `destroy` the elements and then deallocate the space this `StrVec` allocated. The `for` loop calls the allocator member `destroy` in reverse order, starting with the last constructed element and finishing with the first:

```
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`:

```
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`:

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

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

```
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 value-like 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

C++  
11

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 `realloc` 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` 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 `realloc` Member

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

```
void StrVec::realloc()
{
    // 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., strings), 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.



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

C++  
11

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:

```
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.



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:

```
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.



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

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.

C++  
11

```
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.



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 `Strings` 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:

```
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 possibility that moving an object of our class type might throw.

One way to 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:

C++  
11

```
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.



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 explicitly 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:

```
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 `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.



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:

```
// 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
```



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:

```
// 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.



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 (nonconst) rvalue:

```
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`:

```
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.



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:

```
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:

```
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 swaps 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:

```
// 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:

```
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:

```
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`:

```
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.



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 `nonconst`.

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:

```
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 `nonconst` 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.



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`:

```
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:

```
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:

```
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 concatenating these strings.

Prior to the new standard, there was no way to prevent such usage. In order to maintain backward compatibility, 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.

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:

```
C++ 11
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:

```

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:

```

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:

```

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:

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

When we define `const` member 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:

```
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.



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:

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

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

```
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 nonconst) 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 nonstatic 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 nonstatic 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 nonstatic 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.

# C H A P T E R      14

## O V E R L O A D E D   O P E R A T I O N S

### A N D   C O N V E R S I O N S

## CONTENTS

---

|  |     |
|--|-----|
| Section 14.1 Basic Concepts . . . . .                      | 552 |
| Section 14.2 Input and Output Operators . . . . .          | 556 |
| Section 14.3 Arithmetic and Relational Operators . . . . . | 560 |
| Section 14.4 Assignment Operators . . . . .                | 563 |
| Section 14.5 Subscript Operator . . . . .                  | 564 |
| Section 14.6 Increment and Decrement Operators . . . . .   | 566 |
| Section 14.7 Member Access Operators . . . . .             | 569 |
| Section 14.8 Function-Call Operator . . . . .              | 571 |
| Section 14.9 Overloading, Conversions, and Operators       | 579 |
| Chapter Summary . . . . .                                  | 590 |
| Defined Terms . . . . .                                    | 590 |

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

```
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:

```
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.



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:

```
// 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).

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  $x == (y + z)$ .

**Table 14.1: Operators**

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

## 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:

```
// 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:

```
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.



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:

```
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 `vectors` that hold `strings`, identify which version of `==` is applied in each of the following expressions:

- |                                 |                                       |
|---------------------------------|---------------------------------------|
| (a) "cobble" == "stone"         | (b) <code>svec1[0] == svec2[0]</code> |
| (c) <code>svec1 == svec2</code> | (d) <code>"svec1[0] == "stone"</code> |

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

- |                    |                     |                     |                        |                           |                             |                     |                     |
|--------------------|---------------------|---------------------|------------------------|---------------------------|-----------------------------|---------------------|---------------------|
| (a) <code>%</code> | (b) <code>%=</code> | (c) <code>++</code> | (d) <code>-&gt;</code> | (e) <code>&lt;&lt;</code> | (f) <code>&amp;&amp;</code> | (g) <code>==</code> | (h) <code>()</code> |
|--------------------|---------------------|---------------------|------------------------|---------------------------|-----------------------------|---------------------|---------------------|

**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) <code>Book</code>    | (b) <code>Date</code>   | (c) <code>Employee</code> |
| (d) <code>Vehicle</code> | (e) <code>Object</code> | (f) <code>Tree</code>     |



## 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:

```
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.



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:

```
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 nonpublic 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 your `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 (nonconst) object into which to read. The operator usually returns a reference to its given stream. The second parameter must be nonconst 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:

```
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.



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:

```
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.



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?

```
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:

```
// 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`.



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:

```
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.



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 requirement 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.



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:

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

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

```
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:

```
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`.



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:

```
// 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;
}
```



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.



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 non`const` 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.



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):

```
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 non`const`, we can assign to that element; if we subscript a `const` object, we can't:

```
// 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.



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):

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



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:

```
// 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(curr, "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`:

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



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:

```
// 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.



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:

```
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:

```
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.



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 decrement 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:

```
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

```
(*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.



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:

```
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:

```
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.



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:

```
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:

```
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:

```
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`:

```
// 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

```
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 strings 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:

```
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 strings. 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:

```
// 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

```

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:

```
// 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 `ints`; `plus<Sales_data>` applies `+` to `Sales_data`s; and so on:

```

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**

| <i>Arithmetic</i> | <i>Relational</i>   | <i>Logical</i>    |
|-------------------|---------------------|-------------------|
| 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>`,

```
// 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:

```

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 `ints`.

### 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 `ints` 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:

```
// 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 divide {
    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

```
// 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:

```
// 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 an object of type `divide` in `binops`:

```
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**.

C++  
11

`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:

```
function<int(int, int)> f1 = add;           // function pointer
function<int(int, int)> f2 = divide(); // 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:

```
// 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:

```
map<string, function<int(int, int)>> binops = {
    {"+", add},                                // function pointer
    {"-", std::minus<int>()},                  // library function object
    {"/", divide()},                           // 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:

```
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 divide 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`:

```
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:

```
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
```

**Table 14.3: Operations on function**

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

Alternatively, we can use a lambda to disambiguate:

```
// 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 `ints`. That call can match only the version of `add` that takes two `ints`, and so that is the function that is called when the lambda is executed.



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.



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:

```
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`:

```
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 conversion operator to convert a `SmallInt` to an `int` and then convert the resulting `int` value to another arithmetic type:

```
// 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:

```
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 Surprising 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:

```
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

To prevent such problems, the new standard introduced **explicit conversion operators**:

```
CPP 11
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:

```
SmallInt si = 3; // ok: the SmallInt constructor is not explicit
si + 3; // error: implicit 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.



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.



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`:

```
// 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:

```
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:

```
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:

```
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`.



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 `nonexplicit` constructors to those that are "obviously right."

```
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:

```
manip(C(10)); // ok: calls manip(const C&)
```



Need 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:

```
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.



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

```
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 distinguish 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.

```
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.

```
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.



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:

```
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:

```
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 `ints`.



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:

```
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. Nonexplicit constructors that can be called with a single argument define conversions from the parameter type to the class type; nonexplicit 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. Nonexplicit 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.

# C H A P T E R      15

## O B J E C T - O R I E N T E D

## P R O G R A M M I N G

### CONTENTS

---

|  |     |
|--|-----|
| Section 15.1 OOP: An Overview . . . . .                  | 592 |
| Section 15.2 Defining Base and Derived Classes . . . . . | 594 |
| Section 15.3 Virtual Functions . . . . .                 | 603 |
| Section 15.4 Abstract Base Classes . . . . .             | 608 |
| Section 15.5 Access Control and Inheritance . . . . .    | 611 |
| Section 15.6 Class Scope under Inheritance . . . . .     | 617 |
| Section 15.7 Constructors and Copy Control . . . . .     | 622 |
| Section 15.8 Containers and Inheritance . . . . .        | 630 |
| Section 15.9 Text Queries Revisited . . . . .            | 634 |
| Chapter Summary . . . . .                                | 649 |
| Defined Terms . . . . .                                  | 649 |

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:

```
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:

```
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:

```
// 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`:

```
// 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**.



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:

```
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.



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 nonstatic 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:

```
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.

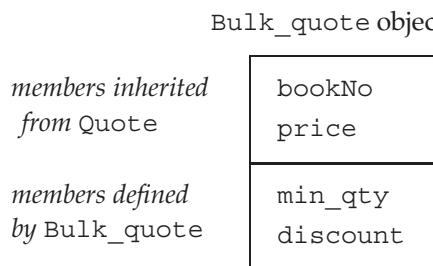
A derived class may include the `virtual` keyword on the functions it overrides, but it is not required to do so. For reasons we'll explain in § 15.3 (p. 606), the new standard lets a derived class explicitly note that it intends a member function to override a virtual that it inherits. It does so by 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.

```

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.



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.



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:

```
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.



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:

```
// if the specified number of items are purchased, use the discounted price
double Bulk_quote::net_price(std::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.

```
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:

```
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:

```
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:

```
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:

```
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

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`:

C++ 11

```
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 to 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



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.



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.



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):

```
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:

```
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:

```
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**.



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

### 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.

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.

### 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`:

```
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.

```
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.



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).



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.

C++  
11

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:

```
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:

```
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.



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:

```
// 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.



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.



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:

```
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:

```
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:

```
// 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 overriding) 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`:

```
// 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.



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`:

```
// 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:

```

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:

```

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`:

```
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:

```
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.



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:

```
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:

```
// 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
};
```



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):

```
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.



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:

```
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.



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:

```
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:

```
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`:

```
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):

```
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

```
Derived d(42);
cout << d.get_mem() << endl; // prints 42
```

then the output would be 42.



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:

```
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.



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:

```
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:

```
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:

```
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)`:

```
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:

```
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:

```
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 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 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

C++  
11

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`,

```
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 its 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):

```
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.



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:

```
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,

```
// 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.



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:

```

// 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:

```

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.



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

C++  
11

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`:

```
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

```
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:

```
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).



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:

```
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:

```
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_ptrs` 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 `isbns` of the objects to which the `shared_ptrs` point. We initialize our `multiset` to use this comparison function through an in-class initializer (§ 7.3.1, p. 274):

```
// 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\_ptrs 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:

```
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:

```
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

```
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):

```
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.

```
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`:

```

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 `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

### **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.

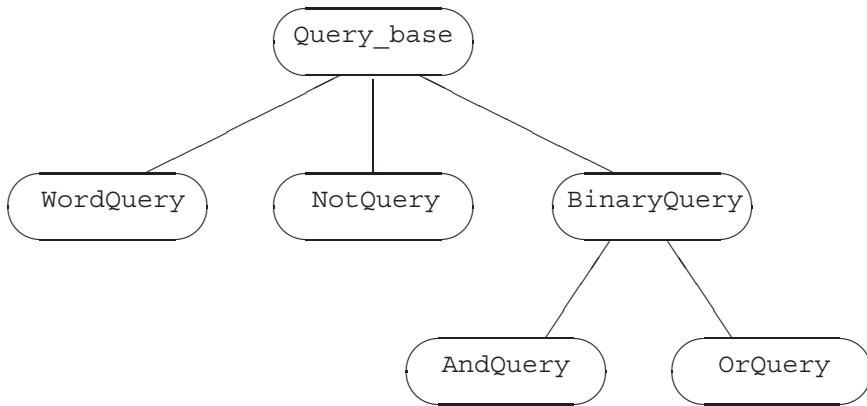
When we define a class as publicly inherited from another, the derived class should reflect an “Is A” relationship to the base class. In well-designed class hierarchies, objects of a publicly derived class can be used wherever an object of the base class is expected.

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.

`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**



### **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

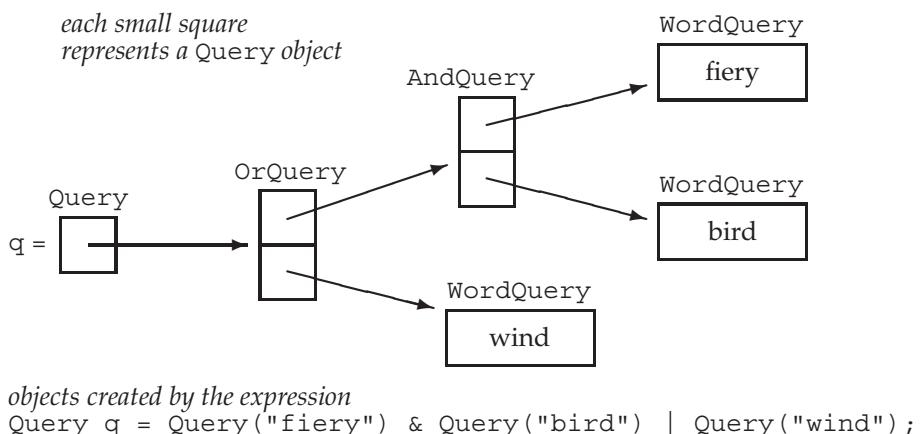
```
Query q = Query("fiery") & Query("bird") | Query("wind");
```

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`.

**Figure 15.3: Objects Created by `Query` Expressions**



## 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.

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).

### 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:

- (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:

```
// 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.

| 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.                                                                                                     |

## 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:

```
// 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:

```
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

```
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:

```
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`:

```
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:

```

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

```

// 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:

```

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.



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:

```
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):

- List the constructors executed in processing that expression.
- List the calls to `rep` that are made from `cout << q`.
- 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.

```
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.



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:

```

// 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:

```

// 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 sets.

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:

```
// 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 sequential 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 inaccessible 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 inher-

its. 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.

# C H A P T E R 16

## TEMPLATES AND GENERIC PROGRAMMING

### CONTENTS

---

|                                                    |     |
|----------------------------------------------------|-----|
| Section 16.1 Defining a Template . . . . .         | 652 |
| Section 16.2 Template Argument Deduction . . . . . | 678 |
| Section 16.3 Overloading and Templates . . . . .   | 694 |
| Section 16.4 Variadic Templates . . . . .          | 699 |
| Section 16.5 Template Specializations . . . . .    | 706 |
| Chapter Summary . . . . .                          | 713 |
| Defined Terms . . . . .                            | 713 |

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 vectors, 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:

```
// 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

```
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.



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

```
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

```
// 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`:

```
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:

```
// 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`:

```
// 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:

```
// 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:

```
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

```
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.



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:

```
// 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):

```
// 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.



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



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:

```
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,

```
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`.



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 `strings`. 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:

```
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:

```
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

```
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:

```
// 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`.



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`

```
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

```
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

```
ret-type StrBlob::member-name (parm-list)
```

the corresponding `Blob` member will look like

```
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:

```
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:

```

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:

```

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:

```

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:

```

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`:

```

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

```
// 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).



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:

```
// 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):

```
// 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;
```



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:

```
// 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:

```
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:

```
// 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:

```
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:

```
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 pairs 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:

```
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:

```
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 pairs 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):

```
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,

```
// 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:

```
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:

```
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?

```
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:

```
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:

```
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:

```
// error: illegal reuse of template parameter name V
template <typename V, typename V> // ...
```

## Template Declarations

A template declaration must include the template parameters :

```
// 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:

```
// 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.



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`:

```
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.



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.

As an example, we'll rewrite `compare` to use the library `less` function-object template (§ 14.8.2, p. 574) by default:

```
// 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:

```
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:

```
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.

- (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 Ordinary (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:

```
// 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`:

```
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 deletes 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):

```
// destroying the the object to which p points
// instantiates DebugDelete::operator()(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:

```
// 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:

```
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:

```
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):

```
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

```
extern template declaration; // instantiation declaration  
template declaration; // instantiation definition
```

C++  
11

where *declaration* is a class or function declaration in which all the template parameters are replaced by the template arguments. For example,

```
// instantiation 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 `nonextern` 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:

```
// 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:

```
// 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.



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.



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:

```
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.

```
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

```
// 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:

```
// 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 `consts` (§ 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:

```
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 permissible 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.



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:

```
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:

```
// 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:

```
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:

```
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:

```
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&`.



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?

```
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?

```
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:

```
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); (e) 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:

```
// 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:

```
// 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

```
// 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:

```
// 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:

```

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:

```

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:

```

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:

```
// 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 strings, 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).

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:

```
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:

```
// 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).

**Table 16.1: Standard Type Transformation Templates**

| For <code>Mod&lt;T&gt;</code> , where <code>Mod</code> is | If <code>T</code> is                  | Then <code>Mod&lt;T&gt;::type</code> is |
|-----------------------------------------------------------|---------------------------------------|-----------------------------------------|
| <code>remove_reference</code>                             | X& or X&&<br>otherwise                | X<br>T                                  |
| <code>add_const</code>                                    | X&, const X, or function<br>otherwise | T<br>const T                            |
| <code>add_lvalue_reference</code>                         | X&<br>X&&<br>otherwise                | T<br>X&<br>T&                           |
| <code>add_rvalue_reference</code>                         | X& or X&&<br>otherwise                | T<br>T&&                                |
| <code>remove_pointer</code>                               | X*                                    | X<br>T                                  |
| <code>add_pointer</code>                                  | X& or X&&<br>otherwise                | X*<br>T*                                |
| <code>make_signed</code>                                  | unsigned X<br>otherwise               | X<br>T                                  |
| <code>make_unsigned</code>                                | signed type<br>otherwise              | unsigned T<br>T                         |
| <code>remove_extent</code>                                | X[n]<br>otherwise                     | X<br>T                                  |
| <code>remove_all_extents</code>                           | X[n1] [n2] ...<br>otherwise           | X<br>T                                  |

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?

```
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`:

```
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:

```
// 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 `ints` or the version that takes `strings`. 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:

```
// 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.



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

```
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 *consts* 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:

```
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:

```
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:

```
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.

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&&`



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:

```
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

```
// 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&`):

```
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.



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:

```
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):

```
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:

```
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?

```
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:

```
// 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:

```
string s1("hi!"), s2;
s2 = std::move(string("bye!")); // ok: moving from an rvalue
s2 = std::move(s1); // ok: but after the assignment 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`.

C++  
11

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`

reallocating 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::reallocation` in § 13.5 (p. 530):

```
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:

```
// 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:

```
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:

```
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 a plain, nonreference type, `int`, not an `int&`. That is, the instantiation of this call to `flip1` is

```
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 `constness` 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 `constness`, 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):

```
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`.



A function parameter that is an rvalue reference to a template type parameter (i.e., `T&&`) preserves the `constness` 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:

```
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`:

```
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&&`.

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:

```
template <typename Type> intermediary(Type &&arg)
{
    finalFn(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.



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:

```
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&&`.



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.



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:

```
// 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:

```
// 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:

```
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:

```
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:

```
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.



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 strings inside double quotes:

```
// print strings inside double quotes
string debug_rep(const string &s)
{
    return '"' + s + '"';
}
```

Now, when we call `debug_rep` on a `string`,

```
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.



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:

```
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 `strings`, we can define two more nontemplate overloads:

```
// 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:

```
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&`.



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:

```
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:

```
// 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:

```
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`:

```
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:

C++  
11

```
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:

```
// 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:

```
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:

```
print(cout, i, s, 42); // two parameters in the pack
```

the recursion will execute as follows:

| Call                                                                             | <code>t</code> | <code>rest...</code> |
|----------------------------------------------------------------------------------|----------------|----------------------|
| <code>print(cout, i, s, 42)</code>                                               | <code>i</code> | <code>s, 42</code>   |
| <code>print(cout, s, 42)</code>                                                  | <code>s</code> | <code>42</code>      |
| <code>print(cout, 42)</code> calls the nonvariadic version of <code>print</code> |                |                      |

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).



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:

```
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:

```
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

```
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 strings:

```
// 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

```
errorMsg(cerr, fcnName, code.num(), otherData, "other", item);
```

will execute as if we had written

```
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:

```
// 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

```
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.



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.

C++  
11

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):

```
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):

```
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< $T_i$ >( $t_i$ )
```

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

```
svec.emplace_back(10, 'c'); // adds cccccccccc as a new last element
```

the pattern in the call to `construct` will expand to

```
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:

```
svec.emplace_back(s1 + s2); // uses the move constructor
```

the argument to `emplace_back` is an rvalue, which is passed to `construct` as

```
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:

```
// 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):

```
// 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:

```
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:

```
// 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:

```
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.



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.



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:

```
// 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 curly braces will be part of the `std` namespace.

The following defines a specialization of `hash` for `Sales_data`:

```
// 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:

```
// 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`:

```
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`.



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.



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:

```
// 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:

```
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:

```
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:

```
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 `nonextern` 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 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 specified 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 ellipsis (e.g., `class...`, `typename...`, or `type-name...`).

# P A R T

# IV

## ADVANCED TOPICS

### CONTENTS

---

|                                                       |     |
|-------------------------------------------------------|-----|
| Chapter 17 Specialized Library Facilities . . . . .   | 717 |
| Chapter 18 Tools for Large Programs . . . . .         | 771 |
| Chapter 19 Specialized Tools and Techniques . . . . . | 819 |

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 exception handling, which is why we introduced the basics of exception handling in Chapter 5. However, the need to deal with run-time errors tends to be more important and harder to manage in problems that require large programming teams. In Chapter 18 we review some additional useful exception-handling facilities. We also look in more detail at how exceptions are handled, and show how we can define and use our own exception classes. This section will also cover improvements from the new standard regarding specifying that a particular function will not throw.

Large-scale applications often use code from multiple independent vendors. Combining independently developed libraries would be difficult (if not impossible) if vendors had to put the names they define into a single namespace. Independently developed libraries would almost inevitably use names in common with one another; a name defined in one library would conflict with the use of that name in another library. To avoid name collisions, we can define names inside a namespace.

Whenever we use a name from the standard library, we are using a name defined in the namespace named `std`. Chapter 18 shows how we can define our own namespaces.

Chapter 18 closes by looking at an important but infrequently used language feature: multiple inheritance. Multiple inheritance is most useful for fairly complicated inheritance hierarchies.

Chapter 19 covers several specialized tools and techniques that are applicable to particular kinds of problems. Among the features covered in this chapter are how to redefine how memory allocation works; C++ support for run-time type identification (RTTI), which let us determine the actual type of an expression at run time; and how we can define and use pointers to class members. Pointers to class members differ from pointers to ordinary data or functions. Ordinary pointers only vary based on the type of the object or function. Pointers to members must also reflect the class to which the member belongs. We'll also look at three additional aggregate types: unions, nested classes, and local classes. The chapter closes by looking briefly at a collection of features that are inherently nonportable: the `volatile` qualifier, bit-fields, and linkage directives.

# C H A P T E R      17

## SPECIALIZED LIBRARY FACILITIES

### CONTENTS

---

|                                                     |     |
|-----------------------------------------------------|-----|
| Section 17.1 The <code>tuple</code> Type . . . . .  | 718 |
| Section 17.2 The <code>bitset</code> Type . . . . . | 723 |
| Section 17.3 Regular Expressions . . . . .          | 728 |
| Section 17.4 Random Numbers . . . . .               | 745 |
| Section 17.5 The IO Library Revisited . . . . .     | 752 |
| Chapter Summary . . . . .                           | 769 |
| Defined Terms . . . . .                             | 769 |

The latest standard greatly increased the size and scope of the library. Indeed, the portion of the standard devoted to the library more than doubled between the first release in 1998 and the 2011 standard. As a result, covering every C++ library class is well beyond the scope of this Primer. However, there are four library facilities that, although more specialized than other library facilities we've covered, are general enough to warrant discussion in an introductory book: `tuples`, `bitsets`, random-number generation, and regular expressions. In addition, we will also cover some additional, special-purpose parts of the IO library.

*The library* constitutes nearly two-thirds of the text of the new standard. Although we cannot cover every library facility in depth, there remain a few library facilities that are likely to be of use in many applications: tuples, bitsets, regular expressions, and random numbers. We'll also look at some additional IO library capabilities: format control, unformatted IO, and random access.

## 17.1 The tuple Type

A **tuple** is a template that is similar to a pair (§ 11.2.3, p. 426). Each pair type has different types for its members, but every pair always has exactly two members.

C++  
11

A tuple also has members whose types vary from one tuple type to another, but a tuple can have any number of members. Each distinct tuple type has a fixed number of members, but the number of members in one tuple type can differ from the number of members in another.

A tuple is most useful when we want to combine some data into a single object but do not want to bother to define a data structure to represent those data. Table 17.1 lists the operations that tuples support. The tuple type, along with its companion types and functions, are defined in the tuple header.

*Note*

A tuple can be thought of as a “quick and dirty” data structure.

### 17.1.1 Defining and Initializing tuples

When we define a tuple, we name the type(s) of each of its members:

```
tuple<size_t, size_t, size_t> threeD; // all three members set to 0
tuple<string, vector<double>, int, list<int>>
    someVal("constants", {3.14, 2.718}, 42, {0,1,2,3,4,5});
```

When we create a tuple object, we can use the default tuple constructor, which value initializes (§ 3.3.1, p. 98) each member, or we can supply an initializer for each member as we do in the initialization of someVal. This tuple constructor is explicit (§ 7.5.4, p. 296), so we must use the direct initialization syntax:

```
tuple<size_t, size_t, size_t> threeD = {1,2,3}; // error
tuple<size_t, size_t, size_t> threeD{1,2,3}; // ok
```

Alternatively, similar to the make\_pair function (§ 11.2.3, p. 428), the library defines a make\_tuple function that generates a tuple object:

```
// tuple that represents a bookstore transaction: ISBN, count, price per book
auto item = make_tuple("0-999-78345-X", 3, 20.00);
```

Like make\_pair, the make\_tuple function uses the types of the supplied initializers to infer the type of the tuple. In this case, item is a tuple whose type is tuple<const char\*, int, double>.

**Table 17.1: Operations on tuples**

|                                                               |                                                                                                                                                                                                                         |
|---------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>tuple&lt;T1, T2, ..., Tn&gt; t;</code>                  | t is a tuple with as many members as there are types T1 ... Tn. The members are value initialized (§ 3.3.1, p. 98).                                                                                                     |
| <code>tuple&lt;T1, T2, ..., Tn&gt; t(v1, v2, ..., vn);</code> | t is a tuple with types T1 ... Tn in which each member is initialized from the corresponding initializer, vi. This constructor is explicit (§ 7.5.4, p. 296).                                                           |
| <code>make_tuple(v1, v2, ..., vn)</code>                      | Returns a tuple initialized from the given initializers. The type of the tuple is inferred from the types of the initializers.                                                                                          |
| <code>t1 == t2</code>                                         | Two tuples are equal if they have the same number of members and if each pair of members are equal. Uses each member's underlying == operator. Once a member is found to be unequal, subsequent members are not tested. |
| <code>t1 relop t2</code>                                      | Relational operations on tuples using dictionary ordering (§ 9.2.7, p. 340). The tuples must have the same number of members. Members of t1 are compared with the corresponding members from t2 using the < operator    |
| <code>get&lt;i&gt;(t)</code>                                  | Returns a reference to the i <sup>th</sup> data member of t; if t is an lvalue, the result is an lvalue reference; otherwise, it is an rvalue reference. All members of a tuple are public.                             |
| <code>tuple_size&lt;tupleType&gt;::value</code>               | A class template that can be instantiated by a tuple type and has a public constexpr static data member named value of type size_t that is number of members in the specified tuple type.                               |
| <code>tuple_element&lt;i, tupleType&gt;::type</code>          | A class template that can be instantiated by an integral constant and a tuple type and has a public member named type that is the type of the specified members in the specified tuple type.                            |

## Accessing the Members of a tuple

A pair always has two members, which makes it possible for the library to give these members names (i.e., first and second). No such naming convention is possible for tuple because there is no limit on the number of members a tuple type can have. As a result, the members are unnamed. Instead, we access the members of a tuple through a library function template named `get`. To use get we must specify an explicit template argument (§ 16.2.2, p. 682), which is the position of the member we want to access. We pass a tuple object to get, which returns a reference to the specified member:

```
auto book = get<0>(item);           // returns the first member of item
auto cnt = get<1>(item);            // returns the second member of item
auto price = get<2>(item) / cnt;    // returns the last member of item
get<2>(item) *= 0.8;               // apply 20% discount
```

The value inside the brackets must be an integral constant expression (§ 2.4.4, p. 65). As usual, we count from 0, meaning that `get<0>` is the first member.

If we have a tuple whose precise type details we don't know, we can use two auxilliary class templates to find the number and types of the tuple's members:

```
typedef decltype(item) trans; // trans is the type of item
// returns the number of members in object's of type trans
size_t sz = tuple_size<trans>::value; // returns 3
// cnt has the same type as the second member in item
tuple_element<1, trans>::type cnt = get<1>(item); // cnt is an int
```

To use `tuple_size` or `tuple_element`, we need to know the type of a tuple object. As usual, the easiest way to determine an object's type is to use `decltype` (§ 2.5.3, p. 70). Here, we use `decltype` to define a type alias for the type of `item`, which we use to instantiate both templates.

`tuple_size` has a public static data member named `value` that is the number of members in the specified tuple. The `tuple_element` template takes an index as well as a tuple type. `tuple_element` has a public type member named `type` that is the type of the specified member of the specified tuple type. Like `get`, `tuple_element` uses indices starting at 0.

## Relational and Equality Operators

The `tuple` relational and equality operators behave similarly to the corresponding operations on containers (§ 9.2.7, p. 340). These operators execute pairwise on the members of the left-hand and right-hand tuples. We can compare two tuples only if they have the same number of members. Moreover, to use the equality or inequality operators, it must be legal to compare each pair of members using the `==` operator; to use the relational operators, it must be legal to use `<`. For example:

```
tuple<string, string> duo("1", "2");
tuple<size_t, size_t> twoD(1, 2);
bool b = (duo == twoD); // error: can't compare a size_t and a string
tuple<size_t, size_t, size_t> threeD(1, 2, 3);
b = (twoD < threeD); // error: differing number of members
tuple<size_t, size_t> origin(0, 0);
b = (origin < twoD); // ok: b is true
```



Because `tuple` defines the `<` and `==` operators, we can pass sequences of tuples to the algorithms and can use a `tuple` as key type in an ordered container.

### EXERCISES SECTION 17.1.1

**Exercise 17.1:** Define a `tuple` that holds three `int` values and initialize the members to 10, 20, and 30.

**Exercise 17.2:** Define a `tuple` that holds a `string`, a `vector<string>`, and a `pair<string, int>`.

**Exercise 17.3:** Rewrite the `TextQuery` programs from § 12.3 (p. 484) to use a `tuple` instead of the `QueryResult` class. Explain which design you think is better and why.

## 17.1.2 Using a tuple to Return Multiple Values

A common use of `tuple` is to return multiple values from a function. For example, our bookstore might be one of several stores in a chain. Each store would have a transaction file that holds data on each book that the store recently sold. We might want to look at the sales for a given book in all the stores.

We'll assume that we have a file of transactions for each store. Each of these per-store transaction files will contain all the transactions for each book grouped together. We'll further assume that some other function reads these transaction files, builds a `vector<Sales_data>` for each store, and puts those vectors in a `vector` of vectors:

```
// each element in files holds the transactions for a particular store
vector<vector<Sales_data>> files;
```

We'll write a function that will search `files` looking for the stores that sold a given book. For each store that has a matching transaction, we'll create a `tuple` to hold the index of that store and two iterators. The index will be the position of the matching store in `files`. The iterators will mark the first and one past the last record for the given book in that store's `vector<Sales_data>`.

### A Function That Returns a tuple

We'll start by writing the function to find a given book. This function's arguments are the `vector` of `vectors` just described, and a `string` that represents the book's ISBN. Our function will return a `vector` of `tuples` that will have an entry for each store with at least one sale for the given book:

```
// matches has three members: an index of a store and iterators into that store's vector
typedef tuple<vector<Sales_data>::size_type,
              vector<Sales_data>::const_iterator,
              vector<Sales_data>::const_iterator> matches;

// files holds the transactions for every store
// findBook returns a vector with an entry for each store that sold the given book
vector<matches>
findBook(const vector<vector<Sales_data>> &files,
         const string &book)
{
    vector<matches> ret; // initially empty
    // for each store find the range of matching books, if any
    for (auto it = files.cbegin(); it != files.cend(); ++it) {
        // find the range of Sales_data that have the same ISBN
        auto found = equal_range(it->cbegin(), it->cbegin(),
                                book, compareIsbn);
        if (found.first != found.second) // this store had sales
            // remember the index of this store and the matching range
            ret.push_back(make_tuple(it - files.cbegin(),
                                    found.first, found.second));
    }
    return ret; // empty if no matches found
}
```

The `for` loop iterates through the elements in `files`. Those elements are themselves vectors. Inside the `for` we call a library algorithm named `equal_range`, which operates like the associative container member of the same name (§ 11.3.5, p. 439). The first two arguments to `equal_range` are iterators denoting an input sequence (§ 10.1, p. 376). The third argument is a value. By default, `equal_range` uses the `<` operator to compare elements. Because `Sales_data` does not have a `<` operator, we pass a pointer to the `compareIsbn` function (§ 11.2.2, p. 425).

The `equal_range` algorithm returns a pair of iterators that denote a range of elements. If book is not found, then the iterators will be equal, indicating that the range is empty. Otherwise, the first member of the returned pair will denote the first matching transaction and second will be one past the last.

## Using a tuple Returned by a Function

Once we have built our vector of stores with matching transactions, we need to process these transactions. In this program, we'll report the total sales results for each store that has a matching sale:

```
void reportResults(istream &in, ostream &os,
                   const vector<vector<Sales_data>> &files)
{
    string s; // book to look for
    while (in >> s) {
        auto trans = findBook(files, s); // stores that sold this book
        if (trans.empty()) {
            cout << s << " not found in any stores" << endl;
            continue; // get the next book to look for
        }
        for (const auto &store : trans) // for every store with a sale
            // get<n> returns the specified member from the tuple in store
            os << "store " << get<0>(store) << " sales: "
                << accumulate(get<1>(store), get<2>(store),
                               Sales_data(s))
                << endl;
    }
}
```

The `while` loop repeatedly reads the `istream` named `in` to get the next book to process. We call `findBook` to see if `s` is present, and assign the results to `trans`. We use `auto` to simplify writing the type of `trans`, which is a vector of tuples.

If `trans` is empty, there were no sales for `s`. In this case, we print a message and return to the `while` to get the next book to look for.

The `for` loop binds `store` to each element in `trans`. Because we don't intend to change the elements in `trans`, we declare `store` as a reference to `const`. We use `get` to print the relevant data: `get<0>` is the index of the corresponding store, `get<1>` is the iterator denoting the first transaction, and `get<2>` is the iterator one past the last.

Because `Sales_data` defines the addition operator (§ 14.3, p. 560), we can use the library `accumulate` algorithm (§ 10.2.1, p. 379) to sum the transactions. We

pass a `Sales_data` object initialized by the `Sales_data` constructor that takes a string (§ 7.1.4, p. 264) as the starting point for the summation. That constructor initializes the `bookNo` member from the given string and the `units_sold` and `revenue` members to zero.

### EXERCISES SECTION 17.1.2

**Exercise 17.4:** Write and test your own version of the `findBook` function.

**Exercise 17.5:** Rewrite `findBook` to return a pair that holds an index and a pair of iterators.

**Exercise 17.6:** Rewrite `findBook` so that it does not use `tuple` or `pair`.

**Exercise 17.7:** Explain which version of `findBook` you prefer and why.

**Exercise 17.8:** What would happen if we passed `Sales_data()` as the third parameter to `accumulate` in the last code example in this section?

## 17.2 The `bitset` Type

In § 4.8 (p. 152) we covered the built-in operators that treat an integral operand as a collection of bits. The standard library defines the `bitset` class to make it easier to use bit operations and possible to deal with collections of bits that are larger than the longest integral type. The `bitset` class is defined in the `bitset` header.

### 17.2.1 Defining and Initializing `bitsets`

Table 17.2 (overleaf) lists the constructors for `bitset`. The `bitset` class is a class template that, like the `array` class, has a fixed size (§ 9.2.4, p. 336). When we define a `bitset`, we say how many bits the `bitset` will contain:

```
bitset<32> bitvec(1U); // 32 bits; low-order bit is 1, remaining bits are 0
```

The size must be a constant expression (§ 2.4.4, p. 65). This statement defines `bitvec` as a `bitset` that holds 32 bits. Just as with the elements of a `vector`, the bits in a `bitset` are not named. Instead, we refer to them positionally. The bits are numbered starting at 0. Thus, `bitvec` has bits numbered 0 through 31. The bits starting at 0 are referred to as the **low-order** bits, and those ending at 31 are referred to as **high-order** bits.

#### Initializing a `bitset` from an `unsigned` Value

When we use an integral value as an initializer for a `bitset`, that value is converted to `unsigned long long` and is treated as a bit pattern. The bits in the `bitset` are a copy of that pattern. If the size of the `bitset` is greater than the number of bits in an `unsigned long long`, then the remaining high-order bits

| Table 17.2: Ways to Initialize a <code>bitset</code>                                                                                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>bitset&lt;n&gt; b;</code>                                                                                                                                                                                  | <code>b</code> has <code>n</code> bits; each bit is 0. This constructor is a <code>constexpr</code> (§ 7.5.6, p. 299).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <code>bitset&lt;n&gt; b(u);</code>                                                                                                                                                                               | <code>b</code> is a copy of the <code>n</code> low-order bits of <code>unsigned long long</code> value <code>u</code> . If <code>n</code> is greater than the size of an <code>unsigned long long</code> , the high-order bits beyond those in the <code>unsigned long long</code> are set to zero. This constructor is a <code>constexpr</code> (§ 7.5.6, p. 299).                                                                                                                                                                                                                                                              |
| <code>bitset&lt;n&gt; b(s, pos, m, zero, one);</code>                                                                                                                                                            | <code>b</code> is a copy of the <code>m</code> characters from the <code>string</code> <code>s</code> starting at position <code>pos</code> . <code>s</code> may contain only the characters <code>zero</code> and <code>one</code> ; if <code>s</code> contains any other character, throws <code>invalid_argument</code> . The characters are stored in <code>b</code> as <code>zero</code> and <code>one</code> , respectively. <code>pos</code> defaults to 0, <code>m</code> defaults to <code>string::npos</code> , <code>zero</code> defaults to ' <code>0</code> ', and <code>one</code> defaults to ' <code>1</code> '. |
| <code>bitset&lt;n&gt; b(cp, pos, m, zero, one);</code>                                                                                                                                                           | Same as the previous constructor, but copies from the character array to which <code>cp</code> points. If <code>m</code> is not supplied, then <code>cp</code> must point to a C-style string. If <code>m</code> is supplied, there must be at least <code>m</code> characters that are <code>zero</code> or <code>one</code> starting at <code>cp</code> .                                                                                                                                                                                                                                                                      |
| <b>The constructors that take a <code>string</code> or character pointer are <code>explicit</code> (§ 7.5.4, p. 296). The ability to specify alternate characters for 0 and 1 was added in the new standard.</b> |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |

are set to zero. If the size of the `bitset` is less than that number of bits, then only the low-order bits from the given value are used; the high-order bits beyond the size of the `bitset` object are discarded:

```
// bitvec1 is smaller than the initializer; high-order bits from the initializer are discarded
bitset<13> bitvec1(0xbeef); // bits are 1111011101111
// bitvec2 is larger than the initializer; high-order bits in bitvec2 are set to zero
bitset<20> bitvec2(0xbeef); // bits are 0000101111011101111
// on machines with 64-bit long long OULL is 64 bits of 0, so ~OULL is 64 ones
bitset<128> bitvec3(~OULL); // bits 0...63 are one; 63...127 are zero
```

### Initializing a `bitset` from a `string`

We can initialize a `bitset` from either a `string` or a pointer to an element in a character array. In either case, the characters represent the bit pattern directly. As usual, when we use strings to represent numbers, the characters with the lowest indices in the string correspond to the high-order bits, and vice versa:

```
bitset<32> bitvec4("1100"); // bits 2 and 3 are 1, all others are 0
```

If the `string` contains fewer characters than the size of the `bitset`, the high-order bits are set to zero.

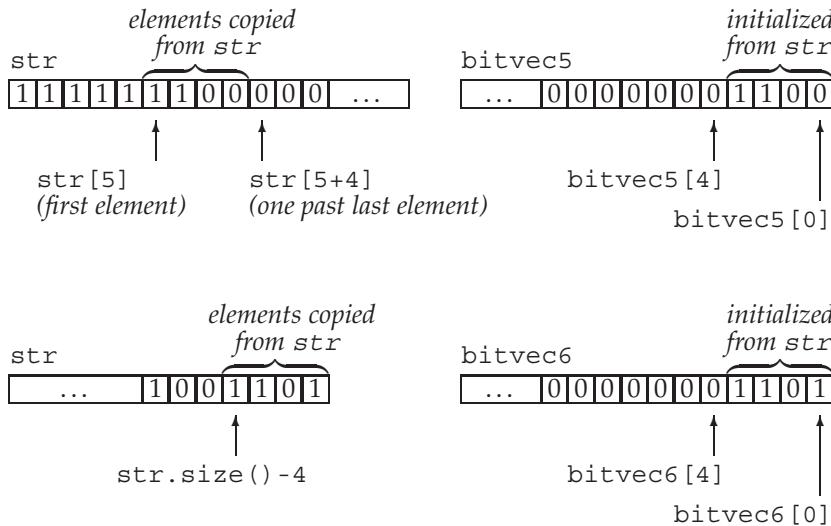


The indexing conventions of `strings` and `bitsets` are inversely related: The character in the `string` with the highest subscript (the right-most character) is used to initialize the low-order bit in the `bitset` (the bit with subscript 0). When you initialize a `bitset` from a `string`, it is essential to remember this difference.

We need not use the entire string as the initial value for the `bitset`. Instead, we can use a substring as the initializer:

```
string str("1111111000000011001101");
bitset<32> bitvec5(str, 5, 4); // four bits starting at str[5], 1100
bitset<32> bitvec6(str, str.size()-4); // use last four characters
```

Here `bitvec5` is initialized by the substring in `str` starting at `str[5]` and continuing for four positions. As usual, the right-most character of the substring represents the lowest-order bit. Thus, `bitvec5` is initialized with bit positions 3 through 0 set to 1100 and the remaining bits set to 0. The initializer for `bitvec6` passes a string and a starting point, so `bitvec6` is initialized from the characters in `str` starting four from the end of `str`. The remainder of the bits in `bitvec6` are initialized to zero. We can view these initializations as



### EXERCISES SECTION 17.2.1

**Exercise 17.9:** Explain the bit pattern each of the following `bitset` objects contains:

- `bitset<64> bitvec(32);`
- `bitset<32> bv(1010101);`
- `string bstr; cin >> bstr; bitset<8>bv(bstr);`

### 17.2.2 Operations on `bitsets`

The `bitset` operations (Table 17.3 (overleaf)) define various ways to test or set one or more bits. The `bitset` class also supports the bitwise operators that we covered in § 4.8 (p. 152). The operators have the same meaning when applied to `bitset` objects as the built-in operators have when applied to `unsigned` operands.

**Table 17.3: `bitset` Operations**

|                                     |                                                                                                                                                                                                            |
|-------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>b.any()</code>                | Is any bit in b on?                                                                                                                                                                                        |
| <code>b.all()</code>                | Are all the bits in b on?                                                                                                                                                                                  |
| <code>b.none()</code>               | Are no bits in b on?                                                                                                                                                                                       |
| <code>b.count()</code>              | Number of bits in b that are on.                                                                                                                                                                           |
| <code>b.size()</code>               | A <code>constexpr</code> function (§ 2.4.4, p. 65) that returns the number of bits in b.                                                                                                                   |
| <code>b.test(pos)</code>            | Returns <code>true</code> if bit at position pos is on, <code>false</code> otherwise.                                                                                                                      |
| <code>b.set(pos, v)</code>          | Sets the bit at position pos to the <code>bool</code> value v. v defaults to <code>true</code> . If no arguments, turns on all the bits in b.                                                              |
| <code>b.set()</code>                | Turns off the bit at position pos or turns off all the bits in b.                                                                                                                                          |
| <code>b.reset(pos)</code>           | Changes the state of the bit at position pos or of every bit in b.                                                                                                                                         |
| <code>b.reset()</code>              |                                                                                                                                                                                                            |
| <code>b.flip(pos)</code>            | Changes the state of the bit at position pos or of every bit in b.                                                                                                                                         |
| <code>b.flip()</code>               |                                                                                                                                                                                                            |
| <code>b[pos]</code>                 | Gives access to the bit in b at position pos; if b is <code>const</code> , then <code>b[pos]</code> returns a <code>bool</code> value <code>true</code> if the bit is on, <code>false</code> otherwise.    |
| <code>b.to_ulong()</code>           | Returns an <code>unsigned long</code> or an <code>unsigned long long</code> with the same bits as in b. Throws <code>overflow_error</code> if the bit pattern in b won't fit in the indicated result type. |
| <code>b.to_string(zero, one)</code> | Returns a <code>string</code> representing the bit pattern in b. zero and one default to '0' and '1' and are used to represent the bits 0 and 1 in b.                                                      |
| <code>os &lt;&lt; b</code>          | Prints the bits in b as the characters 1 or 0 to the stream os.                                                                                                                                            |
| <code>is &gt;&gt; b</code>          | Reads characters from is into b. Reading stops when the next character is not a 1 or 0 or when <code>b.size()</code> bits have been read.                                                                  |

Several operations—`count`, `size`, `all`, `any`, and `none`—take no arguments and return information about the state of the entire `bitset`. Others—`set`, `reset`, and `flip`—change the state of the `bitset`. The members that change the `bitset` are overloaded. In each case, the version that takes no arguments applies the given operation to the entire set; the versions that take a position apply the operation to the given bit:

```
bitset<32> bitvec(1U); // 32 bits; low-order bit is 1, remaining bits are 0
bool is_set = bitvec.any();           // true, one bit is set
bool is_not_set = bitvec.none();     // false, one bit is set
bool all_set = bitvec.all();         // false, only one bit is set
size_t onBits = bitvec.count();      // returns 1
size_t sz = bitvec.size();          // returns 32
bitvec.flip();                     // reverses the value of all the bits in bitvec
bitvec.reset();                    // sets all the bits to 0
bitvec.set();                      // sets all the bits to 1
```

The `any` operation returns `true` if one or more bits of the `bitset` object are turned on—that is, are equal to 1. Conversely, `none` returns `true` if all the bits are zero. The new standard introduced the `all` operation, which returns `true` if all the bits are on. The `count` and `size` operations return a `size_t` (§ 3.5.2, p. 116) equal to

the number of bits that are set, or the total number of bits in the object, respectively. The `size` function is a `constexpr` and so can be used where a constant expression is required (§ 2.4.4, p. 65).

The `flip`, `set`, `reset`, and `test` members let us read or write the bit at a given position:

```
bitvec.flip(0);      // reverses the value of the first bit
bitvec.set(bitvec.size() - 1); // turns on the last bit
bitvec.set(0, 0); // turns off the first bit
bitvec.reset(i); // turns off the ith bit
bitvec.test(0); // returns false because the first bit is off
```

The subscript operator is overloaded on `const`. The `const` version returns a `bool` value `true` if the bit at the given index is on, `false` otherwise. The `nonconst` version returns a special type defined by `bitset` that lets us manipulate the bit value at the given index position:

```
bitvec[0] = 0;           // turn off the bit at position 0
bitvec[31] = bitvec[0]; // set the last bit to the same value as the first bit
bitvec[0].flip();       // flip the value of the bit at position 0
~bitvec[0];             // equivalent operation; flips the bit at position 0
bool b = bitvec[0];     // convert the value of bitvec[0] to bool
```

## Retrieving the Value of a `bitset`

The `to_ulong` and `to_ullong` operations return a value that holds the same bit pattern as the `bitset` object. We can use these operations only if the size of the `bitset` is less than or equal to the corresponding size, `unsigned long` for `to_ulong` and `unsigned long long` for `to_ullong`:

```
unsigned long ulong = bitvec3.to_ulong();
cout << "ulong = " << ulong << endl;
```



These operations throw an `overflow_error` exception (§ 5.6, p. 193) if the value in the `bitset` does not fit in the specified type.

## `bitset` IO Operators

The input operator reads characters from the input stream into a temporary object of type `string`. It reads until it has read as many characters as the size of the corresponding `bitset`, or it encounters a character other than 1 or 0, or it encounters end-of-file or an input error. The `bitset` is then initialized from that temporary `string` (§ 17.2.1, p. 724). If fewer characters are read than the size of the `bitset`, the high-order bits are, as usual, set to 0.

The output operator prints the bit pattern in a `bitset` object:

```
bitset<16> bits;
cin >> bits; // read up to 16 1 or 0 characters from cin
cout << "bits: " << bits << endl; // print what we just read
```

## Using `bitsets`

To illustrate using `bitsets`, we'll reimplement the grading code from § 4.8 (p. 154) that used an `unsigned long` to represent the pass/fail quiz results for 30 students:

```
bool status;
// version using bitwise operators
unsigned long quizA = 0;           // this value is used as a collection of bits
quizA |= 1UL << 27;              // indicate student number 27 passed
status = quizA & (1UL << 27);    // check how student number 27 did
quizA &= ~ (1UL << 27);         // student number 27 failed

// equivalent actions using the bitset library
bitset<30> quizB;               // allocate one bit per student; all bits initialized to 0
quizB.set(27);                  // indicate student number 27 passed
status = quizB[27];              // check how student number 27 did
quizB.reset(27);                // student number 27 failed
```

### EXERCISES SECTION 17.2.2

**Exercise 17.10:** Using the sequence 1, 2, 3, 5, 8, 13, 21, initialize a `bitset` that has a 1 bit in each position corresponding to a number in this sequence. Default initialize another `bitset` and write a small program to turn on each of the appropriate bits.

**Exercise 17.11:** Define a data structure that contains an integral object to track responses to a true/false quiz containing 10 questions. What changes, if any, would you need to make in your data structure if the quiz had 100 questions?

**Exercise 17.12:** Using the data structure from the previous question, write a function that takes a question number and a value to indicate a true/false answer and updates the quiz results accordingly.

**Exercise 17.13:** Write an integral object that contains the correct answers for the true/false quiz. Use it to generate grades on the quiz for the data structure from the previous two exercises.

## 17.3 Regular Expressions

A **regular expression** is a way of describing a sequence of characters. Regular expressions are a stunningly powerful computational device. However, describing the languages used to define regular expressions is well beyond the scope of this Primer. Instead, we'll focus on how to use the C++ regular-expression library (RE library), which is part of the new library. The RE library, which is defined in the `regex` header, involves several components, listed in Table 17.4.

C++  
11



If you are not already familiar with using regular expressions, you might want to skim this section to get an idea of the kinds of things regular expressions can do.

**Table 17.4: Regular Expression Library Components**

|                              |                                                                                                               |
|------------------------------|---------------------------------------------------------------------------------------------------------------|
| <code>regex</code>           | Class that represents a regular expression                                                                    |
| <code>regex_match</code>     | Matches a sequence of characters against a regular expression                                                 |
| <code>regex_search</code>    | Finds the first subsequence that matches the regular expression                                               |
| <code>regex_replace</code>   | Replaces a regular expression using a given format                                                            |
| <code>sregex_iterator</code> | Iterator adaptor that calls <code>regex_search</code> to iterate through the matches in a <code>string</code> |
| <code>smatch</code>          | Container class that holds the results of searching a <code>string</code>                                     |
| <code>ssub_match</code>      | Results for a matched subexpression in a <code>string</code>                                                  |

The `regex` class represents a regular expression. Aside from initialization and assignment, `regex` has few operations. The operations on `regex` are listed in Table 17.6 (p. 731).

The functions `regex_match` and `regex_search` determine whether a given character sequence matches a given `regex`. The `regex_match` function returns `true` if the entire input sequence matches the expression; `regex_search` returns `true` if there is a substring in the input sequence that matches. There is also a `regex_replace` function that we'll describe in § 17.3.4 (p. 741).

The arguments to the `regex` functions are described in Table 17.5 (overleaf). These functions return a `bool` and are overloaded: One version takes an additional argument of type `smatch`. If present, these functions store additional information about a successful match in the given `smatch` object.

### 17.3.1 Using the Regular Expression Library

As a fairly simple example, we'll look for words that violate a well-known spelling rule of thumb, "*i* before *e* except after *c*":

```
// find the characters ei that follow a character other than c
string pattern("[^c]ei");
// we want the whole word in which our pattern appears
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern); // construct a regex to find pattern
smatch results; // define an object to hold the results of a search
// define a string that has text that does and doesn't match pattern
string test_str = "receipt freind theif receive";
// use r to find a match to pattern in test_str
if (regex_search(test_str, results, r)) // if there is a match
    cout << results.str() << endl; // print the matching word
```

We start by defining a `string` to hold the regular expression we want to find. The regular expression `[^c]` says we want any character that is not a '`c`', and `[^c]ei` says we want any such letter that is followed by the letters `ei`. This pattern describes strings containing exactly three characters. We want the entire word that contains this pattern. To match the word, we need a regular expression that will match the letters that come before and after our three-letter pattern.

**Table 17.5: Arguments to `regex_search` and `regex_match`**

**Note:** These operations return `bool` indicating whether a match was found.

- (`seq`, `m`, `r`, `mft`) Look for the regular expression in the `regex` object `r` in the character sequence `seq`. `seq` can be a `string`, a pair of iterators denoting a range, or a pointer to a null-terminated character array.  
`m` is a *match* object, which is used to hold details about the match.  
`m` and `seq` must have compatible types (see § 17.3.1 (p. 733)).  
`mft` is an optional `regex_constants::match_flag_type` value.  
These values, listed in Table 17.13 (p. 744), affect the match process.

That regular expression consists of zero or more letters followed by our original three-letter pattern followed by zero or more additional characters. By default, the regular-expression language used by `regex` objects is ECMAScript. In ECMAScript, the pattern `[[:alpha:]]` matches any alphabetic character, and the symbols `+` and `*` signify that we want “one or more” or “zero or more” matches, respectively. Thus, `[[:alpha:]]*` will match zero or more characters.

Having stored our regular expression in `pattern`, we use it to initialize a `regex` object named `r`. We next define a `string` that we’ll use to test our regular expression. We initialize `test_str` with words that match our pattern (e.g., “freind” and “theif”) and words (e.g., “receipt” and “receive”) that don’t. We also define an `smatch` object named `results`, which we will pass to `regex_search`. If a match is found, `results` will hold the details about where the match occurred.

Next we call `regex_search`. If `regex_search` finds a match, it returns `true`. We use the `str` member of `results` to print the part of `test_str` that matched our pattern. The `regex_search` function stops looking as soon as it finds a matching substring in the input sequence. Thus, the output will be

```
freind
```

§ 17.3.2 (p. 734) will show how to find all the matches in the input.

## Specifying Options for a `regex` Object

When we define a `regex` or call `assign` on a `regex` to give it a new value, we can specify one or more flags that affect how the `regex` operates. These flags control the processing done by that object. The last six flags listed in Table 17.6 indicate the language in which the regular expression is written. Exactly one of the flags that specify a language must be set. By default, the `ECMAScript` flag is set, which causes the `regex` to use the ECMA-262 specification, which is the regular expression language that many Web browsers use.

The other three flags let us specify language-independent aspects of the regular-expression processing. For example, we can indicate that we want the regular expression to be matched in a case-independent manner.

As one example, we can use the `icase` flag to find file names that have a particular file extension. Most operating systems recognize extensions in a case-independent manner—we can store a C++ program in a file that ends in `.cc`, or

**Table 17.6: `regex` (and `wregex`) Operations**

|                               |                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>regex r(re)</code>      | <i>re</i> represents a regular expression and can be a string, a pair of iterators denoting a range of characters, a pointer to a null-terminated character array, a character pointer and a count, or a braced list of characters. <i>f</i> are flags that specify how the object will execute. <i>f</i> is set from the values listed below. If <i>f</i> is not specified, it defaults to ECMAScript. |
| <code>r1 = re</code>          | Replace the regular expression in <i>r1</i> with <i>re</i> . <i>re</i> represents a regular expression and can be another <code>regex</code> , a string, a pointer to a null-terminated character array, or a braced list of characters.                                                                                                                                                                |
| <code>r1.assign(re, f)</code> | Same effect as using the assignment operator (=). <i>re</i> and optional flag <i>f</i> same as corresponding arguments to <code>regex</code> constructors.                                                                                                                                                                                                                                              |
| <code>r.mark_count()</code>   | Number of subexpressions (which we'll cover in § 17.3.3 (p. 738)) in <i>r</i> .                                                                                                                                                                                                                                                                                                                         |
| <code>r.flags()</code>        | Returns the flags set for <i>r</i> .                                                                                                                                                                                                                                                                                                                                                                    |

*Note: Constructors and assignment operations may throw exceptions of type `regex_error`.*

#### Flags Specified When a `regex` Is Defined

##### Defined in `regex` and `regex_constants::syntax_option_type`

|                         |                                                                     |
|-------------------------|---------------------------------------------------------------------|
| <code>icase</code>      | Ignore case during the match                                        |
| <code>nosubs</code>     | Don't store subexpression matches                                   |
| <code>optimize</code>   | Favor speed of execution over speed of construction                 |
| <code>ECMAScript</code> | Use grammar as specified by ECMA-262                                |
| <code>basic</code>      | Use POSIX basic regular-expression grammar                          |
| <code>extended</code>   | Use POSIX extended regular-expression grammar                       |
| <code>awk</code>        | Use grammar from the POSIX version of the <code>awk</code> language |
| <code>grep</code>       | Use grammar from the POSIX version of <code>grep</code>             |
| <code>egrep</code>      | Use grammar from the POSIX version of <code>egrep</code>            |

.Cc, or .cC, or .CC. We'll write a regular expression to recognize any of these along with other common file extensions as follows:

```
// one or more alphanumeric characters followed by a '.' followed by "cpp" or "cxx" or "cc"
regex r("[[:alnum:]]+\\".(cpp|cxx|cc)$", regex::icase);
smatch results;
string filename;
while (cin >> filename)
    if (regex_search(filename, results, r))
        cout << results.str() << endl; // print the current match
```

This expression will match a string of one or more letters or digits followed by a period and followed by one of three file extensions. The regular expression will match the file extensions regardless of case.

Just as there are special characters in C++ (§ 2.1.3, p. 39), regular-expression languages typically also have special characters. For example, the dot (.) character usually matches any character. As we do in C++, we can escape the special nature of a character by preceding it with a backslash. Because the backslash is also a special character in C++, we must use a second backslash inside a string literal to indicate to C++ that we want a backslash. Hence, we must write \\ . to represent a regular expression that will match a period.

## Errors in Specifying or Using a Regular Expression

We can think of a regular expression as itself a “program” in a simple programming language. That language is not interpreted by the C++ compiler. Instead, a regular expression is “compiled” at run time when a `regex` object is initialized with or assigned a new pattern. As with any programming language, it is possible that the regular expressions we write can have errors.



It is important to realize that the syntactic correctness of a regular expression is evaluated at run time.

If we make a mistake in writing a regular expression, then at *run time* the library will throw an exception (§ 5.6, p. 193) of type `regex_error`. Like the standard exception types, `regex_error` has a `what` operation that describes the error that occurred (§ 5.6.2, p. 195). A `regex_error` also has a member named `code` that returns a numeric code corresponding to the type of error that was encountered. The values `code` returns are implementation defined. The standard errors that the RE library can throw are listed in Table 17.7.

For example, we might inadvertently omit a bracket in a pattern:

```
try {
    // error: missing close bracket afteralnum; the constructor will throw
    regex r("[[:alnum:]]+\\" .(cpp|cxx|cc)$", regex::icase);
} catch (regex_error e)
{ cout << e.what() << "\nerror: " << e.code() << endl; }
```

When run on our system, this program generates

```
regex_error(error_brack):
The expression contained mismatched [ and ].
code: 4
```

**Table 17.7: Regular Expression Error Conditions**

| Defined in <code>regex</code> and in <code>regex_constants::error_type</code> |                                                                                        |
|-------------------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <code>error_collate</code>                                                    | Invalid collating element request                                                      |
| <code>error_ctype</code>                                                      | Invalid character class                                                                |
| <code>error_escape</code>                                                     | Invalid escape character or trailing escape                                            |
| <code>error_backref</code>                                                    | Invalid back reference                                                                 |
| <code>error_brack</code>                                                      | Mismatched bracket ( [ or ] )                                                          |
| <code>error_paren</code>                                                      | Mismatched parentheses ( ( or ) )                                                      |
| <code>error_brace</code>                                                      | Mismatched brace ( { or } )                                                            |
| <code>error_badbrace</code>                                                   | Invalid range inside a { }                                                             |
| <code>error_range</code>                                                      | Invalid character range (e.g., [z-a])                                                  |
| <code>error_space</code>                                                      | Insufficient memory to handle this regular expression                                  |
| <code>error_badrepeat</code>                                                  | A repetition character (*, ?, +, or {}) was not preceded by a valid regular expression |
| <code>error_complexity</code>                                                 | The requested match is too complex                                                     |
| <code>error_stack</code>                                                      | Insufficient memory to evaluate a match                                                |

Our compiler defines the `code` member to return the position of the error as listed in Table 17.7, counting, as usual, from zero.

#### **ADVICE: AVOID CREATING UNNECESSARY REGULAR EXPRESSIONS**

As we've seen, the "program" that a regular expression represents is compiled at run time, not at compile time. Compiling a regular expression can be a surprisingly slow operation, especially if you're using the extended regular-expression grammar or are using complicated expressions. As a result, constructing a `regex` object and assigning a new regular expression to an existing `regex` can be time-consuming. To minimize this overhead, you should try to avoid creating more `regex` objects than needed. In particular, if you use a regular expression in a loop, you should create it outside the loop rather than recompiling it on each iteration.

## **Regular Expression Classes and the Input Sequence Type**

We can search any of several types of input sequence. The input can be ordinary `char` data or `wchar_t` data and those characters can be stored in a library `string` or in an array of `char` (or the wide character versions, `wstring` or array of `wchar_t`). The RE library defines separate types that correspond to these differing types of input sequences.

For example, the `regex` class holds regular expressions of type `char`. The library also defines a `wregex` class that holds type `wchar_t` and has all the same operations as `regex`. The only difference is that the initializers of a `wregex` must use `wchar_t` instead of `char`.

The match and iterator types (which we will cover in the following sections) are more specific. These types differ not only by the character type, but also by whether the sequence is in a library `string` or an array: `smatch` represents `string` input sequences; `cmatch`, character array sequences; `wsmatch`, wide string (`wstring`) input; and `wcmatch`, arrays of wide characters.

**Table 17.8: Regular Expression Library Classes**

| If Input Sequence Has Type  | Use Regular Expression Classes                                                                            |
|-----------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>string</code>         | <code>regex</code> , <code>smatch</code> , <code>ssub_match</code> , and <code>sregex_iterator</code>     |
| <code>const char*</code>    | <code>regex</code> , <code>cmatch</code> , <code>csub_match</code> , and <code>cregex_iterator</code>     |
| <code>wstring</code>        | <code>wregex</code> , <code>wsmatch</code> , <code>wssub_match</code> , and <code>wsregex_iterator</code> |
| <code>const wchar_t*</code> | <code>wregex</code> , <code>wcmatch</code> , <code>wcsub_match</code> , and <code>wcregex_iterator</code> |

The important point is that the RE library types we use must match the type of the input sequence. Table 17.8 indicates which types correspond to which kinds of input sequences. For example:

```
regex r("[:alnum:]"]+\.\(cpp|cxx|cc)\$", regex::icase);
smatch results; // will match a string input sequence, but not char*
if (regex_search("myfile.cc", results, r)) // error: char* input
    cout << results.str() << endl;
```

The (C++) compiler will reject this code because the type of the match argument and the type of the input sequence do not match. If we want to search a character array, then we must use a `cmatch` object:

```
cmatch results; // will match character array input sequences
if (regex_search("myfile.cc", results, rx))
    cout << results.str() << endl; // print the current match
```

In general, our programs will use `string` input sequences and the corresponding `string` versions of the RE library components.

### EXERCISES SECTION 17.3.1

**Exercise 17.14:** Write several regular expressions designed to trigger various errors. Run your program to see what output your compiler generates for each error.

**Exercise 17.15:** Write a program using the pattern that finds words that violate the “*i* before *e* except after *c*” rule. Have your program prompt the user to supply a word and indicate whether the word is okay or not. Test your program with words that do and do not violate the rule.

**Exercise 17.16:** What would happen if your `regex` object in the previous program were initialized with “`[^c]ei`”? Test your program using that pattern to see whether your expectations were correct.

## 17.3.2 The Match and Regex Iterator Types

The program on page 729 that found violations of the “*i* before *e* except after *c*” grammar rule printed only the first match in its input sequence. We can get all the matches by using an `sregex_iterator`. The regex iterators are iterator adaptors (§ 9.6, p. 368) that are bound to an input sequence and a `regex` object. As described in Table 17.8 (on the previous page), there are specific regex iterator types that correspond to each of the different types of input sequences. The iterator operations are described in Table 17.9 (p. 736).

When we bind an `sregex_iterator` to a `string` and a `regex` object, the iterator is automatically positioned on the first match in the given `string`. That is, the `sregex_iterator` constructor calls `regex_search` on the given `string` and `regex`. When we dereference the iterator, we get an `smatch` object corresponding to the results from the most recent search. When we increment the iterator, it calls `regex_search` to find the next match in the input `string`.

### Using an `sregex_iterator`

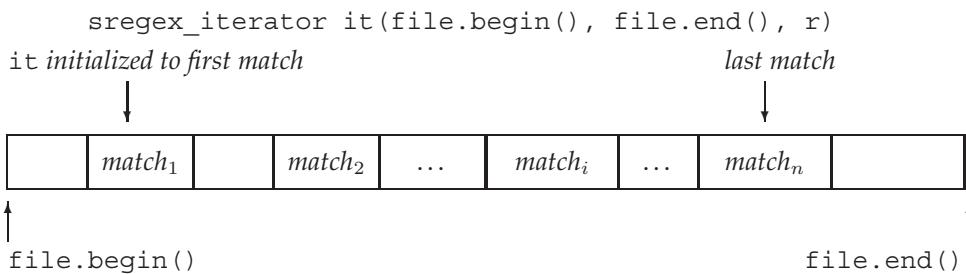
As an example, we’ll extend our program to find all the violations of the “*i* before *e* except after *c*” grammar rule in a file of text. We’ll assume that the `string` named `file` holds the entire contents of the input file that we want to search. This version of the program will use the same pattern as our original one, but will use a `sregex_iterator` to do the search:

```
// find the characters ei that follow a character other than c
string pattern("[^c]ei");
// we want the whole word in which our pattern appears
pattern = "[[:alpha:]]*" + pattern + "[[:alpha:]]*";
regex r(pattern, regex::icase); // we'll ignore case in doing the match
// it will repeatedly call regex_search to find all matches in file
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it)
    cout << it->str() << endl; // matched word
```

The for loop iterates through each match to `r` inside `file`. The initializer in the for defines `it` and `end_it`. When we define `it`, the `sregex_iterator` constructor calls `regex_search` to position `it` on the first match in `file`. The empty `sregex_iterator`, `end_it`, acts as the off-the-end iterator. The increment in the for “advances” the iterator by calling `regex_search`. When we dereference the iterator, we get an `smatch` object representing the current match. We call the `str` member of the match to print the matching word.

We can think of this loop as jumping from match to match as illustrated in Figure 17.1.

**Figure 17.1: Using an `sregex_iterator`**



## Using the Match Data

If we run this loop on `test_str` from our original program, the output would be

```
freind
theif
```

However, finding just the words that match our expression is not so useful. If we ran the program on a larger input sequence—for example, on the text of this chapter—we’d want to see the context within which the word occurs, such as

```
hey read or write according to the type
    >>> being <<
handled. The input operators ignore whi
```

In addition to letting us print the part of the input string that was matched, the match classes give us more detailed information about the match. The operations on these types are listed in Table 17.10 (p. 737) and Table 17.11 (p. 741).

**Table 17.9: `sregex_iterator` Operations**

**These operations also apply to `cregex_iterator`,  
`wsregex_iterator`, and `wcregex_iterator`**

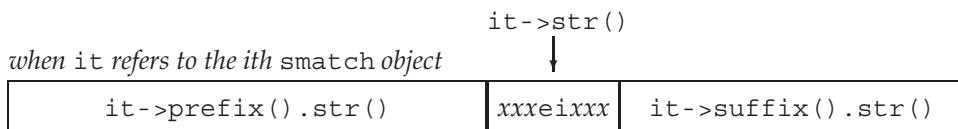
|                                                        |                                                                                                                                                                                                 |
|--------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>sregex_iterator</code> <code>it(b, e, r);</code> | it is an <code>sregex_iterator</code> that iterates through the string denoted by iterators b and e. Calls <code>regex_search(b, e, r)</code> to position it on the first match in the input.   |
| <code>sregex_iterator</code> <code>end;</code>         | Off-the-end iterator for <code>sregex_iterator</code> .                                                                                                                                         |
| <code>*it</code>                                       | Returns a reference to the <code>smatch</code> object or a pointer to the <code>smatch</code> object from the most recent call to <code>regex_search</code> .                                   |
| <code>it-&gt;</code>                                   | Calls <code>regex_search</code> on the input sequence starting just after the current match. The prefix version returns a reference to the incremented iterator; postfix returns the old value. |
| <code>it++</code>                                      |                                                                                                                                                                                                 |
| <code>it1 == it2</code>                                | Two <code>sregex_iterator</code> s are equal if they are both the off-the-end iterator.                                                                                                         |
| <code>it1 != it2</code>                                | Two non-end iterators are equal if they are constructed from the same input sequence and <code>regex</code> object.                                                                             |

We'll have more to say about the `smatch` and `ssub_match` types in the next section. For now, what we need to know is that these types let us see the context of a match. The match types have members named `prefix` and `suffix`, which return a `ssub_match` object representing the part of the input sequence ahead of and after the current match, respectively. A `ssub_match` object has members named `str` and `length`, which return the matched string and size of that string, respectively. We can use these operations to rewrite the loop of our grammar program:

```
// same for loop header as before
for (sregex_iterator it(file.begin(), file.end(), r), end_it;
     it != end_it; ++it) {
    auto pos = it->prefix().length(); // size of the prefix
    pos = pos > 40 ? pos - 40 : 0; // we want up to 40 characters
    cout << it->prefix().str().substr(pos) // last part of the prefix
        << "\n\t>>> " << it->str() << " <<<\n" // matched word
        << it->suffix().str().substr(0, 40) // first part of the suffix
        << endl;
}
```

The loop itself operates the same way as our previous program. What's changed is the processing inside the `for`, which is illustrated in Figure 17.2.

We call `prefix`, which returns an `ssub_match` object that represents the part of `file` ahead of the current match. We call `length` on that `ssub_match` to find out how many characters are in the part of `file` ahead of the match. Next we adjust `pos` to be the index of the character 40 from the end of the prefix. If the prefix has fewer than 40 characters, we set `pos` to 0, which means we'll print the entire prefix. We use `substr` (§ 9.5.1, p. 361) to print from the given position to the end of the prefix.

**Figure 17.2: The `smatch` Object Representing a Particular Match**

Having printed the characters that precede the match, we next print the match itself with some additional formatting so that the matched word will stand out in the output. After printing the matched portion, we print (up to) the first 40 characters in the part of `file` that comes after this match.

**Table 17.10: `smatch` Operations**

These operations also apply to the `cmatch`, `wsmatch`, `wcmatch` and the corresponding `csub_match`, `wssub_match`, and `wcsub_match` types.

|                            |                                                                                                                                                                                                           |
|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>m.ready()</code>     | true if <code>m</code> has been set by a call to <code>regex_search</code> or <code>regex_match</code> ; false otherwise. Operations on <code>m</code> are undefined if <code>ready</code> returns false. |
| <code>m.size()</code>      | Zero if the match failed; otherwise, one plus the number of subexpressions in the most recently matched regular expression.                                                                               |
| <code>m.empty()</code>     | true if <code>m.size()</code> is zero.                                                                                                                                                                    |
| <code>m.prefix()</code>    | An <code>ssub_match</code> representing the sequence before the match.                                                                                                                                    |
| <code>m.suffix()</code>    | An <code>ssub_match</code> representing the part after the end of the match.                                                                                                                              |
| <code>m.format(...)</code> | See Table 17.12 (p. 742).                                                                                                                                                                                 |

In the operations that take an index, `n` defaults to zero and must be less than `m.size()`.

The first submatch (the one with index 0) represents the overall match.

|                                               |                                                                                                                                                                                                                    |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>m.length(n)</code>                      | Size of the <code>n</code> th matched subexpression.                                                                                                                                                               |
| <code>m.position(n)</code>                    | Distance of the <code>n</code> th subexpression from the start of the sequence.                                                                                                                                    |
| <code>m.str(n)</code>                         | The matched string for the <code>n</code> th subexpression.                                                                                                                                                        |
| <code>m[n]</code>                             | <code>ssub_match</code> object corresponding to the <code>n</code> th subexpression.                                                                                                                               |
| <code>m.begin()</code> , <code>m.end()</code> | Iterators across the <code>sub_match</code> elements in <code>m</code> . As usual, <code>cbegin</code> , <code>cbegin()</code> , <code>m.cend()</code> and <code>cend</code> return <code>const_iterators</code> . |

## EXERCISES SECTION 17.3.2

**Exercise 17.17:** Update your program so that it finds all the words in an input sequence that violate the “ei” grammar rule.

**Exercise 17.18:** Revise your program to ignore words that contain “ei” but are not misspellings, such as “albeit” and “neighbor.”

### 17.3.3 Using Subexpressions

A pattern in a regular expression often contains one or more **subexpressions**. A subexpression is a part of the pattern that itself has meaning. Regular-expression grammars typically use parentheses to denote subexpressions.

As an example, the pattern that we used to match C++ files (§ 17.3.1, p. 730) used parentheses to group the possible file extensions. Whenever we group alternatives using parentheses, we are also declaring that those alternatives form a subexpression. We can rewrite that expression so that it gives us access to the file name, which is the part of the pattern that precedes the period, as follows:

```
// r has two subexpressions: the first is the part of the file name before the period
// the second is the file extension
regex r("([:alnum:]+)\.\.(cpp|cxx|cc)$", regex::icase);
```

Our pattern now has two parenthesized subexpressions:

- `([:alnum:]+)`, which is a sequence of one or more characters
- `(cpp|cxx|cc)`, which is the file extension

We can also rewrite the program from § 17.3.1 (p. 730) to print just the file name by changing the output statement:

```
if (regex_search(filename, results, r))
    cout << results.str(1) << endl; // print the first subexpression
```

As in our original program, we call `regex_search` to look for our pattern `r` in the string named `filename`, and we pass the `smatch` object `results` to hold the results of the match. If the call succeeds, then we print the results. However, in this program, we print `str(1)`, which is the match for the first subexpression.

In addition to providing information about the overall match, the match objects provide access to each matched subexpression in the pattern. The submatches are accessed positionally. The first submatch, which is at position 0, represents the match for the entire pattern. Each subexpression appears in order thereafter. Hence, the file name, which is the first subexpression in our pattern, is at position 1, and the file extension is in position 2.

For example, if the file name is `foo.cpp`, then `results.str(0)` will hold `foo.cpp`; `results.str(1)` will be `foo`; and `results.str(2)` will be `cpp`. In this program, we want the part of the name before the period, which is the first subexpression, so we print `results.str(1)`.

### Subexpressions for Data Validation

One common use for subexpressions is to validate data that must match a specific format. For example, U.S. phone numbers have ten digits, consisting of an area code and a seven-digit local number. The area code is often, but not always, enclosed in parentheses. The remaining seven digits can be separated by a dash, a dot, or a space; or not separated at all. We might want to allow data with any of these formats and reject numbers in other forms. We'll do a two-step process: First,

we'll use a regular expression to find sequences that might be phone numbers and then we'll call a function to complete the validation of the data.

Before we write our phone number pattern, we need to describe a few more aspects of the ECMAScript regular-expression language:

- $\backslash\{d\}$  represents a single digit and  $\backslash\{d\}\{n\}$  represents a sequence of  $n$  digits. (E.g.,  $\backslash\{d\}\{3\}$  matches a sequence of three digits.)
- A collection of characters inside square brackets allows a match to any of those characters. (E.g.,  $[-\cdot]$  matches a dash, a dot, or a space. Note that a dot has no special meaning inside brackets.)
- A component followed by '?' is optional. (E.g.,  $\backslash\{d\}\{3\}[-\cdot]?\backslash\{d\}\{4\}$  matches three digits followed by an optional dash, period, or space, followed by four more digits. This pattern would match 555-0132 or 555.0132 or 555 0132 or 5550132.)
- Like C++, ECMAScript uses a backslash to indicate that a character should represent itself, rather than its special meaning. Because our pattern includes parentheses, which are special characters in ECMAScript, we must represent the parentheses that are part of our pattern as  $\backslash($  or  $\backslash)$ .

Because backslash is a special character in C++, each place that a  $\backslash$  appears in the pattern, we must use a second backslash to indicate to C++ that we want a backslash. Hence, we write  $\backslash\backslash\{d\}\{3\}$  to represent the regular expression  $\backslash\{d\}\{3\}$ .

In order to validate our phone numbers, we'll need access to the components of the pattern. For example, we'll want to verify that if a number uses an opening parenthesis for the area code, it also uses a close parenthesis after the area code. That is, we'd like to reject a number such as (908.555.1800.

To get at the components of the match, we need to define our regular expression using subexpressions. Each subexpression is marked by a pair of parentheses:

```
// our overall expression has seven subexpressions: (ddd) separator ddd separator dddd
// subexpressions 1, 3, 4, and 6 are optional; 2, 5, and 7 hold the number
"(\backslash()?) (\backslash\{d\}\{3\}) (\backslash())? ([-\cdot])? (\backslash\{d\}\{3\}) ([-\cdot])? (\backslash\{d\}\{4\})";
```

Because our pattern uses parentheses, and because we must escape backslashes, this pattern can be hard to read (and write!). The easiest way to read it is to pick off each (parenthesized) subexpression:

1.  $(\backslash()?)$  an optional open parenthesis for the area code
2.  $(\backslash\{d\}\{3\})$  the area code
3.  $(\backslash())?$  an optional close parenthesis for the area code
4.  $([-\cdot])?$  an optional separator after the area code
5.  $(\backslash\{d\}\{3\})$  the next three digits of the number
6.  $([-\cdot])?$  another optional separator
7.  $(\backslash\{d\}\{4\})$  the final four digits of the number

The following code uses this pattern to read a file and find data that match our overall phone pattern. It will call a function named `valid` to check whether the number has a valid format:

```
string phone =
    "(\\()?(\\d{3})(\\))?([- . ])?(\\d{3})([- . ]?)?(\\d{4})";
regex r(phone); // a regex to find our pattern
smatch m;
string s;
// read each record from the input file
while (getline(cin, s)) {
    // for each matching phone number
    for (sregex_iterator it(s.begin(), s.end(), r), end_it;
         it != end_it; ++it)
        // check whether the number's formatting is valid
        if (valid(*it))
            cout << "valid: " << it->str() << endl;
        else
            cout << "not valid: " << it->str() << endl;
}
```

## Using the Submatch Operations

We'll use submatch operations, which are outlined in Table 17.11, to write the `valid` function. It is important to keep in mind that our pattern has seven subexpressions. As a result, each `smatch` object will contain eight `ssub_match` elements. The element at [0] represents the overall match; the elements [1]...[7] represent each of the corresponding subexpressions.

When we call `valid`, we know that we have an overall match, but we do not know which of our optional subexpressions were part of that match. The `matched` member of the `ssub_match` corresponding to a particular subexpression is `true` if that subexpression is part of the overall match.

In a valid phone number, the area code is either fully parenthesized or not parenthesized at all. Therefore, the work `valid` does depends on whether the number starts with a parenthesis or not:

```
bool valid(const smatch& m)
{
    // if there is an open parenthesis before the area code
    if (m[1].matched)
        // the area code must be followed by a close parenthesis
        // and followed immediately by the rest of the number or a space
        return m[3].matched
        && (m[4].matched == 0 || m[4].str() == " ");
    else
        // then there can't be a close after the area code
        // the delimiters between the other two components must match
        return !m[3].matched
        && m[4].str() == m[6].str();
}
```

**Table 17.11: Submatch Operations**

|                                                                                                                                             |                                                                                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>Note: These operations apply to <code>ssub_match</code>, <code>csub_match</code>, <code>wssub_match</code>, <code>wcsub_match</code></i> |                                                                                                                                                                                                                  |
| <code>matched</code>                                                                                                                        | A public <code>bool</code> data member that indicates whether this <code>ssub_match</code> was matched.                                                                                                          |
| <code>first</code>                                                                                                                          | public data members that are iterators to the start and one past the end of the matching sequence. If there was no match, then <code>first</code> and <code>second</code> are equal.                             |
| <code>second</code>                                                                                                                         |                                                                                                                                                                                                                  |
| <code>length()</code>                                                                                                                       | The size of this match. Returns 0 if <code>matched</code> is <code>false</code> .                                                                                                                                |
| <code>str()</code>                                                                                                                          | Returns a string containing the matched portion of the input. Returns the empty string if <code>matched</code> is <code>false</code> .                                                                           |
| <code>s = ssub</code>                                                                                                                       | Convert the <code>ssub_match</code> object <code>ssub</code> to the <code>string</code> <code>s</code> . Equivalent to <code>s = ssub.str()</code> . The conversion operator is not explicit (§ 14.9.1, p. 581). |

We start by checking whether the first subexpression (i.e., the open parenthesis) matched. That subexpression is in `m[1]`. If it matched, then the number starts with an open parenthesis. In this case, the overall number is valid if the subexpression following the area code also matched (meaning that there was a close parenthesis after the area code). Moreover, if the number is correctly parenthesized, then the next character must be a space or the first digit in the next part of the number.

If `m[1]` didn't match (i.e., there was no open parenthesis), the subexpression following the area code must also be empty. If it's empty, then the number is valid if the remaining separators are equal and not otherwise.

### EXERCISES SECTION 17.3.3

**Exercise 17.19:** Why is it okay to call `m[4].str()` without first checking whether `m[4]` was matched?

**Exercise 17.20:** Write your own version of the program to validate phone numbers.

**Exercise 17.21:** Rewrite your phone number program from § 8.3.2 (p. 323) to use the `valid` function defined in this section.

**Exercise 17.22:** Rewrite your phone program so that it allows any number of whitespace characters to separate the three parts of a phone number.

**Exercise 17.23:** Write a regular expression to find zip codes. A zip code can have five or nine digits. The first five digits can be separated from the remaining four by a dash.

### 17.3.4 Using `regex_replace`

Regular expressions are often used when we need not only to find a given sequence but also to replace that sequence with another one. For example, we might want to translate U.S. phone numbers into the form "ddd.ddd.dddd," where the area code and next three digits are separated by a dot.

When we want to find and replace a regular expression in the input sequence, we call `regex_replace`. Like the search functions, `regex_replace`, which is described in Table 17.12, takes an input character sequence and a `regex` object. We must also pass a string that describes the output we want.

We compose a replacement string by including the characters we want, intermixed with subexpressions from the matched substring. In this case, we want to use the second, fifth, and seventh subexpressions in our replacement string. We'll ignore the first, third, fourth, and sixth, because these were used in the original formatting of the number but are not part of our replacement format. We refer to a particular subexpression by using a \$ symbol followed by the index number for a subexpression:

```
string fmt = "$2.$5.$7"; // reformat numbers to dddddd.ddd
```

We can use our regular-expression pattern and the replacement string as follows:

```
regex r(phone); // a regex to find our pattern
string number = "(908) 555-0132";
cout << regex_replace(number, r, fmt) << endl;
```

The output from this program is

```
908.555.0132
```

**Table 17.12: Regular Expression Replace Operations**

|                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>m.format(dest, fmt, mft)</code>                               | Produces formatted output using the format string <code>fmt</code> , the match in <code>m</code> , and the optional <code>match_flag_type</code> flags in <code>mft</code> . The first version writes to the output iterator <code>dest</code> (§ 10.5.1, p. 410) and takes <code>fmt</code> that is either a <code>string</code> or a pair of pointers denoting a range in a character array. The second version returns a <code>string</code> that holds the output and takes <code>fmt</code> that is a <code>string</code> or a pointer to a null-terminated character array. <code>mft</code> defaults to <code>format_default</code> .                                                                                                              |
| <code>regex_replace</code><br><code>(dest, seq, r, fmt, mft)</code> | Iterates through <code>seq</code> , using <code>regex_search</code> to find successive matches to <code>regex r</code> . Uses the format string <code>fmt</code> and optional <code>match_flag_type</code> flags in <code>mft</code> to produce its output. The first version writes to the output iterator <code>dest</code> , and takes a pair of iterators to denote <code>seq</code> . The second returns a <code>string</code> that holds the output and <code>seq</code> can be either a <code>string</code> or a pointer to a null-terminated character array. In all cases, <code>fmt</code> can be either a <code>string</code> or a pointer to a null-terminated character array, and <code>mft</code> defaults to <code>match_default</code> . |
| <code>regex_replace</code><br><code>(seq, r, fmt, mft)</code>       |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |

## Replacing Only Part of the Input Sequence

A more interesting use of our regular-expression processing would be to replace phone numbers that are embedded in a larger file. For example, we might have a file of names and phone number that had data like this:

```
morgan (201) 555-0168 862-555-0123
drew (973)555.0130
lee (609) 555-0132 2015550175 800.555-0100
```

that we want to transform to data like this:

```
morgan 201.555.0168 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0100
```

We can generate this transformation with the following program:

```
int main()
{
    string phone =
        "(\\(\\)?(\\d{3})\\)(\\))?([- .])?(\\d{3})([- .])?(\\d{4})";
    regex r(phone); // a regex to find our pattern
    smatch m;
    string s;
    string fmt = "$2.$5.$7"; // reformat numbers to ddd.ddd.dddd
    // read each record from the input file
    while (getline(cin, s))
        cout << regex_replace(s, r, fmt) << endl;
    return 0;
}
```

We read each record into `s` and hand that record to `regex_replace`. This function finds and transforms *all* the matches in its input sequence.

## Flags to Control Matches and Formatting

Just as the library defines flags to direct how to process a regular expression, the library also defines flags that we can use to control the match process or the formatting done during a replacement. These values are listed in Table 17.13 (overleaf). These flags can be passed to the `regex_search` or `regex_match` functions or to the `format` members of class `smatch`.

The match and format flags have type `match_flag_type`. These values are defined in a namespace named `regex_constants`. Like `placeholders`, which we used with `bind` (§ 10.3.4, p. 399), `regex_constants` is a namespace defined inside the `std` namespace. To use a name from `regex_constants`, we must qualify that name with the names of both namespaces:

```
using std::regex_constants::format_no_copy;
```

This declaration says that when our code uses `format_no_copy`, we want the object of that name from the namespace `std::regex_constants`. We can instead provide the alternative form of using that we will cover in § 18.2.2 (p. 792):

```
using namespace std::regex_constants;
```

**Table 17.13: Match Flags**

| Defined in <code>regex_constants::match_flag_type</code> |                                                              |
|----------------------------------------------------------|--------------------------------------------------------------|
| <code>match_default</code>                               | Equivalent to <code>format_default</code>                    |
| <code>match_not_bol</code>                               | Don't treat the first character as the beginning of the line |
| <code>match_not_eol</code>                               | Don't treat the last character as the end of the line        |
| <code>match_not_bow</code>                               | Don't treat the first character as the beginning of a word   |
| <code>match_not_eow</code>                               | Don't treat the last character as the end of a word          |
| <code>match_any</code>                                   | If there is more than one match, any match can be returned   |
| <code>match_not_null</code>                              | Don't match an empty sequence                                |
| <code>match_continuous</code>                            | The match must begin with the first character in the input   |
| <code>match_prev_avail</code>                            | The input sequence has characters before the first           |
| <code>format_default</code>                              | Replacement string uses the ECMAScript rules                 |
| <code>format_sed</code>                                  | Replacement string uses the rules from POSIX sed             |
| <code>format_no_copy</code>                              | Don't output the unmatched parts of the input                |
| <code>format_first_only</code>                           | Replace only the first occurrence                            |

## Using Format Flags

By default, `regex_replace` outputs its entire input sequence. The parts that don't match the regular expression are output without change; the parts that do match are formatted as indicated by the given format string. We can change this default behavior by specifying `format_no_copy` in the call to `regex_replace`:

```
// generate just the phone numbers: use a new format string
string fmt2 = "$2.$5.$7 "; // put space after the last number as a separator
// tell regex_replace to copy only the text that it replaces
cout << regex_replace(s, r, fmt2, format_no_copy) << endl;
```

Given the same input, this version of the program generates

```
201.555.0168 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0100
```

### EXERCISES SECTION 17.3.4

**Exercise 17.24:** Write your own version of the program to reformat phone numbers.

**Exercise 17.25:** Rewrite your phone program so that it writes only the first phone number for each person.

**Exercise 17.26:** Rewrite your phone program so that it writes only the second and subsequent phone numbers for people with more than one phone number.

**Exercise 17.27:** Write a program that reformats a nine-digit zip code as dddd-dddd.

## 17.4 Random Numbers

Programs often need a source of random numbers. Prior to the new standard, both C and C++ relied on a simple C library function named `rand`. That function produces pseudorandom integers that are uniformly distributed in the range from 0 to a system-dependent maximum value that is at least 32767.

C++  
11

The `rand` function has several problems: Many, if not most, programs need random numbers in a different range from the one produced by `rand`. Some applications require random floating-point numbers. Some programs need numbers that reflect a nonuniform distribution. Programmers often introduce nonrandomness when they try to transform the range, type, or distribution of the numbers generated by `rand`.

The random-number library, defined in the `random` header, solves these problems through a set of cooperating classes: **random-number engines** and **random-number distribution classes**. These classes are described in Table 17.14. An engine generates a sequence of unsigned random numbers. A distribution uses an engine to generate random numbers of a specified type, in a given range, distributed according to a particular probability distribution.



C++ programs should not use the library `rand` function. Instead, they should use the `default_random_engine` along with an appropriate distribution object.

**Table 17.14: Random Number Library Components**

|              |                                                                                               |
|--------------|-----------------------------------------------------------------------------------------------|
| Engine       | Types that generate a sequence of random unsigned integers                                    |
| Distribution | Types that use an engine to return numbers according to a particular probability distribution |

### 17.4.1 Random-Number Engines and Distribution

The random-number engines are function-object classes (§ 14.8, p. 571) that define a call operator that takes no arguments and returns a random `unsigned` number. We can generate raw random numbers by calling an object of a random-number engine type:

```
default_random_engine e; // generates random unsigned integers
for (size_t i = 0; i < 10; ++i)
    // e() "calls" the object to produce the next random number
    cout << e() << " ";
```

On our system, this program generates:

```
16807 282475249 1622650073 984943658 1144108930 470211272 ...
```

Here, we defined an object named `e` that has type `default_random_engine`. Inside the `for`, we call the object `e` to obtain the next random number.

The library defines several random-number engines that differ in terms of their performance and quality of randomness. Each compiler designates one of these engines as the `default_random_engine` type. This type is intended to be the engine with the most generally useful properties. Table 17.15 lists the engine operations and the engine types defined by the standard are listed in § A.3.2 (p. 884).

For most purposes, the output of an engine is not directly usable, which is why we described them earlier as raw random numbers. The problem is that the numbers usually span a range that differs from the one we need. *Correctly* transforming the range of a random number is surprisingly hard.

## Distribution Types and Engines

To get a number in a specified range, we use an object of a distribution type:

```
// uniformly distributed from 0 to 9 inclusive
uniform_int_distribution<unsigned> u(0, 9);
default_random_engine e; // generates unsigned random integers
for (size_t i = 0; i < 10; ++i)
    // u uses e as a source of numbers
    // each call returns a uniformly distributed value in the specified range
    cout << u(e) << " ";
```

This code produces output such as

```
0 1 7 4 5 2 0 6 6 9
```

Here we define `u` as a `uniform_int_distribution<unsigned>`. That type generates uniformly distributed `unsigned` values. When we define an object of this type, we can supply the minimum and maximum values we want. In this program, `u(0, 9)` says that we want numbers to be in the range 0 to 9 *inclusive*. The random number distributions use inclusive ranges so that we can obtain every possible value of the given integral type.

Like the engine types, the distribution types are also function-object classes. The distribution types define a call operator that takes a random-number engine as its argument. The distribution object uses its engine argument to produce random numbers that the distribution object maps to the specified distribution.

Note that we pass the engine object itself, `u(e)`. Had we written the call as `u(e())`, we would have tried to pass the next value generated by `e` to `u`, which would be a compile-time error. We pass the engine, not the next result of the engine, because some distributions may need to call the engine more than once.



When we refer to a **random-number generator**, we mean the combination of a distribution object with an engine.

## Comparing Random Engines and the `rand` Function

For readers familiar with the C library `rand` function, it is worth noting that the output of calling a `default_random_engine` object is similar to the output of `rand`. Engines deliver `unsigned` integers in a system-defined range. The range

for `rand` is 0 to `RAND_MAX`. The range for an engine type is returned by calling the `min` and `max` members on an object of that type:

```
cout << "min: " << e.min() << " max: " << e.max() << endl;
```

On our system this program produces the following output:

```
min: 1 max: 2147483646
```

**Table 17.15: Random Number Engine Operations**

|                                  |                                                                                                     |
|----------------------------------|-----------------------------------------------------------------------------------------------------|
| <code>Engine e;</code>           | Default constructor; uses the default seed for the engine type                                      |
| <code>Engine e(s);</code>        | Uses the integral value <code>s</code> as the seed                                                  |
| <code>e.seed(s)</code>           | Reset the state of the engine using the seed <code>s</code>                                         |
| <code>e.min()</code>             | The smallest and largest numbers this generator will generate                                       |
| <code>e.max()</code>             |                                                                                                     |
| <code>Engine::result_type</code> | The unsigned integral type this engine generates                                                    |
| <code>e.discard(u)</code>        | Advance the engine by <code>u</code> steps; <code>u</code> has type <code>unsigned long long</code> |

## Engines Generate a Sequence of Numbers

Random number generators have one property that often confuses new users: Even though the numbers that are generated appear to be random, a given generator returns the same sequence of numbers each time it is run. The fact that the sequence is unchanging is very helpful during testing. On the other hand, programs that use random-number generators have to take this fact into account.

As one example, assume we need a function that will generate a `vector` of 100 random integers uniformly distributed in the range from 0 to 9. We might think we'd write this function as follows:

```
// almost surely the wrong way to generate a vector of random integers
// output from this function will be the same 100 numbers on every call!
vector<unsigned> bad_randVec()
{
    default_random_engine e;
    uniform_int_distribution<unsigned> u(0, 9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

However, this function will return the same `vector` every time it is called:

```
vector<unsigned> v1(bad_randVec());
vector<unsigned> v2(bad_randVec());
// will print equal
cout << ((v1 == v2) ? "equal" : "not equal") << endl;
```

This code will print `equal` because the vectors `v1` and `v2` have the same values.

The right way to write our function is to make the engine and associated distribution objects `static` (§ 6.1.1, p. 205):

```
// returns a vector of 100 uniformly distributed random numbers
vector<unsigned> good_randVec()
{
    // because engines and distributions retain state, they usually should be
    // defined as static so that new numbers are generated on each call
    static default_random_engine e;
    static uniform_int_distribution<unsigned> u(0, 9);
    vector<unsigned> ret;
    for (size_t i = 0; i < 100; ++i)
        ret.push_back(u(e));
    return ret;
}
```

Because `e` and `u` are `static`, they will hold their state across calls to the function. The first call will use the first 100 random numbers from the sequence `u(e)` generates, the second call will get the next 100, and so on.



A given random-number generator always produces the same sequence of numbers. A function with a local random-number generator should make that generator (both the engine and distribution objects) `static`. Otherwise, the function will generate the identical sequence on each call.

## Seeding a Generator

The fact that a generator returns the same sequence of numbers is helpful during debugging. However, once our program is tested, we often want to cause each run of the program to generate different random results. We do so by providing a `seed`. A seed is a value that an engine can use to cause it to start generating numbers at a new point in its sequence.

We can seed an engine in one of two ways: We can provide the seed when we create an engine object, or we can call the engine's `seed` member:

```
default_random_engine e1;           // uses the default seed
default_random_engine e2(2147483646); // use the given seed value
// e3 and e4 will generate the same sequence because they use the same seed
default_random_engine e3;           // uses the default seed value
e3.seed(32767);                  // call seed to set a new seed value
default_random_engine e4(32767);   // set the seed value to 32767
for (size_t i = 0; i != 100; ++i) {
    if (e1() == e2())
        cout << "unseeded match at iteration: " << i << endl;
    if (e3() != e4())
        cout << "seeded differs at iteration: " << i << endl;
}
```

Here we define four engines. The first two, `e1` and `e2`, have different seeds and

should generate different sequences. The second two, `e3` and `e4`, have the same seed value. These two objects *will* generate the same sequence.

Picking a good seed, like most things about generating good random numbers, is surprisingly hard. Perhaps the most common approach is to call the system `time` function. This function, defined in the `ctime` header, returns the number of seconds since a given epoch. The `time` function takes a single parameter that is a pointer to a structure into which to write the time. If that pointer is null, the function just returns the time:

```
default_random_engine e1(time(0)); // a somewhat random seed
```

Because `time` returns time as the number of seconds, this seed is useful only for applications that generate the seed at second-level, or longer, intervals.



Using `time` as a seed usually doesn't work if the program is run repeatedly as part of an automated process; it might wind up with the same seed several times.

## EXERCISES SECTION 17.4.1

**Exercise 17.28:** Write a function that generates and returns a uniformly distributed random `unsigned int` each time it is called.

**Exercise 17.29:** Allow the user to supply a seed as an optional argument to the function you wrote in the previous exercise.

**Exercise 17.30:** Revise your function again this time to take a minimum and maximum value for the numbers that the function should return.

## 17.4.2 Other Kinds of Distributions

The engines produce `unsigned` numbers, and each number in the engine's range has the same likelihood of being generated. Applications often need numbers of different types or distributions. The library handles both these needs by defining different distributions that, when used with an engine, produce the desired results. Table 17.16 (overleaf) lists the operations supported by the distribution types.

### Generating Random Real Numbers

Programs often need a source of random floating-point values. In particular, programs frequently need random numbers between zero and one.

The most common, *but incorrect*, way to obtain a random floating-point from `rand` is to divide the result of `rand()` by `RAND_MAX`, which is a system-defined upper limit that is the largest random number that `rand` can return. This technique is incorrect because random integers usually have less precision than floating-point numbers, in which case there are some floating-point values that will never be produced as output.

With the new library facilities, we can easily obtain a floating-point random number. We define an object of type `uniform_real_distribution` and let the library handle mapping random integers to random floating-point numbers. As we did for `uniform_int_distribution`, we specify the minimum and maximum values when we define the object:

```
default_random_engine e; // generates unsigned random integers
// uniformly distributed from 0 to 1 inclusive
uniform_real_distribution<double> u(0, 1);
for (size_t i = 0; i < 10; ++i)
    cout << u(e) << " ";
```

This code is nearly identical to the previous program that generated unsigned values. However, because we used a different distribution type, this version generates different results:

```
0.131538 0.45865 0.218959 0.678865 0.934693 0.519416 ...
```

**Table 17.16: Distribution Operations**

|                        |                                                                                                                                                                                                              |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>Dist d;</code>   | Default constructor; makes <code>d</code> ready to use.<br>Other constructors depend on the type of <code>Dist</code> ; see § A.3 (p. 882).<br>The distribution constructors are explicit (§ 7.5.4, p. 296). |
| <code>d(e)</code>      | Successive calls with the same <code>e</code> produce a sequence of random numbers according to the distribution type of <code>d</code> ; <code>e</code> is a random-number engine object.                   |
| <code>d.min()</code>   | Return the smallest and largest numbers <code>d(e)</code> will generate.                                                                                                                                     |
| <code>d.max()</code>   |                                                                                                                                                                                                              |
| <code>d.reset()</code> | Reestablish the state of <code>d</code> so that subsequent uses of <code>d</code> don't depend on values <code>d</code> has already generated.                                                               |

## Using the Distribution's Default Result Type

With one exception, which we'll cover in § 17.4.2 (p. 752), the distribution types are templates that have a single template type parameter that represents the type of the numbers that the distribution generates. These types always generate either a floating-point type or an integral type.

Each distribution template has a default template argument (§ 16.1.3, p. 670). The distribution types that generate floating-point values generate `double` by default. Distributions that generate integral results use `int` as their default. Because the distribution types have only one template parameter, when we want to use the default we must remember to follow the template's name with empty angle brackets to signify that we want the default (§ 16.1.3, p. 671):

```
// empty <> signify we want to use the default result type
uniform_real_distribution<> u(0, 1); // generates double by default
```

## Generating Numbers That Are Not Uniformly Distributed

In addition to correctly generating numbers in a specified range, another advantage of the new library is that we can obtain numbers that are nonuniformly distributed. Indeed, the library defines 20 distribution types! These types are listed in § A.3 (p. 882).

As an example, we'll generate a series of normally distributed values and plot the resulting distribution. Because `normal_distribution` generates floating-point numbers, our program will use the `lround` function from the `cmath` header to round each result to its nearest integer. We'll generate 200 numbers centered around a mean of 4 with a standard deviation of 1.5. Because we're using a normal distribution, we can expect all but about 1 percent of the generated numbers to be in the range from 0 to 8, inclusive. Our program will count how many values appear that map to the integers in this range:

```
default_random_engine e;           // generates random integers
normal_distribution<> n(4, 1.5); // mean 4, standard deviation 1.5
vector<unsigned> vals(9);        // nine elements each 0
for (size_t i = 0; i != 200; ++i) {
    unsigned v = lround(n(e));    // round to the nearest integer
    if (v < vals.size())          // if this result is in range
        ++vals[v];               // count how often each number appears
}
for (size_t j = 0; j != vals.size(); ++j)
    cout << j << ":" << string(vals[j], '*') << endl;
```

We start by defining our random generator objects and a vector named `vals`. We'll use `vals` to count how often each number in the range 0 ... 9 occurs. Unlike most of our programs that use `vector`, we allocate `vals` at its desired size. By doing so, we start out with each element initialized to 0.

Inside the `for` loop, we call `lround(n(e))` to round the value returned by `n(e)` to the nearest integer. Having obtained the integer that corresponds to our floating-point random number, we use that number to index our vector of counters. Because `n(e)` can produce a number outside the range 0 to 9, we check that the number we got is in range before using it to index `vals`. If the number is in range, we increment the associated counter.

When the loop completes, we print the contents of `vals`, which will generate output such as

```
0: ***
1: *****
2: *****
3: *****
4: *****
5: *****
6: *****
7: *****
8: *
```

Here we print a `string` with as many asterisks as the count of the times the current value was returned by our random-number generator. Note that this figure

is not perfectly symmetrical. If it were, that symmetry should give us reason to suspect the quality of our random-number generator.

### The `bernoulli_distribution` Class

We noted that there was one distribution that does not take a template parameter. That distribution is the `bernoulli_distribution`, which is an ordinary class, not a template. This distribution always returns a `bool` value. It returns `true` with a given probability. By default that probability is .5.

As an example of this kind of distribution, we might have a program that plays a game with a user. To play the game, one of the players—either the user or the program—has to go first. We could use a `uniform_int_distribution` object with a range of 0 to 1 to select the first player. Alternatively, we can use a Bernoulli distribution to make this choice. Assuming that we have a function named `play` that plays the game, we might have a loop such as the following to interact with the user:

```
string resp;
default_random_engine e; // e has state, so it must be outside the loop!
bernoulli_distribution b; // 50/50 odds by default
do {
    bool first = b(e); // if true, the program will go first
    cout << (first ? "We go first"
              : "You get to go first") << endl;
    // play the game passing the indicator of who goes first
    cout << ((play(first)) ? "sorry, you lost"
              : "congrats, you won") << endl;
    cout << "play again? Enter 'yes' or 'no'" << endl;
} while (cin >> resp && resp[0] == 'y');
```

We use a `do while` (§ 5.4.4, p. 189) to repeatedly prompt the user to play.



Because engines return the same sequence of numbers (§ 17.4.1, p. 747), it is essential that we declare engines outside of loops. Otherwise, we'd create a new engine on each iteration and generate the same values on each iteration. Similarly, distributions may retain state and should also be defined outside loops.

One reason to use a `bernoulli_distribution` in this program is that doing so lets us give the program a better chance of going first:

```
bernoulli_distribution b(.55); // give the house a slight edge
```

If we use this definition for `b`, then the program has 55/45 odds of going first.

## 17.5 The IO Library Revisited

In Chapter 8 we introduced the basic architecture and most commonly used parts of the IO library. In this section we'll look at three of the more specialized features that the IO library supports: format control, unformatted IO, and random access.

## EXERCISES SECTION 17.4.2

**Exercise 17.31:** What would happen if we defined `b` and `e` inside the `do` loop of the game-playing program from this section?

**Exercise 17.32:** What would happen if we defined `resp` inside the loop?

**Exercise 17.33:** Write a version of the word transformation program from § 11.3.6 (p. 440) that allows multiple transformations for a given word and randomly selects which transformation to apply.

### 17.5.1 Formatted Input and Output

In addition to its condition state (§ 8.1.2, p. 312), each `iostream` object also maintains a format state that controls the details of how IO is formatted. The format state controls aspects of formatting such as the notational base for integral values, the precision of floating-point values, the width of an output element, and so on.

The library defines a set of **manipulators** (§ 1.2, p. 7), listed in Tables 17.17 (p. 757) and 17.18 (p. 760), that modify the format state of a stream. A manipulator is a function or object that affects the state of a stream and can be used as an operand to an input or output operator. Like the input and output operators, a manipulator returns the stream object to which it is applied, so we can combine manipulators and data in a single statement.

Our programs have already used one manipulator, `endl`, which we “write” to an output stream as if it were a value. But `endl` isn’t an ordinary value; instead, it performs an operation: It writes a newline and flushes the buffer.

#### Many Manipulators Change the Format State

Manipulators are used for two broad categories of output control: controlling the presentation of numeric values and controlling the amount and placement of padding. Most of the manipulators that change the format state provide set/unset pairs; one manipulator sets the format state to a new value and the other unsets it, restoring the normal default formatting.



Manipulators that change the format state of the stream usually leave the format state changed for all subsequent IO.

The fact that a manipulator makes a persistent change to the format state can be useful when we have a set of IO operations that want to use the same formatting. Indeed, some programs take advantage of this aspect of manipulators to reset the behavior of one or more formatting rules for all its input or output. In such cases, the fact that a manipulator changes the stream is a desirable property.

However, many programs (and, more importantly, programmers) expect the state of the stream to match the normal library defaults. In these cases, leaving the state of the stream in a nonstandard state can lead to errors. As a result, it is usually best to undo whatever state changes are made as soon as those changes are no longer needed.

## Controlling the Format of Boolean Values

One example of a manipulator that changes the formatting state of its object is the `boolalpha` manipulator. By default, `bool` values print as 1 or 0. A `true` value is written as the integer 1 and a `false` value as 0. We can override this formatting by applying the `boolalpha` manipulator to the stream:

```
cout << "default bool values: " << true << " " << false
<< "\nalpha bool values: " << boolalpha
<< true << " " << false << endl;
```

When executed, this program generates the following:

```
default bool values: 1 0
alpha bool values: true false
```

Once we “write” `boolalpha` on `cout`, we’ve changed how `cout` will print `bool` values from this point on. Subsequent operations that print `bools` will print them as either `true` or `false`.

To undo the format state change to `cout`, we apply `noboolalpha`:

```
bool bool_val = get_status();
cout << boolalpha      // sets the internal state of cout
<< bool_val
<< noboolalpha; // resets the internal state to default formatting
```

Here we change the format of `bool` values only to print the value of `bool_val`. Once that value is printed, we immediately reset the stream back to its initial state.

## Specifying the Base for Integral Values

By default, integral values are written and read in decimal notation. We can change the notational base to octal or hexadecimal or back to decimal by using the manipulators `hex`, `oct`, and `dec`:

```
cout << "default: " << 20 << " " << 1024 << endl;
cout << "octal: " << oct << 20 << " " << 1024 << endl;
cout << "hex: " << hex << 20 << " " << 1024 << endl;
cout << "decimal: " << dec << 20 << " " << 1024 << endl;
```

When compiled and executed, this program generates the following output:

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

Notice that like `boolalpha`, these manipulators change the format state. They affect the immediately following output and all subsequent integral output until the format is reset by invoking another manipulator.



The `hex`, `oct`, and `dec` manipulators affect only integral operands; the representation of floating-point values is unaffected.

## Indicating Base on the Output

By default, when we print numbers, there is no visual cue as to what notational base was used. Is 20, for example, really 20, or an octal representation of 16? When we print numbers in decimal mode, the number is printed as we expect. If we need to print octal or hexadecimal values, it is likely that we should also use the `showbase` manipulator. The `showbase` manipulator causes the output stream to use the same conventions as used for specifying the base of an integral constant:

- A leading `0x` indicates hexadecimal.
- A leading `0` indicates octal.
- The absence of either indicates decimal.

Here we've revised the previous program to use `showbase`:

```
cout << showbase;      // show the base when printing integral values
cout << "default: " << 20 << " " << 1024 << endl;
cout << "in octal: " << oct << 20 << " " << 1024 << endl;
cout << "in hex: " << hex << 20 << " " << 1024 << endl;
cout << "in decimal: " << dec << 20 << " " << 1024 << endl;
cout << noshowbase; // reset the state of the stream
```

The revised output makes it clear what the underlying value really is:

```
default: 20 1024
in octal: 024 02000
in hex: 0x14 0x400
in decimal: 20 1024
```

The `noshowbase` manipulator resets `cout` so that it no longer displays the notational base of integral values.

By default, hexadecimal values are printed in lowercase with a lowercase `x`. We can display the `X` and the hex digits `a-f` as uppercase by applying the `uppercase` manipulator:

```
cout << uppercase << showbase << hex
    << "printed in hexadecimal: " << 20 << " " << 1024
    << nouppercase << noshowbase << dec << endl;
```

This statement generates the following output:

```
printed in hexadecimal: 0X14 0X400
```

We apply the `nouppercase`, `noshowbase`, and `dec` manipulators to return the stream to its original state.

## Controlling the Format of Floating-Point Values

We can control three aspects of floating-point output:

- How many digits of precision are printed

- Whether the number is printed in hexadecimal, fixed decimal, or scientific notation
- Whether a decimal point is printed for floating-point values that are whole numbers

By default, floating-point values are printed using six digits of precision; the decimal point is omitted if the value has no fractional part; and they are printed in either fixed decimal or scientific notation depending on the value of the number. The library chooses a format that enhances readability of the number. Very large and very small values are printed using scientific notation. Other values are printed in fixed decimal.

## Specifying How Much Precision to Print

By default, precision controls the total number of digits that are printed. When printed, floating-point values are rounded, not truncated, to the current precision. Thus, if the current precision is four, then 3.14159 becomes 3.142; if the precision is three, then it is printed as 3.14.

We can change the precision by calling the `precision` member of an IO object or by using the `setprecision` manipulator. The `precision` member is overloaded (§ 6.4, p. 230). One version takes an `int` value and sets the precision to that new value. It returns the *previous* precision value. The other version takes no arguments and returns the current precision value. The `setprecision` manipulator takes an argument, which it uses to set the precision.



The `setprecision` manipulators and other manipulators that take arguments are defined in the `iomanip` header.

The following program illustrates the different ways we can control the precision used to print floating-point values:

```
// cout.precision reports the current precision value
cout << "Precision: " << cout.precision()
      << ", Value: " << sqrt(2.0) << endl;
// cout.precision(12) asks that 12 digits of precision be printed
cout.precision(12);
cout << "Precision: " << cout.precision()
      << ", Value: " << sqrt(2.0) << endl;
// alternative way to set precision using the setprecision manipulator
cout << setprecision(3);
cout << "Precision: " << cout.precision()
      << ", Value: " << sqrt(2.0) << endl;
```

When compiled and executed, the program generates the following output:

```
Precision: 6, Value: 1.41421
Precision: 12, Value: 1.41421356237
Precision: 3, Value: 1.41
```

**Table 17.17: Manipulators Defined in `iostream`**

|                            |                                                                 |
|----------------------------|-----------------------------------------------------------------|
| <code>boolalpha</code>     | Display true and false as strings                               |
| * <code>noboolalpha</code> | Display true and false as 0, 1                                  |
| <code>showbase</code>      | Generate prefix indicating the numeric base of integral values  |
| * <code>noshowbase</code>  | Do not generate notational base prefix                          |
| <code>showpoint</code>     | Always display a decimal point for floating-point values        |
| * <code>noshowpoint</code> | Display a decimal point only if the value has a fractional part |
| <code>showpos</code>       | Display + in nonnegative numbers                                |
| * <code>noshowpos</code>   | Do not display + in nonnegative numbers                         |
| <code>uppercase</code>     | Print 0x in hexadecimal, E in scientific                        |
| * <code>nouppercase</code> | Print 0x in hexadecimal, e in scientific                        |
| <code>dec</code>           | Display integral values in decimal numeric base                 |
| <code>hex</code>           | Display integral values in hexadecimal numeric base             |
| <code>oct</code>           | Display integral values in octal numeric base                   |
| <code>left</code>          | Add fill characters to the right of the value                   |
| <code>right</code>         | Add fill characters to the left of the value                    |
| <code>internal</code>      | Add fill characters between the sign and the value              |
| <code>fixed</code>         | Display floating-point values in decimal notation               |
| <code>scientific</code>    | Display floating-point values in scientific notation            |
| <code>hexfloat</code>      | Display floating-point values in hex (new to C++ 11)            |
| <code>defaultfloat</code>  | Reset the floating-point format to decimal (new to C++ 11)      |
| <code>unitbuf</code>       | Flush buffers after every output operation                      |
| * <code>nounitbuf</code>   | Restore normal buffer flushing                                  |
| * <code>skipws</code>      | Skip whitespace with input operators                            |
| <code>noskipws</code>      | Do not skip whitespace with input operators                     |
| <code>flush</code>         | Flush the <code>ostream</code> buffer                           |
| <code>ends</code>          | Insert null, then flush the <code>ostream</code> buffer         |
| <code>endl</code>          | Insert newline, then flush the <code>ostream</code> buffer      |

\* indicates the default stream state

This program calls the library `sqrt` function, which is found in the `cmath` header. The `sqrt` function is overloaded and can be called on either a `float`, `double`, or `long double` argument. It returns the square root of its argument.

## Specifying the Notation of Floating-Point Numbers



Unless you need to control the presentation of a floating-point number (e.g., to print data in columns or to print data that represents money or a percentage), it is usually best to let the library choose the notation.

We can force a stream to use scientific, fixed, or hexadecimal notation by using the appropriate manipulator. The `scientific` manipulator changes the stream to use scientific notation. The `fixed` manipulator changes the stream to use fixed decimal.

Under the new library, we can also force floating-point values to use hexadecimal format by using `hexfloat`. The new library provides another manipulator, named `defaultfloat`. This manipulator returns the stream to its default state in

C++  
11

which it chooses a notation based on the value being printed.

These manipulators also change the default meaning of the precision for the stream. After executing `scientific`, `fixed`, or `hexfloat`, the precision value controls the number of digits after the decimal point. By default, precision specifies the total number of digits—both before and after the decimal point. Using `fixed` or `scientific` lets us print numbers lined up in columns, with the decimal point in a fixed position relative to the fractional part being printed:

```
cout << "default format: " << 100 * sqrt(2.0) << '\n'
<< "scientific: " << scientific << 100 * sqrt(2.0) << '\n'
<< "fixed decimal: " << fixed << 100 * sqrt(2.0) << '\n'
<< "hexadecimal: " << hexfloat << 100 * sqrt(2.0) << '\n'
<< "use defaults: " << defaultfloat << 100 * sqrt(2.0)
<< "\n\n";
```

produces the following output:

```
default format: 141.421
scientific: 1.414214e+002
fixed decimal: 141.421356
hexadecimal: 0x1.1ad7bcf+7
use defaults: 141.421
```

By default, the hexadecimal digits and the `e` used in scientific notation are printed in lowercase. We can use the uppercase manipulator to show those values in uppercase.

## Printing the Decimal Point

By default, when the fractional part of a floating-point value is 0, the decimal point is not displayed. The `showpoint` manipulator forces the decimal point to be printed:

```
cout << 10.0 << endl;           // prints 10
cout << showpoint << 10.0      // prints 10.0000
<< noshowpoint << endl;       // revert to default format for the decimal point
```

The `noshowpoint` manipulator reinstates the default behavior. The next output expression will have the default behavior, which is to suppress the decimal point if the floating-point value has a 0 fractional part.

## Padding the Output

When we print data in columns, we often need fairly fine control over how the data are formatted. The library provides several manipulators to help us accomplish the control we might need:

- `setw` to specify the minimum space for the *next* numeric or string value.
- `left` to left-justify the output.
- `right` to right-justify the output. Output is right-justified by default.

- `internal` controls placement of the sign on negative values. `internal` left-justifies the sign and right-justifies the value, padding any intervening space with blanks.
- `setfill` lets us specify an alternative character to use to pad the output. By default, the value is a space.



`setw`, like `endl`, does not change the internal state of the output stream. It determines the size of only the *next* output.

The following program illustrates these manipulators:

```
int i = -16;
double d = 3.14159;

// pad the first column to use a minimum of 12 positions in the output
cout << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad the first column and left-justify all columns
cout << left
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << right;           // restore normal justification

// pad the first column and right-justify all columns
cout << right
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad the first column but put the padding internal to the field
cout << internal
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n';

// pad the first column, using # as the pad character
cout << setfill('#')
    << "i: " << setw(12) << i << "next col" << '\n'
    << "d: " << setw(12) << d << "next col" << '\n'
    << setfill(' '); // restore the normal pad character
```

When executed, this program generates

```
i:      -16next col
d:      3.14159next col
i: -16      next col
d: 3.14159      next col
i:      -16next col
d:      3.14159next col
i: -      16next col
d:      3.14159next col
i: -#####16next col
d: #####3.14159next col
```

**Table 17.18: Manipulators Defined in `iomanip`**

|                              |                                     |
|------------------------------|-------------------------------------|
| <code>setfill(ch)</code>     | Fill whitespace with ch             |
| <code>setprecision(n)</code> | Set floating-point precision to n   |
| <code>setw(w)</code>         | Read or write value to w characters |
| <code>setbase(b)</code>      | Output integers in base b           |

## Controlling Input Formatting

By default, the input operators ignore whitespace (blank, tab, newline, formfeed, and carriage return). The following loop

```
char ch;
while (cin >> ch)
    cout << ch;
```

given the input sequence

```
a b      c
d
```

executes four times to read the characters a through d, skipping the intervening blanks, possible tabs, and newline characters. The output from this program is

```
abcd
```

The `noskipws` manipulator causes the input operator to read, rather than skip, whitespace. To return to the default behavior, we apply the `skipws` manipulator:

```
cin >> noskipws; // set cin so that it reads whitespace
while (cin >> ch)
    cout << ch;
cin >> skipws; // reset cin to the default state so that it discards whitespace
```

Given the same input as before, this loop makes seven iterations, reading whitespace as well as the characters in the input. This loop generates

```
a b      c
d
```

### EXERCISES SECTION 17.5.1

**Exercise 17.34:** Write a program that illustrates the use of each manipulator in Tables 17.17 (p. 757) and 17.18.

**Exercise 17.35:** Write a version of the program from page 758, that printed the square root of 2 but this time print hexadecimal digits in uppercase.

**Exercise 17.36:** Modify the program from the previous exercise to print the various floating-point values so that they line up in a column.

## 17.5.2 Unformatted Input/Output Operations

So far, our programs have used only **formatted IO** operations. The input and output operators (`<<` and `>>`) format the data they read or write according to the type being handled. The input operators ignore whitespace; the output operators apply padding, precision, and so on.

The library also provides a set of low-level operations that support **unformatted IO**. These operations let us deal with a stream as a sequence of uninterpreted bytes.

### Single-Byte Operations

Several of the unformatted operations deal with a stream one byte at a time. These operations, which are described in Table 17.19, read rather than ignore whitespace. For example, we can use the unformatted IO operations `get` and `put` to read and write the characters one at a time:

```
char ch;
while (cin.get(ch))
    cout.put(ch);
```

This program preserves the whitespace in the input. Its output is identical to the input. It executes the same way as the previous program that used `noskipws`.

Table 17.19: Single-Byte Low-Level IO Operations

|                             |                                                                                                                          |
|-----------------------------|--------------------------------------------------------------------------------------------------------------------------|
| <code>is.get(ch)</code>     | Put the next byte from the <code>istream</code> <code>is</code> in character <code>ch</code> . Returns <code>is</code> . |
| <code>os.put(ch)</code>     | Put the character <code>ch</code> onto the <code>ostream</code> <code>os</code> . Returns <code>os</code> .              |
| <code>is.get()</code>       | Returns next byte from <code>is</code> as an <code>int</code> .                                                          |
| <code>is.putback(ch)</code> | Put the character <code>ch</code> back on <code>is</code> ; returns <code>is</code> .                                    |
| <code>is.unget()</code>     | Move <code>is</code> back one byte; returns <code>is</code> .                                                            |
| <code>is.peek()</code>      | Return the next byte as an <code>int</code> but doesn't remove it.                                                       |

### Putting Back onto an Input Stream

Sometimes we need to read a character in order to know that we aren't ready for it. In such cases, we'd like to put the character back onto the stream. The library gives us three ways to do so, each of which has subtle differences from the others:

- `peek` returns a copy of the next character on the input stream but does not change the stream. The value returned by `peek` stays on the stream.
- `unget` backs up the input stream so that whatever value was last returned is still on the stream. We can call `unget` even if we do not know what value was last taken from the stream.
- `putback` is a more specialized version of `unget`: It returns the last value read from the stream but takes an argument that must be the same as the one that was last read.

In general, we are guaranteed to be able to put back at most one value before the next read. That is, we are not guaranteed to be able to call `putback` or `unget` successively without an intervening read operation.

### **int Return Values from Input Operations**

The `peek` function and the version of `get` that takes no argument return a character from the input stream as an `int`. This fact can be surprising; it might seem more natural to have these functions return a `char`.

The reason that these functions return an `int` is to allow them to return an end-of-file marker. A given character set is allowed to use every value in the `char` range to represent an actual character. Thus, there is no extra value in that range to use to represent end-of-file.

The functions that return `int` convert the character they return to `unsigned char` and then promote that value to `int`. As a result, even if the character set has characters that map to negative values, the `int` returned from these operations will be a positive value (§ 2.1.2, p. 35). The library uses a negative value to represent end-of-file, which is thus guaranteed to be distinct from any legitimate character value. Rather than requiring us to know the actual value returned, the `cstdio` header defines a `const` named `EOF` that we can use to test if the value returned from `get` is end-of-file. It is essential that we use an `int` to hold the return from these functions:

```
int ch;      // use an int, not a char to hold the return from get ()
// loop to read and write all the data in the input
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

This program operates identically to the one on page 761, the only difference being the version of `get` that is used to read the input.

### **Multi-Byte Operations**

Some unformatted IO operations deal with chunks of data at a time. These operations can be important if speed is an issue, but like other low-level operations, they are error-prone. In particular, these operations require us to allocate and manage the character arrays (§ 12.2, p. 476) used to store and retrieve data. The multi-byte operations are listed in Table 17.20.

The `get` and `getline` functions take the same parameters, and their actions are similar but not identical. In each case, `sink` is a `char` array into which the data are placed. The functions read until one of the following conditions occurs:

- `size - 1` characters are read
- End-of-file is encountered
- The delimiter character is encountered

The difference between these functions is the treatment of the delimiter: `get` leaves the delimiter as the next character of the `istream`, whereas `getline` reads and discards the delimiter. In either case, the delimiter is *not* stored in `sink`.

**Table 17.20: Multi-Byte Low-Level IO Operations**

|                                            |                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>is.get(sink, size, delim)</code>     | Reads up to <code>size</code> bytes from <code>is</code> and stores them in the character array beginning at the address pointed to by <code>sink</code> . Reads until encountering the <code>delim</code> character or until it has read <code>size</code> bytes or encounters end-of-file. If <code>delim</code> is present, it is left on the input stream and not read into <code>sink</code> . |
| <code>is.getline(sink, size, delim)</code> | Same behavior as the three-argument version of <code>get</code> but reads and discards <code>delim</code> .                                                                                                                                                                                                                                                                                         |
| <code>is.read(sink, size)</code>           | Reads up to <code>size</code> bytes into the character array <code>sink</code> . Returns <code>is</code> .                                                                                                                                                                                                                                                                                          |
| <code>is.gcount()</code>                   | Returns number of bytes read from the stream <code>is</code> by the last call to an unformatted read operation.                                                                                                                                                                                                                                                                                     |
| <code>os.write(source, size)</code>        | Writes <code>size</code> bytes from the character array <code>source</code> to <code>os</code> . Returns <code>os</code> .                                                                                                                                                                                                                                                                          |
| <code>is.ignore(size, delim)</code>        | Reads and ignores at most <code>size</code> characters up to and including <code>delim</code> . Unlike the other unformatted functions, <code>ignore</code> has default arguments: <code>size</code> defaults to 1 and <code>delim</code> to end-of-file.                                                                                                                                           |



It is a common error to intend to remove the delimiter from the stream but to forget to do so.

### Determining How Many Characters Were Read

Several of the read operations read an unknown number of bytes from the input. We can call `gcount` to determine how many characters the last unformatted input operation read. It is essential to call `gcount` before any intervening unformatted input operation. In particular, the single-character operations that put characters back on the stream are also unformatted input operations. If `peek`, `unget`, or `putback` are called before calling `gcount`, then the return value will be 0.

#### 17.5.3 Random Access to a Stream

The various stream types generally support random access to the data in their associated stream. We can reposition the stream so that it skips around, reading first the last line, then the first, and so on. The library provides a pair of functions to *seek* to a given location and to *tell* the current location in the associated stream.



Random IO is an inherently system-dependent. To understand how to use these features, you must consult your system's documentation.

Although these `seek` and `tell` functions are defined for all the stream types, whether they do anything useful depends on the device to which the stream is bound. On most systems, the streams bound to `cin`, `cout`, `cerr`, and `clog` do

### **CAUTION: LOW-LEVEL ROUTINES ARE ERROR-PRONE**

In general, we advocate using the higher-level abstractions provided by the library. The IO operations that return `int` are a good example of why.

It is a common programming error to assign the return, from `get` or `peek` to a `char` rather than an `int`. Doing so is an error, but an error the compiler will not detect. Instead, what happens depends on the machine and on the input data. For example, on a machine in which `chars` are implemented as `unsigned chars`, this loop will run forever:

```
char ch;      // using a char here invites disaster!
// the return from cin.get is converted to char and then compared to an int
while ((ch = cin.get()) != EOF)
    cout.put(ch);
```

The problem is that when `get` returns `EOF`, that value will be converted to an `unsigned char` value. That converted value is no longer equal to the `int` value of `EOF`, and the loop will continue forever. Such errors are likely to be caught in testing.

On machines for which `chars` are implemented as `signed chars`, we can't say with confidence what the behavior of the loop might be. What happens when an out-of-bounds value is assigned to a `signed` value is up to the compiler. On many machines, this loop will appear to work, unless a character in the input matches the `EOF` value. Although such characters are unlikely in ordinary data, presumably low-level IO is necessary only when we read binary values that do not map directly to ordinary characters and numeric values. For example, on our machine, if the input contains a character whose value is '`\377`', then the loop terminates prematurely. '`\377`' is the value on our machine to which `-1` converts when used as a `signed char`. If the input has this value, then it will be treated as the (premature) end-of-file indicator.

Such bugs do not happen when we read and write typed values. If you can use the more type-safe, higher-level operations supported by the library, do so.

### **EXERCISES SECTION 17.5.2**

**Exercise 17.37:** Use the unformatted version of `getline` to read a file a line at a time. Test your program by giving it a file that contains empty lines as well as lines that are longer than the character array that you pass to `getline`.

**Exercise 17.38:** Extend your program from the previous exercise to print each word you read onto its own line.

*not* support random access—after all, what would it mean to jump back ten places when we're writing directly to `cout`? We can call the `seek` and `tell` functions, but these functions will fail at run time, leaving the stream in an invalid state.



Because the `istream` and `ostream` types usually do not support random access, the remainder of this section should be considered as applicable to only the `fstream` and `sstream` types.

## Seek and Tell Functions

To support random access, the IO types maintain a marker that determines where the next read or write will happen. They also provide two functions: One repositions the marker by *seeking* to a given position; the second *tells* us the current position of the marker. The library actually defines two pairs of *seek* and *tell* functions, which are described in Table 17.21. One pair is used by input streams, the other by output streams. The input and output versions are distinguished by a suffix that is either a *g* or a *p*. The *g* versions indicate that we are “getting” (reading) data, and the *p* functions indicate that we are “putting” (writing) data.

**Table 17.21: Seek and Tell Functions**

|                               |                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>tellg()</code>          | Return the current position of the marker in an input stream ( <code>tellg</code> ) or an output stream ( <code>tellp</code> ).                                                                                                                                                                                                                                                                                                         |
| <code>seekg(pos)</code>       | Reposition the marker in an input or output stream to the given absolute address in the stream. <code>pos</code> is usually a value returned by a previous call to the corresponding <code>tellg</code> or <code>tellp</code> function.                                                                                                                                                                                                 |
| <code>seekp(off, from)</code> | Reposition the marker for an input or output stream integral number <code>off</code> characters ahead or behind <code>from</code> . <code>from</code> can be one of <ul style="list-style-type: none"><li>• <code>beg</code>, seek relative to the beginning of the stream</li><li>• <code>cur</code>, seek relative to the current position of the stream</li><li>• <code>end</code>, seek relative to the end of the stream</li></ul> |
| <code>seekg(off, from)</code> |                                                                                                                                                                                                                                                                                                                                                                                                                                         |

Logically enough, we can use only the *g* versions on an *istream* and on the types *ifstream* and *istringstream* that inherit from *istream* (§ 8.1, p. 311). We can use only the *p* versions on an *ostream* and on the types that inherit from it, *ofstream* and *ostringstream*. An *iostream*, *fstream*, or *stringstream* can both read and write the associated stream; we can use either the *g* or *p* versions on objects of these types.

## There Is Only One Marker

The fact that the library distinguishes between the “putting” and “getting” versions of the seek and tell functions can be misleading. Even though the library makes this distinction, it maintains only a single marker in a stream—there is *not* a distinct read marker and write marker.

When we’re dealing with an input-only or output-only stream, the distinction isn’t even apparent. We can use only the *g* or only the *p* versions on such streams. If we attempt to call `tellp` on an *ifstream*, the compiler will complain. Similarly, it will not let us call `seekg` on an *ostringstream*.

The *fstream* and *stringstream* types can read and write the same stream. In these types there is a single buffer that holds data to be read and written and a single marker denoting the current position in the buffer. The library maps both the *g* and *p* positions to this single marker.



Because there is only a single marker, we *must* do a seek to reposition the marker whenever we switch between reading and writing.

## Repositioning the Marker

There are two versions of the seek functions: One moves to an “absolute” address within the file; the other moves to a byte offset from a given position:

```
// set the marker to a fixed position
seekg(new_position);      // set the read marker to the given pos_type location
seekp(new_position);      // set the write marker to the given pos_type location

// offset some distance ahead of or behind the given starting point
seekg(offset, from);     // set the read marker offset distance from from
seekp(offset, from);     // offset has type off_type
```

The possible values for `from` are listed in Table 17.21 (on the previous page).

The arguments, `new_position` and `offset`, have machine-dependent types named `pos_type` and `off_type`, respectively. These types are defined in both `istream` and `ostream`. `pos_type` represents a file position and `off_type` represents an offset from that position. A value of type `off_type` can be positive or negative; we can seek forward or backward in the file.

## Accessing the Marker

The `tellg` or `tellp` functions return a `pos_type` value denoting the current position of the stream. The tell functions are usually used to remember a location so that we can subsequently seek back to it:

```
// remember the current write position in mark
ostringstream writeStr;    // output stringstream
ostringstream::pos_type mark = writeStr.tellp();
// ...
if (cancelEntry)
    // return to the remembered position
    writeStr.seekp(mark);
```

## Reading and Writing to the Same File

Let’s look at a programming example. Assume we are given a file to read. We are to write a newline at the end of the file that contains the relative position at which each line begins. For example, given the following file,

```
abcd
efg
hi
j
```

the program should produce the following modified file:

```
abcd
efg
hi
j
5 9 12 14
```

Note that our program need not write the offset for the first line—it always occurs at position 0. Also note that the offset counts must include the invisible newline character that ends each line. Finally, note that the last number in the output is the offset for the line on which our output begins. By including this offset in our output, we can distinguish our output from the file’s original contents. We can read the last number in the resulting file and seek to the corresponding offset to get to the beginning of our output.

Our program will read the file a line at a time. For each line, we’ll increment a counter, adding the size of the line we just read. That counter is the offset at which the next line starts:

```
int main()
{
    // open for input and output and preposition file pointers to end-of-file
    // file mode argument see § 8.4 (p. 319)
    fstream inout("copyOut",
                 fstream::ate | fstream::in | fstream::out);
    if (!inout) {
        cerr << "Unable to open file!" << endl;
        return EXIT_FAILURE; // EXIT_FAILURE see § 6.3.2 (p. 227)
    }
    // inout is opened in ate mode, so it starts out positioned at the end
    auto end_mark = inout.tellg(); // remember original end-of-file position
    inout.seekg(0, fstream::beg); // reposition to the start of the file
    size_t cnt = 0; // accumulator for the byte count
    string line; // hold each line of input
    // while we haven't hit an error and are still reading the original data
    while (inout && inout.tellg() != end_mark
           && getline(inout, line)) { // and can get another line of input
        cnt += line.size() + 1; // add 1 to account for the newline
        auto mark = inout.tellg(); // remember the read position
        inout.seekp(0, fstream::end); // set the write marker to the end
        inout << cnt; // write the accumulated length
        // print a separator if this is not the last line
        if (mark != end_mark) inout << " ";
        inout.seekg(mark); // restore the read position
    }
    inout.seekp(0, fstream::end); // seek to the end
    inout << "\n"; // write a newline at end-of-file
    return 0;
}
```

Our program opens its `fstream` using the `in`, `out`, and `ate` modes (§ 8.4, p. 319). The first two modes indicate that we intend to read and write the same file. Speci-

fying `ate` positions the read and write markers at the end of the file. As usual, we check that the open succeeded, and exit if it did not (§ 6.3.2, p. 227).

Because our program writes to its input file, we can't use end-of-file to signal when it's time to stop reading. Instead, our loop must end when it reaches the point at which the original input ended. As a result, we must first remember the original end-of-file position. Because we opened the file in `ate` mode, `inOut` is already positioned at the end. We store the current (i.e., the original end) position in `end_mark`. Having remembered the end position, we reposition the read marker at the beginning of the file by seeking to the position 0 bytes from the beginning of the file.

The `while` loop has a three-part condition: We first check that the stream is valid; if so, we check whether we've exhausted our original input by comparing the current read position (returned by `tellg`) with the position we remembered in `end_mark`. Finally, assuming that both tests succeeded, we call `getline` to read the next line of input. If `getline` succeeds, we perform the body of the loop.

The loop body starts by remembering the current position in `mark`. We save that position in order to return to it after writing the next relative offset. The call to `seekp` repositions the write marker to the end of the file. We write the counter value and then `seekg` back to the position we remembered in `mark`. Having restored the marker, we're ready to repeat the condition in the `while`.

Each iteration of the loop writes the offset of the next line. Therefore, the last iteration of the loop takes care of writing the offset of the last line. However, we still need to write a newline at the end of the file. As with the other writes, we call `seekp` to position the file at the end before writing the newline.

### EXERCISES SECTION 17.5.3

**Exercise 17.39:** Write your own version of the `seek` program presented in this section.

## CHAPTER SUMMARY

---

This chapter covered additional IO operations and four library types: `tuple`, `bitset`, regular expressions, and random numbers.

A `tuple` is a template that allows us to bundle together members of disparate types into a single object. Each `tuple` contains a specified number of members, but the library imposes no limit on the number of members we can define for a given `tuple` type.

A `bitset` lets us define collections of bits of a specified size. The size of a `bitset` is not constrained to match any of the integral types, and can even exceed them. In addition to supporting the normal bitwise operators (§ 4.8, p. 152), `bitset` defines a number of named operations that let us manipulate the state of particular bits in the `bitset`.

The regular-expression library provides a collection of classes and functions: The `regex` class manages regular expressions written in one of several common regular-expression languages. The `match` classes hold information about a specific match. These classes are used by the `regex_search` and `regex_match` functions. These functions take a `regex` object and a character sequence and detect whether the regular expression in that `regex` matches the given character sequence. The `regex` iterator types are iterator adaptors that use `regex_search` to iterate through an input sequence and return each matching subsequence. There is also a `regex_replace` function that lets us replace the matched part of a given input sequence with a specified alternative.

The random-number library is a collection of random-number engines and distribution classes. A random-number engine returns a sequence of uniformly distributed integral values. The library defines several engines that have different performance characteristics. The `default_random_engine` is defined as the engine that should be suitable for most casual uses. The library also defines 20 distribution types. These distribution types use an engine to deliver random numbers of a specified type in a given range that are distributed according to a specified probability distribution.

## DEFINED TERMS

---

**bitset** Standard library class that holds a collection of bits of a size that is known at compile time, and provides operations to test and set the bits in the collection.

**cmatch** Container of `csub_match` objects that provides information about the match to a `regex` on `const char*` input sequences. The first element in the container describes the overall match results. The subsequent elements describe the results for the subexpressions.

**cregex\_iterator** Like `sregex_iterator`

except that it iterates over an array of `char`.

**csub\_match** Type that holds the results of a regular expression match to a `const char*`. Can represent the entire match or a subexpression.

**default random engine** Type alias for the random number engine intended for normal use.

**formatted IO** IO operations that use the types of the objects being read or written to define the actions of the operations. For-

matted input operations perform whatever transformations are appropriate to the type being read, such as converting ASCII numeric strings to the indicated arithmetic type and (by default) ignoring whitespace. Formatted output routines convert types to printable character representations, pad the output, and may perform other, type-specific transformations.

**get** Template function that returns the specified member for a given tuple. For example, `get<0>(t)` returns the first element from the tuple `t`.

**high-order** Bits in a bitset with the largest indices.

**low-order** Bits in a bitset with the lowest indices.

**manipulator** A function-like object that “manipulates” a stream. Manipulators can be used as the right-hand operand to the overloaded IO operators, `<<` and `>>`. Most manipulators change the internal state of the object. Such manipulators often come in pairs—one to change the state and the other to return the stream to its default state.

**random-number distribution** Standard library type that transforms the output of a random-number engine according to its named distribution. For example, `uniform_int_distribution<T>` generates uniformly distributed integers of type `T`, `normal_distribution<T>` generates normally distributed numbers, and so on.

**random-number engine** Library type that generates random unsigned numbers. Engines are intended to be used only as inputs to random-number distributions.

**random-number generator** Combination of a random-number engine type and a distribution type.

**regex** Class that manages a regular expression.

**regex\_error** Exception type thrown to indicate a syntactic error in a regular expression.

**regex\_match** Function that determines whether the entire input sequence matches the given `regex` object.

**regex\_replace** Function that uses a `regex` object to replace matching subexpressions in an input sequence using a given format.

**regex\_search** Function that uses a `regex` object to find a matching subsequence of a given input sequence.

**regular expression** A way of describing a sequence of characters.

**seed** Value supplied to a random-number engine that causes it to move to a new point in the sequence of number that it generates.

**smatch** Container of `ssub_match` objects that provides information about the match to a `regex` on `string` input sequences. The first element in the container describes the overall match results. The subsequent elements describe the results for the subexpressions.

**sregex\_iterator** Iterator that iterates over a `string` using a given `regex` object to find matches in the given `string`. The constructor positions the iterator on the first match by calling `regex_search`. Incrementing the iterator calls `regex_search` starting just after the current match in the given `string`. Dereferencing the iterator returns an `smatch` object describing the current match.

**ssub\_match** Type that holds results of a regular expression match to a `string`. Can represent the entire match or a subexpression.

**subexpression** Parenthesized component of a regular expression pattern.

**tuple** Template that generates types that hold unnamed members of specified types. There is no fixed limit on the number of members a `tuple` can be defined to have.

**unformatted IO** Operations that treat the stream as an undifferentiated byte stream. Unformatted operations place more of the burden for managing the IO on the user.

# C H A P T E R      18

## T O O L S F O R L A R G E P R O G R A M S

### CONTENTS

---

|                                                         |     |
|---------------------------------------------------------|-----|
| Section 18.1 Exception Handling . . . . .               | 772 |
| Section 18.2 Namespaces . . . . .                       | 785 |
| Section 18.3 Multiple and Virtual Inheritance . . . . . | 802 |
| Chapter Summary . . . . .                               | 816 |
| Defined Terms . . . . .                                 | 816 |

C++ is used on problems small enough to be solved by a single programmer after a few hours' work and on problems requiring enormous systems consisting of tens of millions of lines of code developed and modified by hundreds of programmers over many years. The facilities that we covered in the earlier parts of this book are equally useful across this range of programming problems.

The language includes some features that are most useful on systems that are more complicated than those that a small team can manage. These features—exception handling, namespaces, and multiple inheritance—are the topic of this chapter.

*Large-scale programming* places greater demands on programming languages than do the needs of systems that can be developed by small teams of programmers. Among the needs that distinguish large-scale applications are

- The ability to handle errors across independently developed subsystems
- The ability to use libraries developed more or less independently
- The ability to model more complicated application concepts

This chapter looks at three features in C++ that are aimed at these needs: exception handling, namespaces, and multiple inheritance.

## 18.1 Exception Handling

**Exception handling** allows independently developed parts of a program to communicate about and handle problems that arise at run time. Exceptions let us separate problem detection from problem resolution. One part of the program can detect a problem and can pass the job of resolving that problem to another part of the program. The detecting part need not know anything about the handling part, and vice versa.

In § 5.6 (p. 193) we introduced the basic concepts and mechanics of using exceptions. In this section we'll expand our coverage of these basics. Effective use of exception handling requires understanding what happens when an exception is thrown, what happens when it is caught, and the meaning of the objects that communicate what went wrong.

### 18.1.1 Throwing an Exception

In C++, an exception is **raised** by **throwing** an expression. The type of the thrown expression, together with the current call chain, determines which **handler** will deal with the exception. The selected handler is the one nearest in the call chain that matches the type of the thrown object. The type and contents of that object allow the throwing part of the program to inform the handling part about what went wrong.

When a `throw` is executed, the statement(s) following the `throw` are not executed. Instead, control is transferred from the `throw` to the matching `catch`. That `catch` might be local to the same function or might be in a function that directly or indirectly called the function in which the exception occurred. The fact that control passes from one location to another has two important implications:

- Functions along the call chain may be prematurely exited.
- When a handler is entered, objects created along the call chain will have been destroyed.

Because the statements following a `throw` are not executed, a `throw` is like a `return`: It is usually part of a conditional statement or is the last (or only) statement in a function.

## Stack Unwinding

When an exception is thrown, execution of the current function is suspended and the search for a matching catch clause begins. If the throw appears inside a **try block**, the catch clauses associated with that try are examined. If a matching catch is found, the exception is handled by that catch. Otherwise, if the try was itself nested inside another try, the search continues through the catch clauses of the enclosing tries. If no matching catch is found, the current function is exited, and the search continues in the calling function.

If the call to the function that threw is in a try block, then the catch clauses associated with that try are examined. If a matching catch is found, the exception is handled. Otherwise, if that try was nested, the catch clauses of the enclosing tries are searched. If no catch is found, the calling function is also exited. The search continues in the function that called the just exited one, and so on.

This process, known as **stack unwinding**, continues up the chain of nested function calls until a catch clause for the exception is found, or the main function itself is exited without having found a matching catch.

Assuming a matching catch is found, that catch is entered, and the program continues by executing the code inside that catch. When the catch completes, execution continues at the point immediately after the last catch clause associated with that try block.

If no matching catch is found, the program is exited. Exceptions are intended for events that prevent the program from continuing normally. Therefore, once an exception is raised, it cannot remain unhandled. If no matching catch is found, the program calls the library **terminate** function. As its name implies, terminate stops execution of the program.



An exception that is not caught terminates the program.

## Objects Are Automatically Destroyed during Stack Unwinding

During stack unwinding, blocks in the call chain may be exited prematurely. In general, these blocks will have created local objects. Ordinarily, local objects are destroyed when the block in which they are created is exited. Stack unwinding is no exception. When a block is exited during stack unwinding, the compiler guarantees that objects created in that block are properly destroyed. If a local object is of class type, the destructor for that object is called automatically. As usual, the compiler does no work to destroy objects of built-in type.

If an exception occurs in a constructor, then the object under construction might be only partially constructed. Some of its members might have been initialized, but others might not have been initialized before the exception occurred. Even if the object is only partially constructed, we are guaranteed that the constructed members will be properly destroyed.

Similarly, an exception might occur during initialization of the elements of an array or a library container type. Again, we are guaranteed that the elements (if any) that were constructed before the exception occurred will be destroyed.

## Destructors and Exceptions

The fact that destructors are run—but code inside a function that frees a resource may be bypassed—affects how we structure our programs. As we saw in § 12.1.4 (p. 467), if a block allocates a resource, and an exception occurs before the code that frees that resource, the code to free the resource will not be executed. On the other hand, resources allocated by an object of class type generally will be freed by their destructor. By using classes to control resource allocation, we ensure that resources are properly freed, whether a function ends normally or via an exception.

The fact that destructors are run during stack unwinding affects how we write destructors. During stack unwinding, an exception has been raised but is not yet handled. If a new exception is thrown during stack unwinding and not caught in the function that threw it, `terminate` is called. Because destructors may be invoked during stack unwinding, they should never throw exceptions that the destructor itself does not handle. That is, if a destructor does an operation that might throw, it should wrap that operation in a `try` block and handle it locally to the destructor.

In practice, because destructors free resources, it is unlikely that they will throw exceptions. All of the standard library types guarantee that their destructors will not raise an exception.



During stack unwinding, destructors are run on local objects of class type. Because destructors are run automatically, they should not throw. If, during stack unwinding, a destructor throws an exception that it does not also catch, the program will be terminated.

## The Exception Object

The compiler uses the thrown expression to copy initialize (§ 13.1.1, p. 497) a special object known as the **exception object**. As a result, the expression in a `throw` must have a complete type (§ 7.3.3, p. 278). Moreover, if the expression has class type, that class must have an accessible destructor and an accessible copy or move constructor. If the expression has an array or function type, the expression is converted to its corresponding pointer type.

The exception object resides in space, managed by the compiler, that is guaranteed to be accessible to whatever `catch` is invoked. The exception object is destroyed after the exception is completely handled.

As we've seen, when an exception is thrown, blocks along the call chain are exited until a matching handler is found. When a block is exited, the memory used by the local objects in that block is freed. As a result, it is almost certainly an error to throw a pointer to a local object. It is an error for the same reasons that it is an error to return a pointer to a local object (§ 6.3.2, p. 225) from a function. If the pointer points to an object in a block that is exited before the `catch`, then that local object will have been destroyed before the `catch`.

When we throw an expression, the static, compile-time type (§ 15.2.3, p. 601) of that expression determines the type of the exception object. This point is essential to keep in mind, because many applications throw expressions whose type comes

from an inheritance hierarchy. If a `throw` expression dereferences a pointer to a base-class type, and that pointer points to a derived-type object, then the thrown object is sliced down (§ 15.2.3, p. 603); only the base-class part is thrown.



Throwing a pointer requires that the object to which the pointer points exist wherever the corresponding handler resides.

## EXERCISES SECTION 18.1.1

**Exercise 18.1:** What is the type of the exception object in the following `throws`?

- (a) `range_error r("error");` (b) `exception *p = &r;`  
`throw r;` `throw *p;`

What would happen if the `throw` in (b) were written as `throw p`?

**Exercise 18.2:** Explain what happens if an exception occurs at the indicated point:

```
void exercise(int *b, int *e)
{
    vector<int> v(b, e);
    int *p = new int[v.size()];
    ifstream in("ints");
    // exception occurs here
}
```

**Exercise 18.3:** There are two ways to make the previous code work correctly if an exception is thrown. Describe them and implement them.

## 18.1.2 Catching an Exception

The **exception declaration** in a **catch clause** looks like a function parameter list with exactly one parameter. As in a parameter list, we can omit the name of the catch parameter if the catch has no need to access the thrown expression.

The type of the declaration determines what kinds of exceptions the handler can catch. The type must be a complete type (§ 7.3.3, p. 278). The type can be an lvalue reference but may not be an rvalue reference (§ 13.6.1, p. 532).

When a `catch` is entered, the parameter in its exception declaration is initialized by the exception object. As with function parameters, if the `catch` parameter has a nonreference type, then the parameter in the `catch` is a copy of the exception object; changes made to the parameter inside the `catch` are made to a local copy, not to the exception object itself. If the parameter has a reference type, then like any reference parameter, the `catch` parameter is just another name for the exception object. Changes made to the parameter are made to the exception object.

Also like a function parameter, a `catch` parameter that has a base-class type can be initialized by an exception object that has a type derived from the parameter type. If the `catch` parameter has a nonreference type, then the exception object

will be sliced down (§ 15.2.3, p. 603), just as it would be if such an object were passed to an ordinary function by value. On the other hand, if the parameter is a reference to a base-class type, then the parameter is bound to the exception object in the usual way.

Again, as with a function parameter, the static type of the exception declaration determines the actions that the `catch` may perform. If the `catch` parameter has a base-class type, then the `catch` cannot use any members that are unique to the derived type.



Ordinarily, a `catch` that takes an exception of a type related by inheritance ought to define its parameter as a reference.

## Finding a Matching Handler

During the search for a matching `catch`, the `catch` that is found is not necessarily the one that matches the exception best. Instead, the selected `catch` is the *first* one that matches the exception at all. As a consequence, in a list of `catch` clauses, the most specialized `catch` must appear first.

Because `catch` clauses are matched in the order in which they appear, programs that use exceptions from an inheritance hierarchy must order their `catch` clauses so that handlers for a derived type occur before a `catch` for its base type.

The rules for when an exception matches a `catch` exception declaration are much more restrictive than the rules used for matching arguments with parameter types. Most conversions are not allowed—the types of the exception and the `catch` declaration must match exactly with only a few possible differences:

- Conversions from `nonconst` to `const` are allowed. That is, a `throw` of a `nonconst` object can match a `catch` specified to take a reference to `const`.
- Conversions from derived type to base type are allowed.
- An array is converted to a pointer to the type of the array; a function is converted to the appropriate pointer to function type.

No other conversions are allowed to match a `catch`. In particular, neither the standard arithmetic conversions nor conversions defined for class types are permitted.



Multiple `catch` clauses with types related by inheritance must be ordered from most derived type to least derived.

## Rethrow

Sometimes a single `catch` cannot completely handle an exception. After some corrective actions, a `catch` may decide that the exception must be handled by a function further up the call chain. A `catch` passes its exception out to another `catch` by **rethrowing** the exception. A `rethrow` is a `throw` that is not followed by an expression:

```
throw;
```

An empty throw can appear only in a catch or in a function called (directly or indirectly) from a catch. If an empty throw is encountered when a handler is not active, terminate is called.

A rethrow does not specify an expression; the (current) exception object is passed up the chain.

In general, a catch might change the contents of its parameter. If, after changing its parameter, the catch rethrows the exception, then those changes will be propagated only if the catch's exception declaration is a reference:

```
catch (my_error &eObj) {      // specifier is a reference type
    eObj.status = errCodes::severeErr; // modifies the exception object
    throw; // the status member of the exception object is severeErr
} catch (other_error eObj) { // specifier is a nonreference type
    eObj.status = errCodes::badErr; // modifies the local copy only
    throw; // the status member of the exception object is unchanged
}
```

## The Catch-All Handler

Sometimes we want to catch any exception that might occur, regardless of type. Catching every possible exception can be a problem: Sometimes we don't know what types might be thrown. Even when we do know all the types, it may be tedious to provide a specific catch clause for every possible exception. To catch all exceptions, we use an ellipsis for the exception declaration. Such handlers, sometimes known as **catch-all** handlers, have the form `catch(...)`. A catch-all clause matches any type of exception.

A `catch(...)` is often used in combination with a rethrow expression. The catch does whatever local work can be done and then rethrows the exception:

```
void manip() {
    try {
        // actions that cause an exception to be thrown
    }
    catch (...) {
        // work to partially handle the exception
        throw;
    }
}
```

A `catch(...)` clause can be used by itself or as one of several `catch` clauses.



If a `catch(...)` is used in combination with other `catch` clauses, it must be last. Any `catch` that follows a catch-all can never be matched.

### 18.1.3 Function try Blocks and Constructors

In general, exceptions can occur at any point in the program's execution. In particular, an exception might occur while processing a constructor initializer. Constructor initializers execute before the constructor body is entered. A `catch` inside

## EXERCISES SECTION 18.1.2

**Exercise 18.4:** Looking ahead to the inheritance hierarchy in Figure 18.1 (p. 783), explain what's wrong with the following `try` block. Correct it.

```
try {
    // use of the C++ standard library
} catch(exception) {
    // ...
} catch(const runtime_error &re) {
    // ...
} catch(overflow_error eobj) { /* ... */ }
```

**Exercise 18.5:** Modify the following `main` function to catch any of the exception types shown in Figure 18.1 (p. 783):

```
int main() {
    // use of the C++ standard library
}
```

The handlers should print the error message associated with the exception before calling `abort` (defined in the header `cstdlib`) to terminate `main`.

**Exercise 18.6:** Given the following exception types and `catch` clauses, write a `throw` expression that creates an exception object that can be caught by each `catch` clause:

- (a) class exceptionType { };  
    catch(exceptionType \*pet) { }
- (b) catch(...) { }
- (c) typedef int EXCPTYPE;  
    catch(EXCPTYPE) { }

the constructor body can't handle an exception thrown by a constructor initializer because a `try` block inside the constructor body would not yet be in effect when the exception is thrown.

To handle an exception from a constructor initializer, we must write the constructor as a **function `try` block**. A function `try` block lets us associate a group of `catch` clauses with the initialization phase of a constructor (or the destruction phase of a destructor) as well as with the constructor's (or destructor's) function body. As an example, we might wrap the `Blob` constructors (§ 16.1.2, p. 662) in a function `try` block:

```
template <typename T>
Blob<T>::Blob(std::initializer_list<T> il) try :
    data(std::make_shared<std::vector<T>>(il)) {
    /* empty body */
} catch(const std::bad_alloc &e) { handle_out_of_memory(e); }
```

Notice that the keyword `try` appears before the colon that begins the constructor initializer list and before the curly brace that forms the (in this case empty) constructor function body. The `catch` associated with this `try` can be used to handle

exceptions thrown either from within the member initialization list or from within the constructor body.

It is worth noting that an exception can happen while initializing the constructor's parameters. Such exceptions are *not* part of the function `try` block. The function `try` block handles only exceptions that occur once the constructor begins executing. As with any other function call, if an exception occurs during parameter initialization, that exception is part of the calling expression and is handled in the caller's context.



The only way for a constructor to handle an exception from a constructor initializer is to write the constructor as a function `try` block.

### EXERCISES SECTION 18.1.3

**Exercise 18.7:** Define your `Blob` and `BlobPtr` classes from Chapter 16 to use function `try` blocks for their constructors.

#### 18.1.4 The `noexcept` Exception Specification

It can be helpful both to users and to the compiler to know that a function will not throw any exceptions. Knowing that a function will not throw simplifies the task of writing code that calls that function. Moreover, if the compiler knows that no exceptions will be thrown, it can (sometimes) perform optimizations that must be suppressed if code might throw.

Under the new standard, a function can specify that it does not throw exceptions by providing a **`noexcept` specification**. The keyword `noexcept` following the function parameter list indicates that the function won't throw:

```
void recoup(int) noexcept; // won't throw
void alloc(int);           // might throw
```

C++  
11

These declarations say that `recoup` will not throw any exceptions and that `alloc` might. We say that `recoup` has a **nonthrowing specification**.

The `noexcept` specifier must appear on all of the declarations and the corresponding definition of a function or on none of them. The specifier precedes a trailing return (§ 6.3.3, p. 229). We may also specify `noexcept` on the declaration and definition of a function pointer. It may not appear in a `typedef` or type alias. In a member function the `noexcept` specifier follows any `const` or reference qualifiers, and it precedes `final`, `override`, or `= 0` on a virtual function.

#### Violating the Exception Specification

It is important to understand that the compiler does not check the `noexcept` specification at compile time. In fact, the compiler is not permitted to reject a function with a `noexcept` specifier merely because it contains a `throw` or calls a function that might throw (however, kind compilers will warn about such usages):

```
// this function will compile, even though it clearly violates its exception specification
void f() noexcept           // promises not to throw any exception
{
    throw exception();      // violates the exception specification
}
```

As a result, it is possible that a function that claims it will not throw will in fact throw. If a noexcept function does throw, terminate is called, thereby enforcing the promise not to throw at run time. It is unspecified whether the stack is unwound. As a result, noexcept should be used in two cases: if we are confident that the function won't throw, and/or if we don't know what we'd do to handle the error anyway.

Specifying that a function won't throw effectively promises the *callers* of the nonthrowing function that they will never need to deal with exceptions. Either the function won't throw, or the whole program will terminate; the caller escapes responsibility either way.



The compiler in general cannot, and does not, verify exception specifications at compile time.

## BACKWARD COMPATIBILITY: EXCEPTION SPECIFICATIONS

Earlier versions of C++ had a more elaborate scheme of exception specifications that allowed us to specify the types of exceptions that a function might throw. A function can specify the keyword `throw` followed by a parenthesized list of types that the function might throw. The `throw` specifier appeared in the same place as the `noexcept` specifier does in the current language.

This approach was never widely used and has been deprecated in the current standard. Although these more elaborate specifiers have been deprecated, there is one use of the old scheme that is in widespread use. A function that is designated by `throw()` promises not to throw any exceptions:

```
void recoup(int) noexcept; // recoup doesn't throw
void recoup(int) throw(); // equivalent declaration
```

These declarations of `recoup` both say that `recoup` won't throw.

## Arguments to the `noexcept` Specification

The `noexcept` specifier takes an optional argument that must be convertible to `bool`: If the argument is `true`, then the function won't throw; if the argument is `false`, then the function might throw:

```
void recoup(int) noexcept(true); // recoup won't throw
void alloc(int) noexcept(false); // alloc can throw
```

## The `noexcept` Operator

C++  
11

Arguments to the `noexcept` specifier are often composed using the **`noexcept` operator**. The `noexcept` operator is a unary operator that returns a `bool` rvalue

constant expression that indicates whether a given expression might throw. Like `sizeof` (§ 4.9, p. 156), `noexcept` does not evaluate its operand.

For example, this expression yields `true`:

```
noexcept(recoup(i)) // true if calling recoup can't throw, false otherwise
```

because we declared `recoup` with a `noexcept` specifier. More generally,

```
noexcept(e)
```

is `true` if all the functions called by `e` have nonthrowing specifications and `e` itself does not contain a `throw`. Otherwise, `noexcept(e)` returns `false`.

We can use the `noexcept` operator to form an exception specifier as follows:

```
void f() noexcept(noexcept(g())); // f has same exception specifier as g
```

If the function `g` promises not to throw, then `f` also is nonthrowing. If `g` has no exception specifier, or has an exception specifier that allows exceptions, then `f` also might throw.



`noexcept` has two meanings: It is an exception specifier when it follows a function's parameter list, and it is an operator that is often used as the `bool` argument to a `noexcept` exception specifier.

## Exception Specifications and Pointers, Virtuals, and Copy Control

Although the `noexcept` specifier is not part of a function's type, whether a function has an exception specification affects the use of that function.

A pointer to function and the function to which that pointer points must have compatible specifications. That is, if we declare a pointer that has a nonthrowing exception specification, we can use that pointer only to point to similarly qualified functions. A pointer that specifies (explicitly or implicitly) that it might throw can point to any function, even if that function includes a promise not to throw:

```
// both recoup and pf1 promise not to throw
void (*pf1)(int) noexcept = recoup;
// ok: recoup won't throw; it doesn't matter that pf2 might
void (*pf2)(int) = recoup;
pf1 = alloc; // error: alloc might throw but pf1 said it wouldn't
pf2 = alloc; // ok: both pf2 and alloc might throw
```

If a virtual function includes a promise not to throw, the inherited virtuals must also promise not to throw. On the other hand, if the base allows exceptions, it is okay for the derived functions to be more restrictive and promise not to throw:

```
class Base {
public:
    virtual double f1(double) noexcept; // doesn't throw
    virtual int f2() noexcept(false); // can throw
    virtual void f3(); // can throw
};
```

```

class Derived : public Base {
public:
    double f1(double);           // error: Base::f1 promises not to throw
    int f2() noexcept(false);   // ok: same specification as Base::f2
    void f3() noexcept;         // ok: Derived f3 is more restrictive
};

```

When the compiler synthesizes the copy-control members, it generates an exception specification for the synthesized member. If all the corresponding operation for all the members and base classes promise not to throw, then the synthesized member is noexcept. If any function invoked by the synthesized member can throw, then the synthesized member is noexcept(false). Moreover, if we do not provide an exception specification for a destructor that we do define, the compiler synthesizes one for us. The compiler generates the same specification as it would have generated had it synthesized the destructor for that class.

### EXERCISES SECTION 18.1.4

**Exercise 18.8:** Review the classes you've written and add appropriate exception specifications to their constructors and destructors. If you think one of your destructors might throw, change the code so that it cannot throw.

## 18.1.5 Exception Class Hierarchies

The standard-library exception classes (§ 5.6.3, p. 197) form the inheritance hierarchy (Chapter 15) as shown in Figure 18.1.

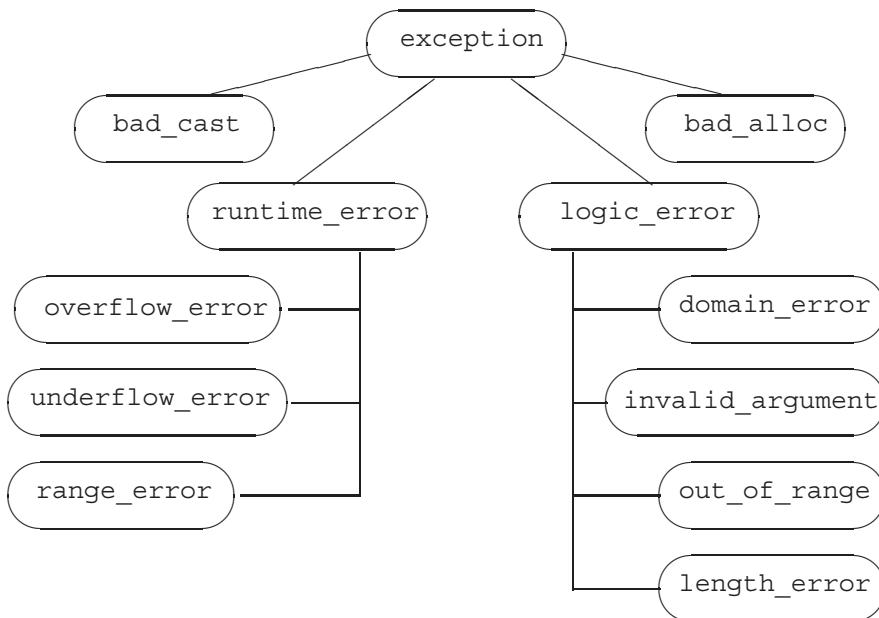
The only operations that the exception types define are the copy constructor, copy-assignment operator, a virtual destructor, and a virtual member named what. The what function returns a const char\* that points to a null-terminated character array, and is guaranteed not to throw any exceptions.

The exception, bad\_cast, and bad\_alloc classes also define a default constructor. The runtime\_error and logic\_error classes do not have a default constructor but do have constructors that take a C-style character string or a library string argument. Those arguments are intended to give additional information about the error. In these classes, what returns the message used to initialize the exception object. Because what is virtual, if we catch a reference to the base-type, a call to the what function will execute the version appropriate to the dynamic type of the exception object.

### Exception Classes for a Bookstore Application

Applications often extend the exception hierarchy by defining classes derived from exception (or from one of the library classes derived from exception). These application-specific classes represent exceptional conditions specific to the application domain.

Figure 18.1: Standard exception Class Hierarchy



If we were building a real bookstore application, our classes would have been much more complicated than the ones presented in this Primer. One such complexity would be how these classes handled exceptions. In fact, we probably would have defined our own hierarchy of exceptions to represent application-specific problems. Our design might include classes such as

```
// hypothetical exception classes for a bookstore application
class out_of_stock: public std::runtime_error {
public:
    explicit out_of_stock(const std::string &s):
        std::runtime_error(s) { }
};

class isbn_mismatch: public std::logic_error {
public:
    explicit isbn_mismatch(const std::string &s):
        std::logic_error(s) { }
    isbn_mismatch(const std::string &s,
                  const std::string &lhs, const std::string &rhs):
        std::logic_error(s), left(lhs), right(rhs) { }
    const std::string left, right;
};
```

Our application-specific exception types inherit them from the standard exception classes. As with any hierarchy, we can think of the exception classes as being

organized into layers. As the hierarchy becomes deeper, each layer becomes a more specific exception. For example, the first and most general layer of the hierarchy is represented by class `exception`. All we know when we catch an object of type `exception` is that something has gone wrong.

The second layer specializes `exception` into two broad categories: run-time or logic errors. Run-time errors represent things that can be detected only when the program is executing. Logic errors are, in principle, errors that we could have detected in our application.

Our bookstore exception classes further refine these categories. The class named `out_of_stock` represents something, particular to our application, that can go wrong at run time. It would be used to signal that an order cannot be fulfilled. The class `isbn_mismatch` represents a more particular form of `logic_error`. In principle, a program could prevent and handle this error by comparing the results of `isbn()` on the objects.

## Using Our Own Exception Types

We use our own exception classes in the same way that we use one of the standard library classes. One part of the program throws an object of one of these types, and another part catches and handles the indicated problem. As an example, we might define the compound addition operator for our `Sales_data` class to throw an error of type `isbn_mismatch` if it detected that the ISBNs didn't match:

```
// throws an exception if both objects do not refer to the same book
Sales_data&
Sales_data::operator+=(const Sales_data& rhs)
{
    if (isbn() != rhs.isbn())
        throw isbn_mismatch("wrong ISBNs", isbn(), rhs.isbn());
    units_sold += rhs.units_sold;
    revenue += rhs.revenue;
    return *this;
}
```

Code that uses the compound addition operator (or ordinary addition operator, which itself uses the compound addition operator) can detect this error, write an appropriate error message, and continue:

```
// use the hypothetical bookstore exceptions
Sales_data item1, item2, sum;
while (cin >> item1 >> item2) { // read two transactions
    try {
        sum = item1 + item2; // calculate their sum
        // use sum
    } catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left ISBN(" << e.left
            << ") right ISBN(" << e.right << ")" << endl;
    }
}
```

**EXERCISES SECTION 18.1.5**

**Exercise 18.9:** Define the bookstore exception classes described in this section and rewrite your `Sales_data` compound assignment operator to throw an exception.

**Exercise 18.10:** Write a program that uses the `Sales_data` addition operator on objects that have differing ISBNs. Write two versions of the program: one that handles the exception and one that does not. Compare the behavior of the programs so that you become familiar with what happens when an uncaught exception occurs.

**Exercise 18.11:** Why is it important that the `what` function doesn't throw?

## 18.2 Namespaces

Large programs tend to use independently developed libraries. Such libraries also tend to define a large number of global names, such as classes, functions, and templates. When an application uses libraries from many different vendors, it is almost inevitable that some of these names will clash. Libraries that put names into the global namespace are said to cause **namespace pollution**.

Traditionally, programmers avoided namespace pollution by using very long names for the global entities they defined. Those names often contained a prefix indicating which library defined the name:

```
class cplusplus_primer_Query { ... };  
string cplusplus_primer_make_plural(size_t, string&);
```

This solution is far from ideal: It can be cumbersome for programmers to write and read programs that use such long names.

**Namespaces** provide a much more controlled mechanism for preventing name collisions. Namespaces partition the global namespace. A namespace is a scope. By defining a library's names inside a namespace, library authors (and users) can avoid the limitations inherent in global names.

### 18.2.1 Namespace Definitions

A namespace definition begins with the keyword `namespace` followed by the namespace name. Following the namespace name is a sequence of declarations and definitions delimited by curly braces. Any declaration that can appear at global scope can be put into a namespace: classes, variables (with their initializations), functions (with their definitions), templates, and other namespaces:

```
namespace cplusplus_primer {  
    class Sales_data { /* ... */;  
        Sales_data operator+(const Sales_data&,  
                             const Sales_data&);  
        class Query { /* ... */ };  
        class Query_base { /* ... */ };  
    } // like blocks, namespaces do not end with a semicolon
```

This code defines a namespace named `cplusplus_primer` with four members: three classes and an overloaded `+` operator.

As with any name, a namespace name must be unique within the scope in which the namespace is defined. Namespaces may be defined at global scope or inside another namespace. They may not be defined inside a function or a class.



A namespace scope does not end with a semicolon.

## Each Namespace Is a Scope

As is the case for any scope, each name in a namespace must refer to a unique entity within that namespace. Because different namespaces introduce different scopes, different namespaces may have members with the same name.

Names defined in a namespace may be accessed directly by other members of the namespace, including scopes nested within those members. Code outside the namespace must indicate the namespace in which the name is defined:

```
cplusplus_primer::Query q =
    cplusplus_primer::Query("hello");
```

If another namespace (say, `AddisonWesley`) also provides a `Query` class and we want to use that class instead of the one defined in `cplusplus_primer`, we can do so by modifying our code as follows:

```
AddisonWesley::Query q = AddisonWesley::Query("hello");
```

## Namespaces Can Be Discontiguous

As we saw in § 16.5 (p. 709), unlike other scopes, a namespace can be defined in several parts. Writing a namespace definition:

```
namespace nsp {
// declarations
}
```

either defines a new namespace named `nsp` or adds to an existing one. If the name `nsp` does not refer to a previously defined namespace, then a new namespace with that name is created. Otherwise, this definition opens an existing namespace and adds declarations to that already existing namespace.

The fact that namespace definitions can be discontiguous lets us compose a namespace from separate interface and implementation files. Thus, a namespace can be organized in the same way that we manage our own class and function definitions:

- Namespace members that define classes, and declarations for the functions and objects that are part of the class interface, can be put into header files. These headers can be included by files that use those namespace members.
- The definitions of namespace members can be put in separate source files.

Organizing our namespaces this way also satisfies the requirement that various entities—non-inline functions, static data members, variables, and so forth—may be defined only once in a program. This requirement applies equally to names defined in a namespace. By separating the interface and implementation, we can ensure that the functions and other names we need are defined only once, but the same declaration will be seen whenever the entity is used.



Namespaces that define multiple, unrelated types should use separate files to represent each type (or each collection of related types) that the namespace defines.

## Defining the Primer Namespace

Using this strategy for separating interface and implementation, we might define the `cplusplus_primer` library in several separate files. The declarations for `Sales_data` and its related functions would be placed in `Sales_data.h`, those for the `Query` classes of Chapter 15 in `Query.h`, and so on. The corresponding implementation files would be in files such as `Sales_data.cc` and `Query.cc`:

```
// ---- Sales_data.h ---
// #includes should appear before opening the namespace
#include <string>
namespace cplusplus_primer {
    class Sales_data { /* ... */;
    Sales_data operator+(const Sales_data&,
                          const Sales_data&);
    // declarations for the remaining functions in the Sales_data interface
}
// ---- Sales_data.cc ---
// be sure any #includes appear before opening the namespace
#include "Sales_data.h"
namespace cplusplus_primer {
    // definitions for Sales_data members and overloaded operators
}
```

A program using our library would include whichever headers it needed. The names in those headers are defined inside the `cplusplus_primer` namespace:

```
// ---- user.cc ---
// names in the Sales_data.h header are in the cplusplus_primer namespace
#include "Sales_data.h"
int main()
{
    using cplusplus_primer::Sales_data;
    Sales_data trans1, trans2;
    // ...
    return 0;
}
```

This program organization gives the developers and the users of our library the needed modularity. Each class is still organized into its own interface and

implementation files. A user of one class need not compile names related to the others. We can hide the implementations from our users, while allowing the files `Sales_data.cc` and `user.cc` to be compiled and linked into one program without causing any compile-time or link-time errors. Developers of the library can work independently on the implementation of each type.

It is worth noting that ordinarily, we do not put a `#include` inside the namespace. If we did, we would be attempting to define all the names in that header as members of the enclosing namespace. For example, if our `Sales_data.h` file opened the `cplusplus_primer` before including the `string` header our program would be in error. It would be attempting to define the `std` namespace nested inside `cplusplus_primer`.

## Defining Namespace Members

Assuming the appropriate declarations are in scope, code inside a namespace may use the short form for names defined in the same (or in an enclosing) namespace:

```
#include "Sales_data.h"
namespace cplusplus_primer {    // reopen cplusplus_primer
    // members defined inside the namespace may use unqualified names
    std::istream&
    operator>>(std::istream& in, Sales_data& s) { /* ... */ }
}
```

It is also possible to define a namespace member outside its namespace definition. The namespace declaration of the name must be in scope, and the definition must specify the namespace to which the name belongs:

```
// namespace members defined outside the namespace must use qualified names
cplusplus_primer::Sales_data
cplusplus_primer::operator+(const Sales_data& lhs,
                           const Sales_data& rhs)
{
    Sales_data ret(lhs);
    // ...
}
```

As with class members defined outside a class, once the fully qualified name is seen, we are in the scope of the namespace. Inside the `cplusplus_primer` namespace, we can use other namespace member names without qualification. Thus, even though `Sales_data` is a member of the `cplusplus_primer` namespace, we can use its unqualified name to define the parameters in this function.

Although a namespace member can be defined outside its namespace, such definitions must appear in an enclosing namespace. That is, we can define the `Sales_data operator+` inside the `cplusplus_primer` namespace or at global scope. We cannot define this operator in an unrelated namespace.

## Template Specializations

Template specializations must be defined in the same namespace that contains the original template (§ 16.5, p. 709). As with any other namespace name, so long as

we have declared the specialization inside the namespace, we can define it outside the namespace:

```
// we must declare the specialization as a member of std
namespace std {
    template <> struct hash<Sales_data>;
}

// having added the declaration for the specialization to std
// we can define the specialization outside the std namespace
template <> struct std::hash<Sales_data>
{
    size_t operator()(const Sales_data& s) const
    { return hash<string>()(s.bookNo) ^
           hash<unsigned>()(s.units_sold) ^
           hash<double>()(s.revenue); }
    // other members as before
};
```

## The Global Namespace

Names defined at global scope (i.e., names declared outside any class, function, or namespace) are defined inside the **global namespace**. The global namespace is implicitly declared and exists in every program. Each file that defines entities at global scope (implicitly) adds those names to the global namespace.

The scope operator can be used to refer to members of the global namespace. Because the global namespace is implicit, it does not have a name; the notation

```
::member_name
```

refers to a member of the global namespace.

## Nested Namespaces

A nested namespace is a namespace defined inside another namespace:

```
namespace cplusplus_primer {
    // first nested namespace: defines the Query portion of the library
    namespace QueryLib {
        class Query { /* ... */ };
        Query operator&(const Query&, const Query&);
        // ...
    }
    // second nested namespace: defines the Sales_data portion of the library
    namespace Bookstore {
        class Quote { /* ... */ };
        class Disc_quote : public Quote { /* ... */ };
        // ...
    }
}
```

The `cplusplus_primer` namespace now contains two nested namespaces: the namespaces named `QueryLib` and `Bookstore`.

A nested namespace is a nested scope—its scope is nested within the namespace that contains it. Nested namespace names follow the normal rules: Names declared in an inner namespace hide declarations of the same name in an outer namespace. Names defined inside a nested namespace are local to that inner namespace. Code in the outer parts of the enclosing namespace may refer to a name in a nested namespace only through its qualified name: For example, the name of the class declared in the nested namespace `QueryLib` is

```
cplusplus_primer::QueryLib::Query
```

## Inline Namespaces

**C++ 11** The new standard introduced a new kind of nested namespace, an **inline namespace**. Unlike ordinary nested namespaces, names in an inline namespace can be used as if they were direct members of the enclosing namespace. That is, we need not qualify names from an inline namespace by their namespace name. We can access them using only the name of the enclosing namespace.

An inline namespace is defined by preceding the keyword `namespace` with the keyword `inline`:

```
inline namespace FifthEd {
    // namespace for the code from the Primer Fifth Edition
}
namespace FifthEd { // implicitly inline
    class Query_base { /* ... */;
        // other Query-related declarations
}
```

The keyword must appear on the first definition of the namespace. If the namespace is later reopened, the keyword `inline` need not be, but may be, repeated.

Inline namespaces are often used when code changes from one release of an application to the next. For example, we can put all the code from the current edition of the Primer into an inline namespace. Code for previous versions would be in non-inlined namespaces:

```
namespace FourthEd {
    class Item_base { /* ... */;
    class Query_base { /* ... */;
        // other code from the Fourth Edition
}
```

The overall `cplusplus_primer` namespace would include the definitions of both namespaces. For example, assuming that each namespace was defined in a header with the corresponding name, we'd define `cplusplus_primer` as follows:

```
namespace cplusplus_primer {
#include "FifthEd.h"
#include "FourthEd.h"
}
```

Because `FifthEd` is inline, code that refers to `cplusplus_primer::` will get the version from that namespace. If we want the earlier edition code, we can access it as we would any other nested namespace, by using the names of all the enclosing namespaces: for example, `cplusplus_primer::FourthEd::Query_base`.

## Unnamed Namespaces

An **unnamed namespace** is the keyword `namespace` followed immediately by a block of declarations delimited by curly braces. Variables defined in an unnamed namespace have static lifetime: They are created before their first use and destroyed when the program ends.

An unnamed namespace may be discontiguous within a given file but does not span files. Each file has its own unnamed namespace. If two files contain unnamed namespaces, those namespaces are unrelated. Both unnamed namespaces can define the same name; those definitions would refer to different entities. If a header defines an unnamed namespace, the names in that namespace define different entities local to each file that includes the header.



Unlike other namespaces, an unnamed namespace is local to a particular file and never spans multiple files.

Names defined in an unnamed namespace are used directly; after all, there is no namespace name with which to qualify them. It is not possible to use the scope operator to refer to members of unnamed namespaces.

Names defined in an unnamed namespace are in the same scope as the scope at which the namespace is defined. If an unnamed namespace is defined at the outermost scope in the file, then names in the unnamed namespace must differ from names defined at global scope:

```
int i;    // global declaration for i
namespace {
    int i;
}
// ambiguous: defined globally and in an unnested, unnamed namespace
i = 10;
```

In all other ways, the members of an unnamed namespace are normal program entities. An unnamed namespace, like any other namespace, may be nested inside another namespace. If the unnamed namespace is nested, then names in it are accessed in the normal way, using the enclosing namespace name(s):

```
namespace local {
    namespace {
        int i;
    }
}
// ok: i defined in a nested unnamed namespace is distinct from global i
local::i = 42;
```

**UNNAMED NAMESPACES REPLACE FILE STATICS**

Prior to the introduction of namespaces, programs declared names as `static` to make them local to a file. The use of *file statics* is inherited from C. In C, a global entity declared `static` is invisible outside the file in which it is declared.



The use of file `static` declarations is deprecated by the C++ standard. File statics should be avoided and unnamed namespaces used instead.

**EXERCISES SECTION 18.2.1**

**Exercise 18.12:** Organize the programs you have written to answer the questions in each chapter into their own namespaces. That is, namespace `chapter15` would contain code for the `Query` programs and `chapter10` would contain the `TextQuery` code. Using this structure, compile the `Query` code examples.

**Exercise 18.13:** When might you use an unnamed namespace?

**Exercise 18.14:** Suppose we have the following declaration of the `operator*` that is a member of the nested namespace `mathLib::MatrixLib`:

```
namespace mathLib {
    namespace MatrixLib {
        class matrix { /* ... */ };
        matrix operator*
            (const matrix &, const matrix &);
        // ...
    }
}
```

How would you declare this operator in global scope?

**18.2.2 Using Namespace Members**

Referring to namespace members as `namespace_name::member_name` is admittedly cumbersome, especially if the namespace name is long. Fortunately, there are ways to make it easier to use namespace members. Our programs have used one of these ways, using declarations (§ 3.1, p. 82). The others, namespace aliases and using directives, will be described in this section.

**Namespace Aliases**

A **namespace alias** can be used to associate a shorter synonym with a namespace name. For example, a long namespace name such as

```
namespace cplusplus_primer { /* ... */ };
```

can be associated with a shorter synonym as follows:

```
namespace primer = cplusplus_primer;
```

A namespace alias declaration begins with the keyword `namespace`, followed by the alias name, followed by the `=` sign, followed by the original namespace name and a semicolon. It is an error if the original namespace name has not already been defined as a namespace.

A namespace alias can also refer to a nested namespace:

```
namespace Qlib = cplusplus_primer::QueryLib;  
Qlib::Query q;
```



A namespace can have many synonyms, or aliases. All the aliases and the original namespace name can be used interchangeably.

## using Declarations: A Recap

A **using declaration** introduces only one namespace member at a time. It allows us to be very specific regarding which names are used in our programs.

Names introduced in a **using declaration** obey normal scope rules: They are visible from the point of the **using declaration** to the end of the scope in which the declaration appears. Entities with the same name defined in an outer scope are hidden. The unqualified name may be used only within the scope in which it is declared and in scopes nested within that scope. Once the scope ends, the fully qualified name must be used.

A **using declaration** can appear in global, local, namespace, or class scope. In class scope, such declarations may only refer to a base class member (§ 15.5, p. 615).

## using Directives

A **using directive**, like a **using declaration**, allows us to use the unqualified form of a namespace name. Unlike a **using declaration**, we retain no control over which names are made visible—they all are.

A **using directive** begins with the keyword `using`, followed by the keyword `namespace`, followed by a namespace name. It is an error if the name is not a previously defined namespace name. A **using directive** may appear in global, local, or namespace scope. It may not appear in a class scope.

These directives make all the names from a specific namespace visible without qualification. The short form names can be used from the point of the **using directive** to the end of the scope in which the **using directive** appears.



Providing a **using directive** for namespaces, such as `std`, that our application does not control reintroduces all the name collision problems inherent in using multiple libraries.

## using Directives and Scope

The scope of names introduced by a **using directive** is more complicated than the scope of names in **using declarations**. As we've seen, a **using declaration** puts the name in the same scope as that of the **using declaration** itself. It is as if the **using declaration** declares a local alias for the namespace member.

A `using` directive does not declare local aliases. Rather, it has the effect of lifting the namespace members into the nearest scope that contains both the namespace itself and the `using` directive.

This difference in scope between a `using` declaration and a `using` directive stems directly from how these two facilities work. In the case of a `using` declaration, we are simply making name directly accessible in the local scope. In contrast, a `using` directive makes the entire contents of a namespace available. In general, a namespace might include definitions that cannot appear in a local scope. As a consequence, a `using` directive is treated as if it appeared in the nearest enclosing namespace scope.

In the simplest case, assume we have a namespace `A` and a function `f`, both defined at global scope. If `f` has a `using` directive for `A`, then in `f` it will be as if the names in `A` appeared in the global scope prior to the definition of `f`:

```
// namespace A and function f are defined at global scope
namespace A {
    int i, j;
}
void f()
{
    using namespace A;      // injects the names from A into the global scope
    cout << i * j << endl; // uses i and j from namespace A
    // ...
}
```

## using Directives Example

Let's look at an example:

```
namespace blip {
    int i = 16, j = 15, k = 23;
    // other declarations
}
int j = 0; // ok: j inside blip is hidden inside a namespace
void manip()
{
    // using directive; the names in blip are "added" to the global scope
    using namespace blip; // clash between ::j and blip::j
    // detected only if j is used
    ++i;           // sets blip::i to 17
    ++j;           // error ambiguous: global j or blip::j?
    ++::j;         // ok: sets global j to 1
    ++blip::j;     // ok: sets blip::j to 16
    int k = 97;   // local k hides blip::k
    ++k;           // sets local k to 98
}
```

The `using` directive in `manip` makes all the names in `blip` directly accessible; code inside `manip` can refer to the names of these members, using their short form.

The members of `blip` appear as if they were defined in the scope in which both `blip` and `manip` are defined. Assuming `manip` is defined at global scope, then the members of `blip` appear as if they were declared in global scope.

When a namespace is injected into an enclosing scope, it is possible for names in the namespace to conflict with other names defined in that (enclosing) scope. For example, inside `manip`, the `blip` member `j` conflicts with the global object named `j`. Such conflicts are permitted, but to use the name, we must explicitly indicate which version is wanted. Any unqualified use of `j` within `manip` is ambiguous.

To use a name such as `j`, we must use the scope operator to indicate which name is wanted. We would write `::j` to obtain the variable defined in global scope. To use the `j` defined in `blip`, we must use its qualified name, `blip::j`.

Because the names are in different scopes, local declarations within `manip` may hide some of the namespace member names. The local variable `k` hides the namespace member `blip::k`. Referring to `k` within `manip` is not ambiguous; it refers to the local variable `k`.

## Headers and using Declarations or Directives

A header that has a `using` directive or declaration at its top-level scope injects names into every file that includes the header. Ordinarily, headers should define only the names that are part of its interface, not names used in its own implementation. As a result, header files should not contain `using` directives or `using` declarations except inside functions or namespaces (§ 3.1, p. 83).

### CAUTION: AVOID USING DIRECTIVES

`using` directives, which inject all the names from a namespace, are deceptively simple to use: With only a single statement, all the member names of a namespace are suddenly visible. Although this approach may seem simple, it can introduce its own problems. If an application uses many libraries, and if the names within these libraries are made visible with `using` directives, then we are back to square one, and the global namespace pollution problem reappears.

Moreover, it is possible that a working program will fail to compile when a new version of the library is introduced. This problem can arise if a new version introduces a name that conflicts with a name that the application is using.

Another problem is that ambiguity errors caused by `using` directives are detected only at the point of use. This late detection means that conflicts can arise long after introducing a particular library. If the program begins using a new part of the library, previously undetected collisions may arise.

Rather than relying on a `using` directive, it is better to use a `using` declaration for each namespace name used in the program. Doing so reduces the number of names injected into the namespace. Ambiguity errors caused by `using` declarations are detected at the point of declaration, not use, and so are easier to find and fix.



One place where `using` directives are useful is in the implementation files of the namespace itself.

## EXERCISES SECTION 18.2.2

**Exercise 18.15:** Explain the differences between using declarations and directives.

**Exercise 18.16:** Explain the following code assuming using declarations for all the members of namespace `Exercise` are located at the location labeled *position 1*. What if they appear at *position 2* instead? Now answer the same question but replace the using declarations with a using directive for namespace `Exercise`.

```
namespace Exercise {
    int ivar = 0;
    double dvar = 0;
    const int limit = 1000;
}
int ivar = 0;
// position 1
void manip() {
    // position 2
    double dvar = 3.1416;
    int iobj = limit + 1;
    ++ivar;
    +++:ivar;
}
```

**Exercise 18.17:** Write code to test your answers to the previous question.

### 18.2.3 Classes, Namespaces, and Scope

Name lookup for names used inside a namespace follows the normal lookup rules: The search looks outward through the enclosing scopes. An enclosing scope might be one or more nested namespaces, ending in the all-encompassing global namespace. Only names that have been declared before the point of use that are in blocks that are still open are considered:

```
namespace A {
    int i;
    namespace B {
        int i;           // hides A::i within B
        int j;
        int f1()
        {
            int j;     // j is local to f1 and hides A::B::j
            return i; // returns B::i
        }
    } // namespace B is closed and names in it are no longer visible
    int f2()
    {
        return j;      // error: j is not defined
    }
    int j = i;       // initialized from A::i
}
```

When a class is wrapped in a namespace, the normal lookup still happens: When a name is used by a member function, look for that name in the member first, then within the class (including base classes), then look in the enclosing scopes, one or more of which might be a namespace:

```
namespace A {
    int i;
    int k;
    class C1 {
        public:
            C1() : i(0), j(0) { }      // ok: initializes C1::i and C1::j
            int f1() { return k; }     // returns A::k
            int f2() { return h; }    // error: h is not defined
            int f3();
        private:
            int i;                      // hides A::i within C1
            int j;
    };
    int h = i;                      // initialized from A::i
}
// member f3 is defined outside class C1 and outside namespace A
int A::C1::f3() { return h; } // ok: returns A::h
```

With the exception of member function definitions that appear inside the class body (§ 7.4.1, p. 283), scopes are always searched upward; names must be declared before they can be used. Hence, the `return` in `f2` will not compile. It attempts to reference the name `h` from namespace `A`, but `h` has not yet been defined. Had that name been defined in `A` before the definition of `C1`, the use of `h` would be legal. Similarly, the use of `h` inside `f3` is okay, because `f3` is defined after `A::h`.



The order in which scopes are examined to find a name can be inferred from the qualified name of a function. The qualified name indicates, in reverse order, the scopes that are searched.

The qualifiers `A::C1::f3` indicate the reverse order in which the class scopes and namespace scopes are to be searched. The first scope searched is that of the function `f3`. Then the class scope of its enclosing class `C1` is searched. The scope of the namespace `A` is searched last before the scope containing the definition of `f3` is examined.

## Argument-Dependent Lookup and Parameters of Class Type



Consider the following simple program:

```
std::string s;
std::cin >> s;
```

As we know, this call is equivalent to (§ 14.1, p. 553):

```
operator>>(std::cin, s);
```

This `operator>>` function is defined by the `string` library, which in turn is defined in the `std` namespace. Yet we can we call `operator>>` without an `std::` qualifier and without a `using` declaration.

We can directly access the output operator because there is an important exception to the rule that names defined in a namespace are hidden. When we pass an object of a class type to a function, the compiler searches the namespace in which the argument's class is defined *in addition* to the normal scope lookup. This exception also applies for calls that pass pointers or references to a class type.

In this example, when the compiler sees the “call” to `operator>>`, it looks for a matching function in the current scope, including the scopes enclosing the output statement. In addition, because the `>>` expression has parameters of class type, the compiler also looks in the namespace(s) in which the types of `cin` and `s` are defined. Thus, for this call, the compiler looks in the `std` namespace, which defines the `istream` and `string` types. When it searches `std`, the compiler finds the `string` output operator function.

This exception in the lookup rules allows nonmember functions that are conceptually part of the interface to a class to be used without requiring a separate `using` declaration. In the absence of this exception to the lookup rules, either we would have to provide an appropriate `using` declaration for the output operator:

```
using std::operator>>;           // needed to allow cin >> s
```

or we would have to use the function-call notation in order to include the namespace qualifier:

```
std::operator>>(std::cin, s);    // ok: explicitly use std:::>>
```

There would be no way to use operator syntax. Either of these declarations is awkward and would make simple uses of the IO library more complicated.

## Lookup and `std::move` and `std::forward`

Many, perhaps even most, C++ programmers never have to think about argument-dependent lookup. Ordinarily, if an application defines a name that is also defined in the library, one of two things is true: Either normal overloading determines (correctly) whether a particular call is intended for the application version or the one from the library, or the application never intends to use the library function.

Now consider the library `move` and `forward` functions. Both of these functions are template functions, and the library defines versions of them that have a single rvalue reference function parameter. As we've seen, in a function template, an rvalue reference parameter can match any type (§ 16.2.6, p. 690). If our application defines a function named `move` that takes a single parameter, then—no matter what type the parameter has—the application's version of `move` will collide with the library version. Similarly for `forward`.

As a result, name collisions with `move` (and `forward`) are more likely than collisions with other library functions. In addition, because `move` and `forward` do very specialized type manipulations, the chances that an application specifically wants to override the behavior of these functions are pretty small.

The fact that collisions are more likely—and are less likely to be intentional—explains why we suggest always using the fully qualified versions of these names (§ 12.1.5, p. 470). So long as we write `std::move` rather than `move`, we know that we will get the version from the standard library.

## Friend Declarations and Argument-Dependent Lookup



Recall that when a class declares a friend, the friend declaration does not make the friend visible (§ 7.2.1, p. 270). However, an otherwise undeclared class or function that is first named in a friend declaration is assumed to be a member of the closest enclosing namespace. The combination of this rule and argument-dependent lookup can lead to surprises:

```
namespace A {
    class C {
        // two friends; neither is declared apart from a friend declaration
        // these functions implicitly are members of namespace A
        friend void f2();           // won't be found, unless otherwise declared
        friend void f(const C&);   // found by argument-dependent lookup
    };
}
```

Here, both `f` and `f2` are members of namespace `A`. Through argument-dependent lookup, we can call `f` even if there is no additional declaration for `f`:

```
int main()
{
    A::C cobj;
    f(cobj);      // ok: finds A::f through the friend declaration in A::C
    f2();         // error: A::f2 not declared
}
```

Because `f` takes an argument of a class type, and `f` is implicitly declared in the same namespace as `C`, `f` is found when called. Because `f2` has no parameter, it will not be found.

### EXERCISES SECTION 18.2.3

**Exercise 18.18:** Given the following typical definition of `swap` § 13.3 (p. 517), determine which version of `swap` is used if `mem1` is a `string`. What if `mem1` is an `int`? Explain how name lookup works in both cases.

```
void swap(T v1, T v2)
{
    using std::swap;
    swap(v1.mem1, v2.mem1);
    // swap remaining members of type T
}
```

**Exercise 18.19:** What if the call to `swap` was `std::swap(v1.mem1, v2.mem1)`?

## 18.2.4 Overloading and Namespaces

Namespaces have two impacts on function matching (§ 6.4, p. 233). One of these should be obvious: A `using` declaration or directive can add functions to the candidate set. The other is much more subtle.



## Argument-Dependent Lookup and Overloading

As we saw in the previous section, name lookup for functions that have class-type arguments includes the namespace in which each argument's class is defined. This rule also impacts how we determine the candidate set. Each namespace that defines a class used as an argument (and those that define its base classes) is searched for candidate functions. Any functions in those namespaces that have the same name as the called function are added to the candidate set. These functions are added *even though they otherwise are not visible at the point of the call*:

```
namespace NS {
    class Quote { /* ... */ };
    void display(const Quote&) { /* ... */ }
}
// Bulk_item's base class is declared in namespace NS
class Bulk_item : public NS::Quote { /* ... */ };
int main() {
    Bulk_item book1;
    display(book1);
    return 0;
}
```

The argument we passed to `display` has class type `Bulk_item`. The candidate functions for the call to `display` are not only the functions with declarations that are in scope where `display` is called, but also the functions in the namespace where `Bulk_item` and its base class, `Quote`, are declared. The function `display(const Quote&)` declared in namespace `NS` is added to the set of candidate functions.

## Overloading and `using` Declarations

To understand the interaction between `using` declarations and overloading, it is important to remember that a `using` declaration declares a name, not a specific function (§ 15.6, p. 621):

```
using NS::print(int); // error: cannot specify a parameter list
using NS::print;      // ok: using declarations specify names only
```

When we write a `using` declaration for a function, all the versions of that function are brought into the current scope.

A `using` declaration incorporates all versions to ensure that the interface of the namespace is not violated. The author of a library provided different functions for a reason. Allowing users to selectively ignore some but not all of the functions from a set of overloaded functions could lead to surprising program behavior.

The functions introduced by a `using` declaration overload any other declarations of the functions with the same name already present in the scope where the `using` declaration appears. If the `using` declaration appears in a local scope, these names hide existing declarations for that name in the outer scope. If the `using` declaration introduces a function in a scope that already has a function of the same name with the same parameter list, then the `using` declaration is in error. Otherwise, the `using` declaration defines additional overloaded instances of the given name. The effect is to increase the set of candidate functions.

## Overloading and `using` Directives

A `using` directive lifts the namespace members into the enclosing scope. If a namespace function has the same name as a function declared in the scope at which the namespace is placed, then the namespace member is added to the overload set:

```
namespace libs_R_us {
    extern void print(int);
    extern void print(double);
}

// ordinary declaration
void print(const std::string &);

// this using directive adds names to the candidate set for calls to print:
using namespace libs_R_us;
// the candidates for calls to print at this point in the program are:
//   print(int) from libs_R_us
//   print(double) from libs_R_us
//   print(const std::string &) declared explicitly

void fooBar(int ival)
{
    print("Value: "); // calls global print(const string &)
    print(ival);      // calls libs_R_us::print(int)
}
```

Differently from how `using` declarations work, it is not an error if a `using` directive introduces a function that has the same parameters as an existing function. As with other conflicts generated by `using` directives, there is no problem unless we try to call the function without specifying whether we want the one from the namespace or from the current scope.

## Overloading across Multiple `using` Directives

If many `using` directives are present, then the names from each namespace become part of the candidate set:

```
namespace AW {
    int print(int);
}

namespace Primer {
    double print(double);
}
```

```
// using directives create an overload set of functions from different namespaces
using namespace AW;
using namespace Primer;
long double print(long double);
int main() {
    print(1);    // calls AW::print(int)
    print(3.1); // calls Primer::print(double)
    return 0;
}
```

The overload set for the function `print` in global scope contains the functions `print(int)`, `print(double)`, and `print(long double)`. These functions are all part of the overload set considered for the function calls in `main`, even though these functions were originally declared in different namespace scopes.

### EXERCISES SECTION 18.2.4

**Exercise 18.20:** In the following code, determine which function, if any, matches the call to `compute`. List the candidate and viable functions. What type conversions, if any, are applied to the argument to match the parameter in each viable function?

```
namespace primerLib {
    void compute();
    void compute(const void *);
}
using primerLib::compute;
void compute(int);
void compute(double, double = 3.4);
void compute(char*, char* = 0);
void f()
{
    compute(0);
}
```

What would happen if the `using` declaration were located in `main` before the call to `compute`? Answer the same questions as before.

## 18.3 Multiple and Virtual Inheritance

**Multiple inheritance** is the ability to derive a class from more than one direct base class (§ 15.2.2, p. 600). A multiply derived class inherits the properties of all its parents. Although simple in concept, the details of intertwining multiple base classes can present tricky design-level and implementation-level problems.

To explore multiple inheritance, we'll use a pedagogical example of a zoo animal hierarchy. Our zoo animals exist at different levels of abstraction. There are the individual animals, distinguished by their names, such as Ling-ling, Mowgli, and Balou. Each animal belongs to a species; Ling-Ling, for example, is a giant

panda. Species, in turn, are members of families. A giant panda is a member of the bear family. Each family, in turn, is a member of the animal kingdom—in this case, the more limited kingdom of a particular zoo.

We'll define an abstract `ZooAnimal` class to hold information that is common to all the zoo animals and provides the most general interface. The `Bear` class will contain information that is unique to the `Bear` family, and so on.

In addition to the `ZooAnimal` classes, our application will contain auxiliary classes that encapsulate various abstractions such as endangered animals. In our implementation of a `Panda` class, for example, a `Panda` is multiply derived from `Bear` and `Endangered`.

### 18.3.1 Multiple Inheritance

The derivation list in a derived class can contain more than one base class:

```
class Bear : public ZooAnimal {  
    class Panda : public Bear, public Endangered { /* ... */ };
```

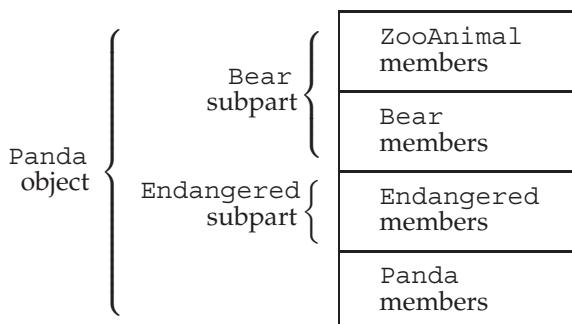
Each base class has an optional access specifier (§ 15.5, p. 612). As usual, if the access specifier is omitted, the specifier defaults to `private` if the `class` keyword is used and to `public` if `struct` is used (§ 15.5, p. 616).

As with single inheritance, the derivation list may include only classes that have been defined and that were not defined as `final` (§ 15.2.2, p. 600). There is no language-imposed limit on the number of base classes from which a class can be derived. A base class may appear only once in a given derivation list.

### Multiply Derived Classes Inherit State from Each Base Class

Under multiple inheritance, an object of a derived class contains a subobject for each of its base classes (§ 15.2.2, p. 597). For example, as illustrated in Figure 18.2, a `Panda` object has a `Bear` part (which itself contains a `ZooAnimal` part), an `Endangered` class part, and the `nonstatic` data members, if any, declared within the `Panda` class.

Figure 18.2: Conceptual Structure of a `Panda` Object



## Derived Constructors Initialize All Base Classes

Constructing an object of derived type constructs and initializes all its base sub-objects. As is the case for inheriting from a single base class (§ 15.2.2, p. 598), a derived type's constructor initializer may initialize only its direct base classes:

```
// explicitly initialize both base classes
Panda::Panda(std::string name, bool onExhibit)
    : Bear(name, onExhibit, "Panda"),
      Endangered(Endangered::critical) { }

// implicitly uses the Bear default constructor to initialize the Bear subobject
Panda::Panda()
    : Endangered(Endangered::critical) { }
```

The constructor initializer list may pass arguments to each of the direct base classes. The order in which base classes are constructed depends on the order in which they appear in the class derivation list. The order in which they appear in the constructor initializer list is irrelevant. A `Panda` object is initialized as follows:

- `ZooAnimal`, the ultimate base class up the hierarchy from `Panda`'s first direct base class, `Bear`, is initialized first.
- `Bear`, the first direct base class, is initialized next.
- `Endangered`, the second direct base, is initialized next.
- `Panda`, the most derived part, is initialized last.

## Inherited Constructors and Multiple Inheritance

C++  
11

Under the new standard, a derived class can inherit its constructors from one or more of its base classes (§ 15.7.4, p. 628). It is an error to inherit the same constructor (i.e., one with the same parameter list) from more than one base class:

```
struct Base1 {
    Base1() = default;
    Base1(const std::string&);
    Base1(std::shared_ptr<int>);

};

struct Base2 {
    Base2() = default;
    Base2(const std::string&);
    Base2(int);

};

// error: D1 attempts to inherit D1::D1 (const string&) from both base classes
struct D1: public Base1, public Base2 {
    using Base1::Base1; // inherit constructors from Base1
    using Base2::Base2; // inherit constructors from Base2
};
```

A class that inherits the same constructor from more than one base class must define its own version of that constructor:

```
struct D2: public Base1, public Base2 {  
    using Base1::Base1; // inherit constructors from Base1  
    using Base2::Base2; // inherit constructors from Base2  
    // D2 must define its own constructor that takes a string  
    D2(const string &s): Base1(s), Base2(s) {}  
    D2() = default; // needed once D2 defines its own constructor  
};
```

## Destructors and Multiple Inheritance

As usual, the destructor in a derived class is responsible for cleaning up resources allocated by that class only—the members and all the base class(es) of the derived class are automatically destroyed. The synthesized destructor has an empty function body.

Destructors are always invoked in the reverse order from which the constructors are run. In our example, the order in which the destructors are called is `~Panda, ~Endangered, ~Bear, ~ZooAnimal`.

## Copy and Move Operations for Multiply Derived Classes

As is the case for single inheritance, classes with multiple bases that define their own copy/move constructors and assignment operators must copy, move, or assign the whole object (§ 15.7.2, p. 623). The base parts of a multiply derived class are automatically copied, moved, or assigned only if the derived class uses the synthesized versions of these members. In the synthesized copy-control members, each base class is implicitly constructed, assigned, or destroyed, using the corresponding member from that base class.

For example, assuming that Panda uses the synthesized members, then the initialization of `ling_ling`:

```
Panda ying_yang("ying_yang");  
Panda ling_ling = ying_yang; // uses the copy constructor
```

will invoke the Bear copy constructor, which in turn runs the ZooAnimal copy constructor before executing the Bear copy constructor. Once the Bear portion of `ling_ling` is constructed, the Endangered copy constructor is run to create that part of the object. Finally, the Panda copy constructor is run. Similarly, for the synthesized move constructor.

The synthesized copy-assignment operator behaves similarly to the copy constructor. It assigns the Bear (and through Bear, the ZooAnimal) parts of the object first. Next, it assigns the Endangered part, and finally the Panda part. Move assignment behaves similarly.

### 18.3.2 Conversions and Multiple Base Classes

Under single inheritance, a pointer or a reference to a derived class can be converted automatically to a pointer or a reference to an accessible base class (§ 15.2.2, p. 597, and § 15.5, p. 613). The same holds true with multiple inheritance. A pointer or reference to any of an object's (accessible) base classes can be used to point or

### EXERCISES SECTION 18.3.1

**Exercise 18.21:** Explain the following declarations. Identify any that are in error and explain why they are incorrect:

- (a) class CADVehicle : public CAD, Vehicle { ... };
- (b) class DblList: public List, public List { ... };
- (c) class iostream: public istream, public ostream { ... };

**Exercise 18.22:** Given the following class hierarchy, in which each class defines a default constructor:

```
class A { ... };
class B : public A { ... };
class C : public B { ... };
class X { ... };
class Y { ... };
class Z : public X, public Y { ... };
class MI : public C, public Z { ... };
```

what is the order of constructor execution for the following definition?

```
MI mi;
```

refer to a derived object. For example, a pointer or reference to ZooAnimal, Bear, or Endangered can be bound to a Panda object:

```
// operations that take references to base classes of type Panda
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const ZooAnimal&);

Panda ying_yang("ying_yang");
print(ying_yang);           // passes Panda to a reference to Bear
highlight(ying_yang);      // passes Panda to a reference to Endangered
cout << ying_yang << endl; // passes Panda to a reference to ZooAnimal
```

The compiler makes no attempt to distinguish between base classes in terms of a derived-class conversion. Converting to each base class is equally good. For example, if there was an overloaded version of `print`:

```
void print(const Bear&);
void print(const Endangered&);
```

an unqualified call to `print` with a Panda object would be a compile-time error:

```
Panda ying_yang("ying_yang");
print(ying_yang);           // error: ambiguous
```

### Lookup Based on Type of Pointer or Reference

As with single inheritance, the static type of the object, pointer, or reference determines which members we can use (§ 15.6, p. 617). If we use a `ZooAnimal` pointer,

only the operations defined in that class are usable. The Bear-specific, Panda-specific, and Endangered portions of the Panda interface are invisible. Similarly, a Bear pointer or reference knows only about the Bear and ZooAnimal members; an Endangered pointer or reference is limited to the Endangered members.

As an example, consider the following calls, which assume that our classes define the virtual functions listed in Table 18.1:

```
Bear *pb = new Panda("ying_yang");
pb->print();           // ok: Panda::print()
pb->cuddle();          // error: not part of the Bear interface
pb->highlight();        // error: not part of the Bear interface
delete pb;              // ok: Panda::~Panda()
```

When a Panda is used via an Endangered pointer or reference, the Panda-specific and Bear portions of the Panda interface are invisible:

```
Endangered *pe = new Panda("ying_yang");
pe->print();           // ok: Panda::print()
pe->toes();             // error: not part of the Endangered interface
pe->cuddle();           // error: not part of the Endangered interface
pe->highlight();         // ok: Panda::highlight()
delete pe;               // ok: Panda::~Panda()
```

**Table 18.1: Virtual Functions in the ZooAnimal/Endangered Classes**

| Function   | Class Defining Own Version                                                   |
|------------|------------------------------------------------------------------------------|
| print      | ZooAnimal::ZooAnimal<br>Bear::Bear<br>Endangered::Endangered<br>Panda::Panda |
| highlight  | Endangered::Endangered<br>Panda::Panda                                       |
| toes       | Bear::Bear<br>Panda::Panda                                                   |
| cuddle     | Panda::Panda                                                                 |
| destructor | ZooAnimal::ZooAnimal<br>Endangered::Endangered                               |

### 18.3.3 Class Scope under Multiple Inheritance

Under single inheritance, the scope of a derived class is nested within the scope of its direct and indirect base classes (§ 15.6, p. 617). Lookup happens by searching up the inheritance hierarchy until the given name is found. Names defined in a derived class hide uses of that name inside a base.

Under multiple inheritance, this same lookup happens *simultaneously* among all the direct base classes. If a name is found through more than one base class, then use of that name is ambiguous.

## EXERCISES SECTION 18.3.2

**Exercise 18.23:** Using the hierarchy in exercise 18.22 along with class D defined below, and assuming each class defines a default constructor, which, if any, of the following conversions are not permitted?

```
class D : public X, public C { ... };
D *pd = new D;
(a) X *px = pd;      (b) A *pa = pd;
(c) B *pb = pd;      (d) C *pc = pd;
```

**Exercise 18.24:** On page 807 we presented a series of calls made through a Bear pointer that pointed to a Panda object. Explain each call assuming we used a ZooAnimal pointer pointing to a Panda object instead.

**Exercise 18.25:** Assume we have two base classes, Base1 and Base2, each of which defines a virtual member named print and a virtual destructor. From these base classes we derive the following classes, each of which redefines the print function:

```
class D1 : public Base1 { /* ... */ };
class D2 : public Base2 { /* ... */ };
class MI : public D1, public D2 { /* ... */ };
```

Using the following pointers, determine which function is used in each call:

```
Base1 *pb1 = new MI;
Base2 *pb2 = new MI;
D1 *pd1 = new MI;
D2 *pd2 = new MI;

(a) pb1->print();  (b) pd1->print();  (c) pd2->print();
(d) delete pb2;    (e) delete pd1;    (f) delete pd2;
```

In our example, if we use a name through a Panda object, pointer, or reference, both the Endangered and the Bear/ZooAnimal subtrees are examined in parallel. If the name is found in more than one subtree, then the use of the name is ambiguous. It is perfectly legal for a class to inherit multiple members with the same name. However, if we want to use that name, we must specify which version we want to use.



When a class has multiple base classes, it is possible for that derived class to inherit a member with the same name from two or more of its base classes. Unqualified uses of that name are ambiguous.

For example, if both ZooAnimal and Endangered define a member named max\_weight, and Panda does not define that member, this call is an error:

```
double d = ying_yang.max_weight();
```

The derivation of Panda, which results in Panda having two members named max\_weight, is perfectly legal. The derivation generates a *potential* ambiguity. That ambiguity is avoided if no Panda object ever calls max\_weight. The error

would also be avoided if each call to `max_weight` specifically indicated which version to run—`ZooAnimal::max_weight` or `Endangered::max_weight`. An error results only if there is an ambiguous attempt to use the member.

The ambiguity of the two inherited `max_weight` members is reasonably obvious. It might be more surprising to learn that an error would be generated even if the two inherited functions had different parameter lists. Similarly, it would be an error even if the `max_weight` function were `private` in one class and `public` or `protected` in the other. Finally, if `max_weight` were defined in `Bear` and not in `ZooAnimal`, the call would still be in error.

As always, name lookup happens before type checking (§ 6.4.1, p. 234). When the compiler finds `max_weight` in two different scopes, it generates an error noting that the call is ambiguous.

The best way to avoid potential ambiguities is to define a version of the function in the derived class that resolves the ambiguity. For example, we should give our `Panda` class a `max_weight` function that resolves the ambiguity:

```
double Panda::max_weight() const
{
    return std::max(ZooAnimal::max_weight(),
                    Endangered::max_weight());
}
```

### EXERCISES SECTION 18.3.3

**Exercise 18.26:** Given the hierarchy in the box on page 810, why is the following call to `print` an error? Revise MI to allow this call to `print` to compile and execute correctly.

```
MI mi;
mi.print(42);
```

**Exercise 18.27:** Given the class hierarchy in the box on page 810 and assuming we add a function named `foo` to MI as follows:

```
int ival;
double dval;
void MI::foo(double cval)
{
    int dval;
    // exercise questions occur here
}
```

- (a) List all the names visible from within `MI::foo`.
- (b) Are any names visible from more than one base class?
- (c) Assign to the local instance of `dval` the sum of the `dval` member of `Base1` and the `dval` member of `Derived`.
- (d) Assign the value of the last element in `MI::dvec` to `Base2::fval`.
- (e) Assign `cval` from `Base1` to the first character in `sval` from `Derived`.

**CODE FOR EXERCISES TO SECTION 18.3.3**

```

struct Basel {
    void print(int) const;           // public by default
protected:
    int      ival;
    double   dval;
    char     cval;
private:
    int      *id;
};

struct Base2 {
    void print(double) const;       // public by default
protected:
    double   fval;
private:
    double   dval;
};

struct Derived : public Basel {
    void print(std::string) const;   // public by default
protected:
    std::string sval;
    double      dval;
};

struct MI : public Derived, public Base2 {
    void print(std::vector<double>); // public by default
protected:
    int          *ival;
    std::vector<double>  dvec;
};

```

### 18.3.4 Virtual Inheritance

Although the derivation list of a class may not include the same base class more than once, a class can inherit from the same base class more than once. It might inherit the same base indirectly from two of its own direct base classes, or it might inherit a particular class directly and indirectly through another of its base classes.

As an example, the IO library `istream` and `ostream` classes each inherit from a common abstract base class named `basic_ios`. That class holds the stream's buffer and manages the stream's condition state. The class `iostream`, which can both read and write to a stream, inherits directly from both `istream` and `ostream`. Because both types inherit from `basic_ios`, `iostream` inherits that base class twice, once through `istream` and once through `ostream`.

By default, a derived object contains a separate subpart corresponding to each class in its derivation chain. If the same base class appears more than once in the derivation, then the derived object will have more than one subobject of that type.

This default doesn't work for a class such as `iostream`. An `iostream` object

wants to use the same buffer for both reading and writing, and it wants its condition state to reflect both input and output operations. If an `iostream` object has two copies of its `basic_ios` class, this sharing isn't possible.

In C++ we solve this kind of problem by using **virtual inheritance**. Virtual inheritance lets a class specify that it is willing to share its base class. The shared base-class subobject is called a **virtual base class**. Regardless of how often the same virtual base appears in an inheritance hierarchy, the derived object contains only one, shared subobject for that virtual base class.

## A Different Panda Class

In the past, there was some debate as to whether panda belongs to the raccoon or the bear family. To reflect this debate, we can change Panda to inherit from both Bear and Raccoon. To avoid giving Panda two `ZooAnimal` base parts, we'll define Bear and Raccoon to inherit virtually from `ZooAnimal`. Figure 18.3 illustrates our new hierarchy.

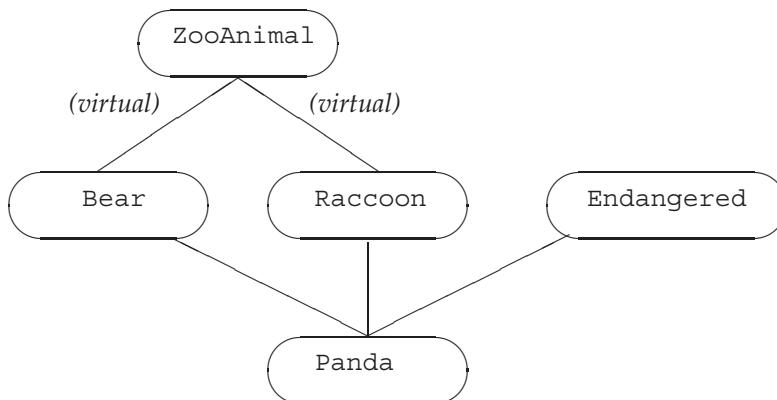
Looking at our new hierarchy, we'll notice a nonintuitive aspect of virtual inheritance. The virtual derivation has to be made before the need for it appears. For example, in our classes, the need for virtual inheritance arises only when we define Panda. However, if Bear and Raccoon had not specified `virtual` on their derivation from `ZooAnimal`, the designer of the Panda class would be out of luck.

In practice, the requirement that an intermediate base class specify its inheritance as virtual rarely causes any problems. Ordinarily, a class hierarchy that uses virtual inheritance is designed at one time either by one individual or by a single project design group. It is exceedingly rare for a class to be developed independently that needs a virtual base in one of its base classes and in which the developer of the new base class cannot change the existing hierarchy.



Virtual derivation affects the classes that subsequently derive from a class with a virtual base; it doesn't affect the derived class itself.

Figure 18.3: Virtual Inheritance Panda Hierarchy



## Using a Virtual Base Class

We specify that a base class is virtual by including the keyword `virtual` in the derivation list:

```
// the order of the keywords public and virtual is not significant
class Raccoon : public virtual ZooAnimal { /* ... */ };
class Bear : virtual public ZooAnimal { /* ... */ };
```

Here we've made `ZooAnimal` a virtual base class of both `Bear` and `Raccoon`.

The `virtual` specifier states a willingness to share a single instance of the named base class within a subsequently derived class. There are no special constraints on a class used as a virtual base class.

We do nothing special to inherit from a class that has a virtual base:

```
class Panda : public Bear,
              public Raccoon, public Endangered {
};
```

Here `Panda` inherits `ZooAnimal` through both its `Raccoon` and `Bear` base classes. However, because those classes inherited virtually from `ZooAnimal`, `Panda` has only one `ZooAnimal` base subpart.

## Normal Conversions to Base Are Supported

An object of a derived class can be manipulated (as usual) through a pointer or a reference to an accessible base-class type regardless of whether the base class is virtual. For example, all of the following `Panda` base-class conversions are legal:

```
void dance(const Bear&);
void rummage(const Raccoon&);
ostream& operator<<(ostream&, const ZooAnimal&);
Panda ying_yang;
dance(ying_yang);    // ok: passes Panda object as a Bear
rummage(ying_yang); // ok: passes Panda object as a Raccoon
cout << ying_yang;  // ok: passes Panda object as a ZooAnimal
```

## Visibility of Virtual Base-Class Members

Because there is only one shared subobject corresponding to each shared virtual base, members in that base can be accessed directly and unambiguously. Moreover, if a member from the virtual base is overridden along only one derivation path, then that overridden member can still be accessed directly. If the member is overridden by more than one base, then the derived class generally must define its own version as well.

For example, assume class `B` defines a member named `x`; class `D1` inherits virtually from `B` as does class `D2`; and class `D` inherits from `D1` and `D2`. From the scope of `D`, `x` is visible through both of its base classes. If we use `x` through a `D` object, there are three possibilities:

- If `x` is not defined in either `D1` or `D2` it will be resolved as a member in `B`; there is no ambiguity. A `D` object contains only one instance of `x`.

- If *x* is a member of *B* and also a member in one, but not both, of *D1* and *D2*, there is again no ambiguity—the version in the derived class is given precedence over the shared virtual base class, *B*.
- If *x* is defined in both *D1* and *D2*, then direct access to that member is ambiguous.

As in a nonvirtual multiple inheritance hierarchy, ambiguities of this sort are best resolved by the derived class providing its own instance of that member.

### EXERCISES SECTION 18.3.4

**Exercise 18.28:** Given the following class hierarchy, which inherited members can be accessed without qualification from within the *VMI* class? Which require qualification? Explain your reasoning.

```
struct Base {  
    void bar(int); // public by default  
protected:  
    int ival;  
};  
struct Derived1 : virtual public Base {  
    void bar(char); // public by default  
    void foo(char);  
protected:  
    char cval;  
};  
struct Derived2 : virtual public Base {  
    void foo(int); // public by default  
protected:  
    int ival;  
    char cval;  
};  
class VMI : public Derived1, public Derived2 { };
```

### 18.3.5 Constructors and Virtual Inheritance

In a virtual derivation, the virtual base is initialized by the *most derived constructor*. In our example, when we create a *Panda* object, the *Panda* constructor alone controls how the *ZooAnimal* base class is initialized.

To understand this rule, consider what would happen if normal initialization rules applied. In that case, a virtual base class might be initialized more than once. It would be initialized along each inheritance path that contains that virtual base. In our *ZooAnimal* example, if normal initialization rules applied, both *Bear* and *Raccoon* would initialize the *ZooAnimal* part of a *Panda* object.

Of course, each class in the hierarchy might at some point be the “most derived” object. As long as we can create independent objects of a type derived from

a virtual base, the constructors in that class must initialize its virtual base. For example, in our hierarchy, when a Bear (or a Raccoon) object is created, there is no further derived type involved. In this case, the Bear (or Raccoon) constructors directly initialize their ZooAnimal base as usual:

```
Bear::Bear(std::string name, bool onExhibit):
    ZooAnimal(name, onExhibit, "Bear") { }
Raccoon::Raccoon(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Raccoon") { }
```

When a Panda is created, it is the most derived type and controls initialization of the shared ZooAnimal base. Even though ZooAnimal is not a direct base of Panda, the Panda constructor initializes ZooAnimal:

```
Panda::Panda(std::string name, bool onExhibit)
    : ZooAnimal(name, onExhibit, "Panda"),
      Bear(name, onExhibit),
      Raccoon(name, onExhibit),
      Endangered(Endangered::critical),
      sleeping_flag(false) { }
```

## How a Virtually Inherited Object Is Constructed

The construction order for an object with a virtual base is slightly modified from the normal order: The virtual base subparts of the object are initialized first, using initializers provided in the constructor for the most derived class. Once the virtual base subparts of the object are constructed, the direct base subparts are constructed in the order in which they appear in the derivation list.

For example, when a Panda object is created:

- The (virtual base class) ZooAnimal part is constructed first, using the initializers specified in the Panda constructor initializer list.
- The Bear part is constructed next.
- The Raccoon part is constructed next.
- The third direct base, Endangered, is constructed next.
- Finally, the Panda part is constructed.

If the Panda constructor does not explicitly initialize the ZooAnimal base class, then the ZooAnimal default constructor is used. If ZooAnimal doesn't have a default constructor, then the code is in error.



Virtual base classes are always constructed prior to nonvirtual base classes regardless of where they appear in the inheritance hierarchy.

## Constructor and Destructor Order

A class can have more than one virtual base class. In that case, the virtual subobjects are constructed in left-to-right order as they appear in the derivation list. For

example, in the following whimsical `TeddyBear` derivation, there are two virtual base classes: `ToyAnimal`, a direct virtual base, and `ZooAnimal`, which is a virtual base class of `Bear`:

```
class Character { /* ... */ };
class BookCharacter : public Character { /* ... */ };
class ToyAnimal { /* ... */ };
class TeddyBear : public BookCharacter,
                  public Bear, public virtual ToyAnimal
{ /* ... */ };
```

The direct base classes are examined in declaration order to determine whether there are any virtual base classes. If so, the virtual bases are constructed first, followed by the nonvirtual base-class constructors in declaration order. Thus, to create a `TeddyBear`, the constructors are invoked in the following order:

```
ZooAnimal();           // Bear's virtual base class
ToyAnimal();           // direct virtual base class
Character();          // indirect base class of first nonvirtual base class
BookCharacter();       // first direct nonvirtual base class
Bear();                // second direct nonvirtual base class
TeddyBear();           // most derived class
```

The same order is used in the synthesized copy and move constructors, and members are assigned in this order in the synthesized assignment operators. As usual, an object is destroyed in reverse order from which it was constructed. The `TeddyBear` part will be destroyed first and the `ZooAnimal` part last.

### EXERCISES SECTION 18.3.5

**Exercise 18.29:** Given the following class hierarchy:

```
class Class { ... };
class Base : public Class { ... };
class D1 : virtual public Base { ... };
class D2 : virtual public Base { ... };
class MI : public D1, public D2 { ... };
class Final : public MI, public Class { ... };
```

- (a) In what order are constructors and destructors run on a `Final` object?
- (b) A `Final` object has how many `Base` parts? How many `Class` parts?
- (c) Which of the following assignments is a compile-time error?

```
Base *pb;      Class *pc;      MI *pmi;      D2 *pd2;
(a) pb = new Class;    (b) pc = new Final;
(c) pmi = pb;        (d) pd2 = pmi;
```

**Exercise 18.30:** Define a default constructor, a copy constructor, and a constructor that has an `int` parameter in `Base`. Define the same three constructors in each derived class. Each constructor should use its argument to initialize its `Base` part.

## CHAPTER SUMMARY

---

C++ is used to solve a wide range of problems—from those solvable in a few hours’ time to those that take years of development by large teams. Some features in C++ are most applicable in the context of large-scale problems: exception handling, namespaces, and multiple or virtual inheritance.

Exception handling lets us separate the error-detection part of the program from the error-handling part. When an exception is thrown, the current executing function is suspended and a search is started to find the nearest matching `catch` clause. Local variables defined inside functions that are exited while searching for a `catch` clause are destroyed as part of handling the exception.

Namespaces are a mechanism for managing large, complicated applications built from code produced by independent suppliers. A namespace is a scope in which objects, types, functions, templates, and other namespaces may be defined. The standard library is defined inside the namespace named `std`.

Conceptually, multiple inheritance is a simple notion: A derived class may inherit from more than one direct base class. The derived object consists of the derived part and a base part contributed by each of its base classes. Although conceptually simple, the details can be more complicated. In particular, inheriting from multiple base classes introduces new possibilities for name collisions and resulting ambiguous references to names from the base part of an object.

When a class inherits directly from more than one base class, it is possible that those classes may themselves share another base class. In such cases, the intermediate classes can opt to make their inheritance virtual, which states a willingness to share their virtual base class with other classes in the hierarchy that inherit virtually from that same base class. In this way there is only one copy of the shared virtual base in a subsequently derived class.

## DEFINED TERMS

---

**catch-all** A `catch` clause in which the exception declaration is `(...)`. A catch-all clause catches an exception of any type. It is typically used to catch an exception that is detected locally in order to do local cleanup. The exception is then rethrown to another part of the program to deal with the underlying cause of the problem.

**catch clause** Part of the program that handles an exception. A `catch` clause consists of the keyword `catch` followed by an exception declaration and a block of statements. The code inside a `catch` does whatever is necessary to handle an exception of the type defined in its exception declaration.

**constructor order** Under nonvirtual inheritance, base classes are constructed in the order in which they are named in the class derivation list. Under virtual inheritance, the virtual base class(es) are constructed before any other bases. They are constructed in the order in which they appear in the derivation list of the derived type. Only the most derived type may initialize a virtual base; constructor initializers for that base that appear in the intermediate base classes are ignored.

**exception declaration** `catch` clause declaration that specifies the type of exception that the `catch` can handle. The declaration acts like a parameter list, whose single parameter is initialized by the exception

object. If the exception specifier is a non-reference type, then the exception object is copied to the `catch`.

**exception handling** Language-level support for managing run-time anomalies. One independently developed section of code can detect and “raise” an exception that another independently developed part of the program can “handle.” The error-detecting part of the program throws an exception; the error-handling part handles the exception in a `catch` clause of a `try` block.

**exception object** Object used to communicate between the `throw` and `catch` sides of an exception. The object is created at the point of the `throw` and is a copy of the thrown expression. The exception object exists until the last handler for the exception completes. The type of the object is the static type of the thrown expression.

**file static** Name local to a file that is declared with the `static` keyword. In C and pre-Standard versions of C++, file statics were used to declare objects that could be used in a single file only. File statics are deprecated in C++, having been superseded by the use of unnamed namespaces.

**function try block** Used to catch exceptions from a constructor initializer. The keyword `try` appears before the colon that starts the constructor initializer list (or before the open curly of the constructor body if the initializer list is empty) and closes with one or more `catch` clauses that appear after the close curly of the constructor body.

**global namespace** The (implicit) namespace in each program that holds all global definitions.

**handler** Synonym for a `catch` clause.

**inline namespace** Members of a namespace designated as `inline` can be used as if they were members of an enclosing namespace.

**multiple inheritance** Class with more than one direct base class. The derived class inherits the members of all its base classes. A

separate access specifier may be provided for each base class.

**namespace** Mechanism for gathering all the names defined by a library or other collection of programs into a single scope. Unlike other scopes in C++, a namespace scope may be defined in several parts. The namespace may be opened and closed and reopened again in disparate parts of the program.

**namespace alias** Mechanism for defining a synonym for a given namespace:

```
namespace N1 = N;
```

defines `N1` as another name for the namespace named `N`. A namespace can have multiple aliases; the namespace name or any of its aliases may be used interchangeably.

**namespace pollution** Occurs when all the names of classes and functions are placed in the global namespace. Large programs that use code written by multiple independent parties often encounter collisions among names if these names are global.

**noexcept operator** Operator that returns a `bool` indicating whether a given expression might throw an exception. The expression is unevaluated. The result is a constant expression. Its value is `true` if the expression does not contain a `throw` and calls only functions designated as nonthrowing; otherwise the result is `false`.

**noexcept specification** Keyword used to indicate whether a function throws. When `noexcept` follows a function’s parameter list, it may be optionally followed by a parenthesized constant expression that must be convertible to `bool`. If the expression is omitted, or if it is `true`, the function throws no exceptions. An expression that is `false` or a function that has no exception specification may throw any exception.

**nonthrowing specification** An exception specification that promises that a function won't throw. If a nonthrowing functions does throw, `terminate` is called. Nonthrowing specifiers are `noexcept` without an argument or with an argument that evaluates as `true` and `throw()`.

**raise** Often used as a synonym for `throw`. C++ programmers speak of "throwing" or "raising" an exception interchangably.

**rethrow** A `throw` that does not specify an expression. A `rethrow` is valid only from inside a `catch` clause, or in a function called directly or indirectly from a `catch`. Its effect is to rethrow the exception object that it received.

**stack unwinding** The process whereby the functions are exited in the search for a `catch`. Local objects constructed before the exception are destroyed before entering the corresponding `catch`.

**terminate** Library function that is called if an exception is not caught or if an exception occurs while a handler is in process. `terminate` ends the program.

**throw e** Expression that interrupts the current execution path. Each `throw` transfers control to the nearest enclosing `catch` clause that can handle the type of exception that is thrown. The expression `e` is copied into the exception object.

**try block** Block of statements enclosed by the keyword `try` and one or more `catch` clauses. If the code inside the `try` block raises an exception and one of the `catch` clauses matches the type of the exception, then the exception is handled by that `catch`. Otherwise, the exception is passed out of the `try` to a `catch` further up the call chain.

**unnamed namespace** Namespace that is defined without a name. Names defined in an unnamed namespace may be accessed directly without use of the scope operator. Each file has its own unique unnamed namespace. Names in an unnamed namespace are not visible outside that file.

**using declaration** Mechanism to inject a single name from a namespace into the current scope:

```
using std::cout;
```

makes the name `cout` from the namespace `std` available in the current scope. The name `cout` can subsequently be used without the `std::` qualifier.

**using directive** Declaration of the form

```
using NS;
```

makes *all* the names in the namespace named `NS` available in the nearest scope containing both the `using` directive and the namespace itself.

**virtual base class** Base class that specifies `virtual` in its own derivation list. A virtual base part occurs only once in a derived object even if the same class appears as a virtual base more than once in the hierarchy. In nonvirtual inheritance a constructor may initialize only its direct base class(es). When a class is inherited virtually, that class is initialized by the most derived class, which therefore should include an initializer for all of its virtual parent(s).

**virtual inheritance** Form of multiple inheritance in which derived classes share a single copy of a base that is included in the hierarchy more than once.

**:: operator** Scope operator. Used to access names from a namespace or a class.

# C H A P T E R      19

## SPECIALIZED TOOLS AND TECHNIQUES

### CONTENTS

---

|                                                                  |     |
|------------------------------------------------------------------|-----|
| Section 19.1 Controlling Memory Allocation . . . . .             | 820 |
| Section 19.2 Run-Time Type Identification . . . . .              | 825 |
| Section 19.3 Enumerations . . . . .                              | 832 |
| Section 19.4 Pointer to Class Member . . . . .                   | 835 |
| Section 19.5 Nested Classes . . . . .                            | 843 |
| Section 19.6 <code>union</code> : A Space-Saving Class . . . . . | 847 |
| Section 19.7 Local Classes . . . . .                             | 852 |
| Section 19.8 Inherently Nonportable Features . . . . .           | 854 |
| Chapter Summary . . . . .                                        | 862 |
| Defined Terms . . . . .                                          | 862 |

The first three parts of this book discussed aspects of C++ that most C++ programmers are likely to use at some point. In addition, C++ defines some features that are more specialized. Many programmers will never (or only rarely) need to use the features presented in this chapter.

C++ is intended for use in a wide variety of applications. As a result, it contains features that are particular to some applications and that need never be used by others. In this chapter we look at some of the less-commonly used features in the language.

## 19.1 Controlling Memory Allocation

Some applications have specialized memory allocation needs that cannot be met by the standard memory management facilities. Such applications need to take over the details of how memory is allocated, for example, by arranging for new to put objects into particular kinds of memory. To do so, they can overload the new and delete operators to control memory allocation.

### 19.1.1 Overloading new and delete

Although we say that we can “overload new and delete,” overloading these operators is quite different from the way we overload other operators. In order to understand how we overload these operators, we first need to know a bit more about how new and delete expressions work.

When we use a new expression:

```
// new expressions
string *sp = new string("a value"); // allocate and initialize a string
string *arr = new string[10]; // allocate ten default initialized strings
```

three steps actually happen. First, the expression calls a library function named **operator new** (or **operator new []**). This function allocates raw, untyped memory large enough to hold an object (or an array of objects) of the specified type. Next, the compiler runs the appropriate constructor to construct the object(s) from the specified initializers. Finally, a pointer to the newly allocated and constructed object is returned.

When we use a delete expression to delete a dynamically allocated object:

```
delete sp; // destroy *sp and free the memory to which sp points
delete [] arr; // destroy the elements in the array and free the memory
```

two steps happen. First, the appropriate destructor is run on the object to which `sp` points or on the elements in the array to which `arr` points. Next, the compiler frees the memory by calling a library function named **operator delete** or **operator delete []**, respectively.

Applications that want to take control of memory allocation define their own versions of the **operator new** and **operator delete** functions. Even though the library contains definitions for these functions, we can define our own versions of them and the compiler won’t complain about duplicate definitions. Instead, the compiler will use our version in place of the one defined by the library.



When we define the global operator new and operator delete functions, we take over responsibility for all dynamic memory allocation. These functions *must* be correct: They form a vital part of all processing in the program.

Applications can define operator new and operator delete functions in the global scope and/or as member functions. When the compiler sees a new or delete expression, it looks for the corresponding operator function to call. If the object being allocated (deallocated) has class type, the compiler first looks in the scope of the class, including any base classes. If the class has a member operator new or operator delete, that function is used by the new or delete expression. Otherwise, the compiler looks for a matching function in the global scope. If the compiler finds a user-defined version, it uses that function to execute the new or delete expression. Otherwise, the standard library version is used.

We can use the scope operator to force a new or delete expression to bypass a class-specific function and use the one from the global scope. For example, `::new` will look only in the global scope for a matching operator new function. Similarly for `::delete`.

## The operator new and operator delete Interface

The library defines eight overloaded versions of operator new and delete functions. The first four support the versions of new that can throw a `bad_alloc` exception. The next four support nonthrowing versions of new:

```
// these versions might throw an exception
void *operator new(size_t);                                // allocate an object
void *operator new[](size_t);                             // allocate an array
void *operator delete(void*) noexcept;      // free an object
void *operator delete[](void*) noexcept; // free an array

// versions that promise not to throw; see § 12.1.2 (p. 460)
void *operator new(size_t, nothrow_t&) noexcept;
void *operator new[](size_t, nothrow_t&) noexcept;
void *operator delete(void*, nothrow_t&) noexcept;
void *operator delete[](void*, nothrow_t&) noexcept;
```

The type `nothrow_t` is a struct defined in the new header. This type has no members. The new header also defines a const object named `nothrow`, which users can pass to signal they want the nonthrowing version of new (§ 12.1.2, p. 460). Like destructors, an operator delete must not throw an exception (§ 18.1.1, p. 774). When we overload these operators, we must specify that they will not throw, which we do through the `noexcept` exception specifier (§ 18.1.4, p. 779).

An application can define its own version of any of these functions. If it does so, it must define these functions in the global scope or as members of a class. When defined as members of a class, these operator functions are implicitly static (§ 7.6, p. 302). There is no need to declare them static explicitly, although it is legal to do so. The member new and delete functions must be static because they are used either before the object is constructed (operator new) or after it has been

destroyed (`operator delete`). There are, therefore, no member data for these functions to manipulate.

An `operator new` or `operator new[]` function must have a return type of `void*` and its first parameter must have type `size_t`. That parameter may not have a default argument. The `operator new` function is used when we allocate an object; `operator new[]` is called when we allocate an array. When the compiler calls `operator new`, it initializes the `size_t` parameter with the number of bytes required to hold an object of the specified type; when it calls `operator new[]`, it passes the number of bytes required to store an array of the given number of elements.

When we define our own `operator new` function, we can define additional parameters. A new expression that uses such functions must use the placement form of `new` (§ 12.1.2, p. 460) to pass arguments to these additional parameters. Although generally we may define our version of `operator new` to have whatever parameters are needed, we may not define a function with the following form:

```
void *operator new(size_t, void*); // this version may not be redefined
```

This specific form is reserved for use by the library and may not be redefined.

An `operator delete` or `operator delete[]` function must have a `void` return type and a first parameter of type `void*`. Executing a `delete` expression calls the appropriate `operator` function and initializes its `void*` parameter with a pointer to the memory to free.

When `operator delete` or `operator delete[]` is defined as a class member, the function may have a second parameter of type `size_t`. If present, the additional parameter is initialized with the size in bytes of the object addressed by the first parameter. The `size_t` parameter is used when we delete objects that are part of an inheritance hierarchy. If the base class has a virtual destructor (§ 15.7.1, p. 622), then the size passed to `operator delete` will vary depending on the dynamic type of the object to which the deleted pointer points. Moreover, the version of the `operator delete` function that is run will be the one from the dynamic type of the object.

#### TERMINOLOGY: NEW EXPRESSION VERSUS OPERATOR NEW FUNCTION

The library functions `operator new` and `operator delete` are misleadingly named. Unlike other `operator` functions, such as `operator=`, these functions do not overload the `new` or `delete` expressions. In fact, we cannot redefine the behavior of the `new` and `delete` expressions.

A `new` expression always executes by calling an `operator new` function to obtain memory and then constructing an object in that memory. A `delete` expression always executes by destroying an object and then calling an `operator delete` function to free the memory used by the object.

By providing our own definitions of the `operator new` and `operator delete` functions, we can change how memory is allocated. However, we cannot change this basic meaning of the `new` and `delete` operators.

## The `malloc` and `free` Functions

If you define your own global operator `new` and operator `delete`, those functions must allocate and deallocate memory somehow. Even if you define these functions in order to use a specialized memory allocator, it can still be useful for testing purposes to be able to allocate memory similarly to how the implementation normally does so.

To this end, we can use functions named `malloc` and `free` that C++ inherits from C. These functions, are defined in `cstdlib`.

The `malloc` function takes a `size_t` that says how many bytes to allocate. It returns a pointer to the memory that it allocated, or 0 if it was unable to allocate the memory. The `free` function takes a `void*` that is a copy of a pointer that was returned from `malloc` and returns the associated memory to the system. Calling `free(0)` has no effect.

A simple way to write operator `new` and operator `delete` is as follows:

```
void *operator new(size_t size) {
    if (void *mem = malloc(size))
        return mem;
    else
        throw bad_alloc();
}
void operator delete(void *mem) noexcept { free(mem); }
```

and similarly for the other versions of operator `new` and operator `delete`.

### EXERCISES SECTION 19.1.1

**Exercise 19.1:** Write your own operator `new(size_t)` function using `malloc` and use `free` to write the operator `delete(void*)` function.

**Exercise 19.2:** By default, the allocator class uses operator `new` to obtain storage and operator `delete` to free it. Recompile and rerun your `StrVec` programs (§ 13.5, p. 526) using your versions of the functions from the previous exercise.

## 19.1.2 Placement `new` Expressions

Although the operator `new` and operator `delete` functions are intended to be used by new expressions, they are ordinary functions in the library. As a result, ordinary code can call these functions directly.

In earlier versions of the language—before the allocator (§ 12.2.2, p. 481) class was part of the library—applications that wanted to separate allocation from initialization did so by calling operator `new` and operator `delete`. These functions behave analogously to the `allocate` and `deallocate` members of `allocator`. Like those members, operator `new` and operator `delete` functions allocate and deallocate memory but do not construct or destroy objects.

Differently from an allocator, there is no construct function we can call to construct objects in memory allocated by operator new. Instead, we use the **placement new** form of new (§ 12.1.2, p. 460) to construct an object. As we've seen, this form of new provides extra information to the allocation function. We can use placement new to pass an address, in which case the placement new expression has the form

```
new (place_address) type
new (place_address) type (initializers)
new (place_address) type [size]
new (place_address) type [size] { braced initializer list }
```

where *place\_address* must be a pointer and the *initializers* provide (a possibly empty) comma-separated list of initializers to use to construct the newly allocated object.

When called with an address and no other arguments, placement new uses operator new(size\_t, void\*) to "allocate" its memory. This is the version of operator new that we are not allowed to redefine (§ 19.1.1, p. 822). This function does *not* allocate any memory; it simply returns its pointer argument. The overall new expression then finishes its work by initializing an object at the given address. In effect, placement new allows us to construct an object at a specific, preallocated memory address.



When passed a single argument that is a pointer, a placement new expression constructs an object but does not allocate memory.

Although in many ways using placement new is analogous to the construct member of an allocator, there is one important difference. The pointer that we pass to construct must point to space allocated by the same allocator object. The pointer that we pass to placement new need not point to memory allocated by operator new. Indeed, as we'll see in § 19.6 (p. 851), the pointer passed to a placement new expression need not even refer to dynamic memory.

## Explicit Destructor Invocation

Just as placement new is analogous to using allocate, an explicit call to a destructor is analogous to calling destroy. We call a destructor the same way we call any other member function on an object or through a pointer or reference to an object:

```
string *sp = new string("a value"); // allocate and initialize a string
sp->~string();
```

Here we invoke a destructor directly. The arrow operator dereferences the pointer sp to obtain the object to which sp points. We then call the destructor, which is the name of the type preceded by a tilde (~).

Like calling destroy, calling a destructor cleans up the given object but does not free the space in which that object resides. We can reuse the space if desired.



Calling a destructor destroys an object but does not free the memory.

## 19.2 Run-Time Type Identification

Run-time type identification (RTTI) is provided through two operators:

- The `typeid` operator, which returns the type of a given expression
- The `dynamic_cast` operator, which safely converts a pointer or reference to a base type into a pointer or reference to a derived type

When applied to pointers or references to types that have virtual functions, these operators use the dynamic type (§ 15.2.3, p. 601) of the object to which the pointer or reference is bound.

These operators are useful when we have a derived operation that we want to perform through a pointer or reference to a base-class object and it is not possible to make that operation a virtual function. Ordinarily, we should use virtual functions if we can. When the operation is virtual, the compiler automatically selects the right function according to the dynamic type of the object.

However, it is not always possible to define a virtual. If we cannot use a virtual, we can use one of the RTTI operators. On the other hand, using these operators is more error-prone than using virtual member functions: The programmer must *know* to which type the object should be cast and must check that the cast was performed successfully.



RTTI should be used with caution. When possible, it is better to define a virtual function rather than to take over managing the types directly.

### 19.2.1 The `dynamic_cast` Operator

A `dynamic_cast` has the following form:

```
dynamic_cast<type*>(e)
dynamic_cast<type&>(e)
dynamic_cast<type&&>(e)
```

where *type* must be a class type and (ordinarily) names a class that has virtual functions. In the first case, *e* must be a valid pointer (§ 2.3.2, p. 52); in the second, *e* must be an lvalue; and in the third, *e* must not be an lvalue.

In all cases, the type of *e* must be either a class type that is publicly derived from the target *type*, a public base class of the target *type*, or the same as the target *type*. If *e* has one of these types, then the cast will succeed. Otherwise, the cast fails. If a `dynamic_cast` to a pointer type fails, the result is 0. If a `dynamic_cast` to a reference type fails, the operator throws an exception of type `bad_cast`.

#### Pointer-Type `dynamic_casts`

As a simple example, assume that `Base` is a class with at least one virtual function and that class `Derived` is publicly derived from `Base`. If we have a pointer to `Base` named `bp`, we can cast it, at run time, to a pointer to `Derived` as follows:

```

if (Derived *dp = dynamic_cast<Derived*>(bp) )
{
    // use the Derived object to which dp points
} else { // bp points at a Base object
    // use the Base object to which bp points
}

```

If `bp` points to a `Derived` object, then the cast will initialize `dp` to point to the `Derived` object to which `bp` points. In this case, it is safe for the code inside the `if` to use `Derived` operations. Otherwise, the result of the cast is 0. If `dp` is 0, the condition in the `if` fails. In this case, the `else` clause does processing appropriate to `Base` instead.



We can do a `dynamic_cast` on a null pointer; the result is a null pointer of the requested type.

It is worth noting that we defined `dp` inside the condition. By defining the variable in a condition, we do the cast and corresponding check as a single operation. Moreover, the pointer `dp` is not accessible outside the `if`. If the cast fails, then the unbound pointer is not available for use in subsequent code where we might forget to check whether the cast succeeded.



Performing a `dynamic_cast` in a condition ensures that the cast and test of its result are done in a single expression.

## Reference-Type `dynamic_cast`s

A `dynamic_cast` to a reference type differs from a `dynamic_cast` to a pointer type in how it signals that an error occurred. Because there is no such thing as a null reference, it is not possible to use the same error-reporting strategy for references that is used for pointers. When a cast to a reference type fails, the cast throws a `std::bad_cast` exception, which is defined in the `typeinfo` library header.

We can rewrite the previous example to use references as follows:

```

void f(const Base &b)
{
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
        // use the Derived object to which b referred
    } catch (bad_cast) {
        // handle the fact that the cast failed
    }
}

```

### 19.2.2 The `typeid` Operator

The second operator provided for RTTI is the **`typeid` operator**. The `typeid` operator allows a program to ask of an expression: What type is your object?

**EXERCISES SECTION 19.2.1**

**Exercise 19.3:** Given the following class hierarchy in which each class defines a public default constructor and virtual destructor:

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };
class D : public B, public A { /* ... */ };
```

which, if any, of the following dynamic\_casts fail?

- (a) A \*pa = new C;  
    B \*pb = dynamic\_cast< B\*>(pa);
- (b) B \*pb = new B;  
    C \*pc = dynamic\_cast< C\*>(pb);
- (c) A \*pa = new D;  
    B \*pb = dynamic\_cast< B\*>(pa);

**Exercise 19.4:** Using the classes defined in the first exercise, rewrite the following code to convert the expression \*pa to the type C&:

```
if (C *pc = dynamic_cast< C*>(pa))
    // use C's members
} else {
    // use A's members
}
```

**Exercise 19.5:** When should you use a dynamic\_cast instead of a virtual function?

A typeid expression has the form typeid(e) where e is any expression or a type name. The result of a typeid operation is a reference to a const object of a library type named type\_info, or a type publicly derived from type\_info. § 19.2.4 (p. 831) covers this type in more detail. The type\_info class is defined in the typeinfo header.

The typeid operator can be used with expressions of any type. As usual, top-level const (§ 2.4.3, p. 63) is ignored, and if the expression is a reference, typeid returns the type to which the reference refers. When applied to an array or function, however, the standard conversion to pointer (§ 4.11.2, p. 161) is not done. That is, if we take typeid(a) and a is an array, the result describes an array type, not a pointer type.

When the operand is not of class type or is a class without virtual functions, then the typeid operator indicates the static type of the operand. When the operand is an lvalue of a class type that defines at least one virtual function, then the type is evaluated at run time.

## Using the typeid Operator

Ordinarily, we use typeid to compare the types of two expressions or to compare the type of an expression to a specified type:

```

Derived *dp = new Derived;
Base *bp = dp; // both pointers point to a Derived object
// compare the type of two objects at run time
if (typeid(*bp) == typeid(*dp)) {
    // bp and dp point to objects of the same type
}
// test whether the run-time type is a specific type
if (typeid(*bp) == typeid(Derived)) {
    // bp actually points to a Derived
}

```

In the first `if`, we compare the dynamic types of the objects to which `bp` and `dp` point. If both point to the same type, then the condition succeeds. Similarly, the second `if` succeeds if `bp` currently points to a `Derived` object.

Note that the operands to the `typeid` are objects—we used `*bp`, not `bp`:

```

// test always fails: the type of bp is pointer to Base
if (typeid(bp) == typeid(Derived)) {
    // code never executed
}

```

This condition compares the type `Base*` to type `Derived`. Although the pointer *points* at an object of class type that has virtual functions, the pointer *itself* is not a class-type object. The type `Base*` can be, and is, evaluated at compile time. That type is unequal to `Derived`, so the condition will always fail *regardless of the type of the object to which bp points*.



The `typeid` of a pointer (as opposed to the object to which the pointer points) returns the static, compile-time type of the pointer.

Whether `typeid` requires a run-time check determines whether the expression is evaluated. The compiler evaluates the expression only if the type has virtual functions. If the type has no `virtuals`, then `typeid` returns the static type of the expression; the compiler knows the static type without evaluating the expression.

If the dynamic type of the expression might differ from the static type, then the expression must be evaluated (at run time) to determine the resulting type. The distinction matters when we evaluate `typeid(*p)`. If `p` is a pointer to a type that does not have virtual functions, then `p` does not need to be a valid pointer. Otherwise, `*p` is evaluated at run time, in which case `p` must be a valid pointer. If `p` is a null pointer, then `typeid(*p)` throws a `bad_typeid` exception.

### 19.2.3 Using RTTI

As an example of when RTTI might be useful, consider a class hierarchy for which we'd like to implement the equality operator (§ 14.3.1, p. 561). Two objects are equal if they have the same type and same value for a given set of their data members. Each derived type may add its own data, which we will want to include when we test for equality.

**EXERCISES SECTION 19.2.2**

**Exercise 19.6:** Write an expression to dynamically cast a pointer to a `Query_base` to a pointer to an `AndQuery` (§ 15.9.1, p. 636). Test the cast by using objects of `AndQuery` and of another query type. Print a statement indicating whether the cast works and be sure that the output matches your expectations.

**Exercise 19.7:** Write the same cast, but cast a `Query_base` object to a reference to `AndQuery`. Repeat the test to ensure that your cast works correctly.

**Exercise 19.8:** Write a `typeid` expression to see whether two `Query_base` pointers point to the same type. Now check whether that type is an `AndQuery`.

We might think we could solve this problem by defining a set of virtual functions that would perform the equality test at each level in the hierarchy. Given those virtuals, we would define a single equality operator that operates on references to the base type. That operator could delegate its work to a virtual `equal` operation that would do the real work.

Unfortunately, this strategy doesn't quite work. Virtual functions must have the same parameter type(s) in both the base and derived classes (§ 15.3, p. 605). If we wanted to define a virtual `equal` function, that function must have a parameter that is a reference to the base class. If the parameter is a reference to base, the `equal` function could use only members from the base class. `equal` would have no way to compare members that are in the derived class but not in the base.

We can write our equality operation by realizing that the equality operator ought to return `false` if we attempt to compare objects of differing type. For example, if we try to compare a object of the base-class type with an object of a derived type, the `==` operator should return `false`.

Given this observation, we can now see that we can use RTTI to solve our problem. We'll define an equality operator whose parameters are references to the base-class type. The equality operator will use `typeid` to verify that the operands have the same type. If the operands differ, the `==` will return `false`. Otherwise, it will call a virtual `equal` function. Each class will define `equal` to compare the data elements of its own type. These operators will take a `Base&` parameter but will cast the operand to its own type before doing the comparison.

## The Class Hierarchy

To make the concept a bit more concrete, we'll define the following classes:

```
class Base {  
    friend bool operator==(const Base&, const Base&);  
public:  
    // interface members for Base  
protected:  
    virtual bool equal(const Base&) const;  
    // data and other implementation members of Base  
};
```

```

class Derived: public Base {
public:
    // other interface members for Derived
protected:
    bool equal(const Base&) const;
    // data and other implementation members of Derived
};

```

## A Type-Sensitive Equality Operator

Next let's look at how we might define the overall equality operator:

```

bool operator==(const Base &lhs, const Base &rhs)
{
    // returns false if typeids are different; otherwise makes a virtual call to equal
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}

```

This operator returns `false` if the operands are different types. If they are the same type, then it delegates the real work of comparing the operands to the (virtual) `equal` function. If the operands are `Base` objects, then `Base::equal` will be called. If they are `Derived` objects, `Derived::equal` is called.

## The Virtual `equal` Functions

Each class in the hierarchy must define its own version of `equal`. All of the functions in the derived classes will start the same way: They'll cast their argument to the type of the class itself:

```

bool Derived::equal(const Base &rhs) const
{
    // we know the types are equal, so the cast won't throw
    auto r = dynamic_cast<const Derived&>(rhs);
    // do the work to compare two Derived objects and return the result
}

```

The cast should always succeed—after all, the function is called from the equality operator only after testing that the two operands are the same type. However, the cast is necessary so that the function can access the derived members of the right-hand operand.

## The Base-Class `equal` Function

This operation is a bit simpler than the others:

```

bool Base::equal(const Base &rhs) const
{
    // do whatever is required to compare to Base objects
}

```

There is no need to cast the parameter before using it. Both `*this` and the parameter are `Base` objects, so all the operations available for this object are also defined for the parameter type.

### 19.2.4 The `type_info` Class

The exact definition of the `type_info` class varies by compiler. However, the standard guarantees that the class will be defined in the `typeinfo` header and that the class will provide at least the operations listed in Table 19.1.

The class also provides a public virtual destructor, because it is intended to serve as a base class. When a compiler wants to provide additional type information, it normally does so in a class derived from `type_info`.

**Table 19.1: Operations on `type_info`**

|                            |                                                                                                                                                                      |
|----------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>t1 == t2</code>      | Returns <code>true</code> if the <code>type_info</code> objects <code>t1</code> and <code>t2</code> refer to the same type, <code>false</code> otherwise.            |
| <code>t1 != t2</code>      | Returns <code>true</code> if the <code>type_info</code> objects <code>t1</code> and <code>t2</code> refer to different types, <code>false</code> otherwise.          |
| <code>t.name()</code>      | Returns a C-style character string that is a printable version of the type name. Type names are generated in a system-dependent way.                                 |
| <code>t1.before(t2)</code> | Returns a <code>bool</code> that indicates whether <code>t1</code> comes before <code>t2</code> . The ordering imposed by <code>before</code> is compiler dependent. |

There is no `type_info` default constructor, and the copy and move constructors and the assignment operators are all defined as deleted (§ 13.1.6, p. 507). Therefore, we cannot define, copy, or assign objects of type `type_info`. The only way to create a `type_info` object is through the `typeid` operator.

The `name` member function returns a C-style character string for the name of the type represented by the `type_info` object. The value used for a given type depends on the compiler and in particular is not required to match the type names as used in a program. The only guarantee we have about the return from `name` is that it returns a unique string for each type. For example:

```
int arr[10];
Derived d;
Base *p = &d;

cout << typeid(42).name() << ", "
    << typeid(arr).name() << ", "
    << typeid(Sales_data).name() << ", "
    << typeid(std::string).name() << ", "
    << typeid(p).name() << ", "
    << typeid(*p).name() << endl;
```

This program, when executed on our machine, generates the following output:

```
i, A10_i, 10Sales_data, Ss, P4Base, 7Derived
```



The `type_info` class varies by compiler. Some compilers provide additional member functions that provide additional information about types used in a program. You should consult the reference manual for your compiler to understand the exact `type_info` support provided.

## EXERCISES SECTION 19.2.4

**Exercise 19.9:** Write a program similar to the last one in this section to print the names your compiler uses for common type names. If your compiler gives output similar to ours, write a function that will translate those strings to more human-friendly form.

**Exercise 19.10:** Given the following class hierarchy in which each class defines a public default constructor and virtual destructor, which type name do the following statements print?

```
class A { /* ... */ };
class B : public A { /* ... */ };
class C : public B { /* ... */ };

(a) A *pa = new C;
    cout << typeid(pa).name() << endl;
(b) C cobj;
    A& ra = cobj;
    cout << typeid(&ra).name() << endl;
(c) B *px = new B;
    A& ra = *px;
    cout << typeid(ra).name() << endl;
```

## 19.3 Enumerations

Enumerations let us group together sets of integral constants. Like classes, each enumeration defines a new type. Enumerations are literal types (§ 7.5.6, p. 299).

C++ has two kinds of enumerations: scoped and unscoped. The new standard introduced **scoped enumerations**. We define a scoped enumeration using the keywords `enum class` (or, equivalently, `enum struct`), followed by the enumeration name and a comma-separated list of **enumerators** enclosed in curly braces. A semicolon follows the close curly:



```
enum class open_modes {input, output, append};
```

Here we defined an enumeration type named `open_modes` that has three enumerators: `input`, `output`, and `append`.

We define an **unscoped enumeration** by omitting the `class` (or `struct`) keyword. The enumeration name is optional in an unscoped enum:

```
enum color {red, yellow, green};           // unscoped enumeration
// unnamed, unscoped enum
enum {floatPrec = 6, doublePrec = 10, double_doublePrec = 10};
```

If the `enum` is unnamed, we may define objects of that type only as part of the `enum` definition. As with a class definition, we can provide a comma-separated list of declarators between the close curly and the semicolon that ends the `enum` definition (§ 2.6.1, p. 73).

## Enumerators

The names of the enumerators in a scoped enumeration follow normal scoping rules and are inaccessible outside the scope of the enumeration. The enumerator names in an unscoped enumeration are placed into the same scope as the enumeration itself:

```
enum color {red, yellow, green};      // unscoped enumeration
enum stoplight {red, yellow, green};  // error: redefines enumerators
enum class peppers {red, yellow, green}; // ok: enumerators are hidden
color eyes = green; // ok: enumerators are in scope for an unscoped enumeration
peppers p = green; // error: enumerators from peppers are not in scope
                      //           color::green is in scope but has the wrong type
color hair = color::red; // ok: we can explicitly access the enumerators
peppers p2 = peppers::red; // ok: using red from peppers
```

By default, enumerator values start at 0 and each enumerator has a value 1 greater than the preceding one. However, we can also supply initializers for one or more enumerators:

```
enum class intTypes {
    charTyp = 8, shortTyp = 16, intTyp = 16,
    longTyp = 32, long_longTyp = 64
};
```

As we see with the enumerators for `intTyp` and `shortTyp`, an enumerator value need not be unique. When we omit an initializer, the enumerator has a value 1 greater than the preceding enumerator.

Enumerators are `const` and, if initialized, their initializers must be constant expressions (§ 2.4.4, p. 65). Consequently, each enumerator is itself a constant expression. Because the enumerators are constant expressions, we can use them where a constant expression is required. For example, we can define `constexpr` variables of enumeration type:

```
constexpr intTypes charbits = intTypes::charTyp;
```

Similarly, we can use an `enum` as the expression in a `switch` statement and use the value of its enumerators as the case labels (§ 5.3.2, p. 178). For the same reason, we can also use an enumeration type as a nontype template parameter (§ 16.1.1, p. 654) and can initialize class `static` data members of enumeration type inside the class definition (§ 7.6, p. 302).

## Like Classes, Enumerations Define New Types

So long as the `enum` is named, we can define and initialize objects of that type. An `enum` object may be initialized or assigned only by one of its enumerators or by another object of the same `enum` type:

```
open_modes om = 2;      // error: 2 is not of type open_modes
om = open_modes::input; // ok: input is an enumerator of open_modes
```

Objects or enumerators of an unscoped enumeration type are automatically converted to an integral type. As a result, they can be used where an integral value is required:

```
int i = color::red;    // ok: unscoped enumerator implicitly converted to int
int j = peppers::red; // error: scoped enumerations are not implicitly converted
```

## Specifying the Size of an enum

Although each enum defines a unique type, it is represented by one of the built-in integral types. Under the new standard, we may specify that type by following the enum name with a colon and the name of the type we want to use:

```
C++ 11
enum intValues : unsigned long long {
    charTyp = 255, shortTyp = 65535, intTyp = 65535,
    longTyp = 4294967295UL,
    long_longTyp = 18446744073709551615ULL
};
```

If we do not specify the underlying type, then by default scoped enums have `int` as the underlying type. There is no default for unscoped enums; all we know is that the underlying type is large enough to hold the enumerator values. When the underlying type is specified (including implicitly specified for a scoped enum), it is an error for an enumerator to have a value that is too large to fit in that type.

Being able to specify the underlying type of an enum lets us control the type used across different implementations. We can be confident that our program compiled under one implementation will generate the same code when we compile it on another.

## Forward Declarations for Enumerations

**C++ 11** Under the new standard, we can forward declare an enum. An enum forward declaration must specify (implicitly or explicitly) the underlying size of the enum:

```
// forward declaration of unscoped enum named intValues
enum intValues : unsigned long long; // unscoped, must specify a type
enum class open_modes; // scoped enums can use int by default
```

Because there is no default size for an unscoped enum, every declaration must include the size of that enum. We can declare a scoped enum without specifying a size, in which case the size is implicitly defined as `int`.

As with any declaration, all the declarations and the definition of a given enum must match one another. In the case of enums, this requirement means that the size of the enum must be the same across all declarations and the enum definition. Moreover, we cannot declare a name as an unscoped enum in one context and redeclare it as a scoped enum later:

```
// error: declarations and definition must agree whether the enum is scoped or unscoped
enum class intValues;
enum intValues; // error: intValues previously declared as scoped enum
enum intValues : long; // error: intValues previously declared as int
```

## Parameter Matching and Enumerations

Because an object of enum type may be initialized only by another object of that enum type or by one of its enumerators (§ 19.3, p. 833), an integral value that happens to have the same value as an enumerator cannot be used to call a function expecting an enum argument:

```
// unscoped enumeration; the underlying type is machine dependent
enum Tokens { INLINE = 128, VIRTUAL = 129 };

void ff(Tokens);
void ff(int);

int main() {
    Tokens curTok = INLINE;
    ff(128);      // exactly matches ff(int)
    ff(INLINE);   // exactly matches ff(Tokens)
    ff(curTok);   // exactly matches ff(Tokens)
    return 0;
}
```

Although we cannot pass an integral value to an enum parameter, we can pass an object or enumerator of an unscoped enumeration to a parameter of integral type. When we do so, the enum value promotes to int or to a larger integral type. The actual promotion type depends on the underlying type of the enumeration:

```
void newf(unsigned char);
void newf(int);
unsigned char uc = VIRTUAL;
newf(VIRTUAL); // calls newf(int)
newf(uc);       // calls newf(unsigned char)
```

The enum Tokens has only two enumerators, the larger of which has the value 129. That value can be represented by the type unsigned char, and many compilers will use unsigned char as the underlying type for Tokens. Regardless of its underlying type, objects and the enumerators of Tokens are promoted to int. Enumerators and values of an enum type are not promoted to unsigned char, even if the values of the enumerators would fit.

## 19.4 Pointer to Class Member

A **pointer to member** is a pointer that can point to a nonstatic member of a class. Normally a pointer points to an object, but a pointer to member identifies a member of a class, not an object of that class. static class members are not part of any object, so no special syntax is needed to point to a static member. Pointers to static members are ordinary pointers.

The type of a pointer to member embodies both the type of a class and the type of a member of that class. We initialize such pointers to point to a specific member of a class without identifying an object to which that member belongs. When we use a pointer to member, we supply the object whose member we wish to use.

To explain pointers to members, we'll use a version of the `Screen` class from § 7.3.1 (p. 271):

```
class Screen {
public:
    typedef std::string::size_type pos;
    char get_cursor() const { return contents[cursor]; }
    char get() const;
    char get(pos ht, pos wd) const;
private:
    std::string contents;
    pos cursor;
    pos height, width;
};
```

### 19.4.1 Pointers to Data Members

As with any pointer, we declare a pointer to member using a `*` to indicate that the name we're declaring is a pointer. Unlike ordinary pointers, a pointer to member also incorporates the class that contains the member. Hence, we must precede the `*` with *classname*`::` to indicate that the pointer we are defining can point to a member of *classname*. For example:

```
// pdata can point to a string member of a const (or nonconst) Screen object
const string Screen::*pdata;
```

declares that `pdata` is a “pointer to a member of class `Screen` that has type `const string`.” The data members in a `const` object are themselves `const`. By making our pointer a pointer to `const string` member, we say that we can use `pdata` to point to a member of any `Screen` object, `const` or not. In exchange we can use `pdata` to read, but not write to, the member to which it points.

When we initialize (or assign to) a pointer to member, we say to which member it points. For example, we can make `pdata` point to the `contents` member of an unspecified `Screen` object as follows:

```
pdata = &Screen::contents;
```

Here, we apply the address-of operator not to an object in memory but to a member of the class `Screen`.

Of course, under the new standard, the easiest way to declare a pointer to member is to use `auto` or `decltype`:

```
auto pdata = &Screen::contents;
```

### Using a Pointer to Data Member

It is essential to understand that when we initialize or assign a pointer to member, that pointer does not yet point to any data. It identifies a specific member but not the object that contains that member. We supply the object when we dereference the pointer to member.

Analogous to the member access operators, `.` and `->`, there are two pointer-to-member access operators, `.*` and `->*`, that let us supply an object and dereference the pointer to fetch a member of that object:

```
Screen myScreen, *pScreen = &myScreen;
// .* dereferences pdata to fetch the contents member from the object myScreen
auto s = myScreen.*pdata;
// ->* dereferences pdata to fetch contents from the object to which pScreen points
s = pScreen->*pdata;
```

Conceptually, these operators perform two actions: They dereference the pointer to member to get the member that we want; then, like the member access operators, they fetch that member from an object (`.*`) or through a pointer (`->*`).

## A Function Returning a Pointer to Data Member

Normal access controls apply to pointers to members. For example, the `contents` member of `Screen` is `private`. As a result, the use of `pdata` above must have been inside a member or friend of class `Screen` or it would be an error.

Because data members are typically `private`, we normally can't get a pointer to data member directly. Instead, if a class like `Screen` wanted to allow access to its `contents` member, it would define a function to return a pointer to that member:

```
class Screen {
public:
    // data is a static member that returns a pointer to member
    static const std::string Screen::*data()
        { return &Screen::contents; }
    // other members as before
};
```

Here we've added a `static` member to class `Screen` that returns a pointer to the `contents` member of a `Screen`. The return type of this function is the same type as our original `pdata` pointer. Reading the return type from right to left, we see that `data` returns a pointer to a member of class `Screen` that is a `string` that is `const`. The body of the function applies the address-of operator to the `contents` member, so the function returns a pointer to the `contents` member of `Screen`.

When we call `data`, we get a pointer to member:

```
// data() returns a pointer to the contents member of class Screen
const string Screen::*pdata = Screen::data();
```

As before, `pdata` points to a member of class `Screen` but not to actual data. To use `pdata`, we must bind it to an object of type `Screen`:

```
// fetch the contents of the object named myScreen
auto s = myScreen.*pdata;
```

## EXERCISES SECTION 19.4.1

**Exercise 19.11:** What is the difference between an ordinary data pointer and a pointer to a data member?

**Exercise 19.12:** Define a pointer to member that can point to the cursor member of class Screen. Fetch the value of Screen::cursor through that pointer.

**Exercise 19.13:** Define the type that can represent a pointer to the bookNo member of the Sales\_data class.

## 19.4.2 Pointers to Member Functions

We can also define a pointer that can point to a member function of a class. As with pointers to data members, the easiest way to form a pointer to member function is to use auto to deduce the type for us:

```
// pmf is a pointer that can point to a Screen member function that is const
// that returns a char and takes no arguments
auto pmf = &Screen::get_cursor;
```

Like a pointer to data member, a pointer to a function member is declared using `classname::*`. Like any other function pointer (§ 6.7, p. 247), a pointer to member function specifies the return type and parameter list of the type of function to which this pointer can point. If the member function is a const member (§ 7.1.2, p. 258) or a reference member (§ 13.6.3, p. 546), we must include the `const` or `reference` qualifier as well.

As with normal function pointers, if the member is overloaded, we must distinguish which function we want by declaring the type explicitly (§ 6.7, p. 248). For example, we can declare a pointer to the two-parameter version of get as

```
char (Screen::*pmf2)(Screen::pos, Screen::pos) const;
pmf2 = &Screen::get;
```

The parentheses around `Screen::*` in this declaration are essential due to precedence. Without the parentheses, the compiler treats the following as an (invalid) function declaration:

```
// error: nonmember function p cannot have a const qualifier
char Screen::*p(Screen::pos, Screen::pos) const;
```

This declaration tries to define an ordinary function named `p` that returns a pointer to a member of class Screen that has type `char`. Because it declares an ordinary function, the declaration can't be followed by a `const` qualifier.

Unlike ordinary function pointers, there is no automatic conversion between a member function and a pointer to that member:

```
// pmf points to a Screen member that takes no arguments and returns char
pmf = &Screen::get; // must explicitly use the address-of operator
pmf = Screen::get; // error: no conversion to pointer for member functions
```

## Using a Pointer to Member Function

As when we use a pointer to a data member, we use the `.*` or `->*` operators to call a member function through a pointer to member:

```
Screen myScreen, *pScreen = &myScreen;
// call the function to which pmf points on the object to which pScreen points
char c1 = (pScreen->*pmf) ();
// passes the arguments 0, 0 to the two-parameter version of get on the object myScreen
char c2 = (myScreen.*pmf2) (0, 0);
```

The calls `(myScreen->*pmf) ()` and `(pScreen.*pmf2) (0, 0)` require the parentheses because the precedence of the call operator is higher than the precedence of the pointer-to-member operators.

Without the parentheses,

```
myScreen.*pmf ()
```

would be interpreted to mean

```
myScreen.* (pmf ())
```

This code says to call the function named `pmf` and use its return value as the operand of the pointer-to-member operator `(.* )`. However, `pmf` is not a function, so this code is in error.



Because of the relative precedence of the call operator, declarations of pointers to member functions and calls through such pointers must use parentheses: `(C::*p) (parms)` and `(obj.*p) (args)`.

## Using Type Aliases for Member Pointers

Type aliases or `typedefs` (§ 2.5.1, p. 67) make pointers to members considerably easier to read. For example, the following type alias defines `Action` as an alternative name for the type of the two-parameter version of `get`:

```
// Action is a type that can point to a member function of Screen
// that returns a char and takes two pos arguments
using Action =
char (Screen::*) (Screen::pos, Screen::pos) const;
```

`Action` is another name for the type “pointer to a `const` member function of class `Screen` taking two parameters of type `pos` and returning `char`.” Using this alias, we can simplify the definition of a pointer to `get` as follows:

```
Action get = &Screen::get; // get points to the get member of Screen
```

As with any other function pointer, we can use a pointer-to-member function type as the return type or as a parameter type in a function. Like any other parameter, a pointer-to-member parameter can have a default argument:

```
// action takes a reference to a Screen and a pointer to a Screen member function
Screen& action(Screen&, Action = &Screen::get);
```

`action` is a function taking two parameters, which are a reference to a `Screen` object and a pointer to a member function of class `Screen` that takes two pos parameters and returns a `char`. We can call `action` by passing it either a pointer or the address of an appropriate member function in `Screen`:

```
Screen myScreen;
// equivalent calls:
action(myScreen);           // uses the default argument
action(myScreen, get);      // uses the variable get that we previously defined
action(myScreen, &Screen::get); // passes the address explicitly
```



Type aliases make code that uses pointers to members much easier to read and write.

## Pointer-to-Member Function Tables

One common use for function pointers and for pointers to member functions is to store them in a function table (§ 14.8.3, p. 577). For a class that has several members of the same type, such a table can be used to select one from the set of these members. Let's assume that our `Screen` class is extended to contain several member functions, each of which moves the cursor in a particular direction:

```
class Screen {
public:
    // other interface and implementation members as before
    Screen& home();          // cursor movement functions
    Screen& forward();
    Screen& back();
    Screen& up();
    Screen& down();
};
```

Each of these new functions takes no parameters and returns a reference to the `Screen` on which it was invoked.

We might want to define a `move` function that can call any one of these functions and perform the indicated action. To support this new function, we'll add a static member to `Screen` that will be an array of pointers to the cursor movement functions:

```
class Screen {
public:
    // other interface and implementation members as before
    // Action is a pointer that can be assigned any of the cursor movement members
    using Action = Screen& (Screen::*)();
    // specify which direction to move; enum see § 19.3 (p. 832)
    enum Directions { HOME, FORWARD, BACK, UP, DOWN };
    Screen& move(Directions);
private:
    static Action Menu[];        // function table
};
```

The array named `Menu` will hold pointers to each of the cursor movement functions. Those functions will be stored at the offsets corresponding to the enumerators in `Directions`. The `move` function takes an enumerator and calls the appropriate function:

```
Screen& Screen::move(Directions cm)
{
    // run the element indexed by cm on this object
    return (this->*Menu[cm])(); // Menu [cm] points to a member function
}
```

The call inside `move` is evaluated as follows: The `Menu` element indexed by `cm` is fetched. That element is a pointer to a member function of the `Screen` class. We call the member function to which that element points on behalf of the object to which `this` points.

When we call `move`, we pass it an enumerator that indicates which direction to move the cursor:

```
Screen myScreen;
myScreen.move(Screen::HOME); // invokes myScreen.home
myScreen.move(Screen::DOWN); // invokes myScreen.down
```

What's left is to define and initialize the table itself:

```
Screen::Action Screen::Menu[] = { &Screen::home,
                                  &Screen::forward,
                                  &Screen::back,
                                  &Screen::up,
                                  &Screen::down,
};
```

### EXERCISES SECTION 19.4.2

**Exercise 19.14:** Is the following code legal? If so, what does it do? If not, why?

```
auto pmf = &Screen::get_cursor;
pmf = &Screen::get;
```

**Exercise 19.15:** What is the difference between an ordinary function pointer and a pointer to a member function?

**Exercise 19.16:** Write a type alias that is a synonym for a pointer that can point to the `avg_price` member of `Sales_data`.

**Exercise 19.17:** Define a type alias for each distinct `Screen` member function type.

### 19.4.3 Using Member Functions as Callable Objects

As we've seen, to make a call through a pointer to member function, we must use the `.*` or `->*` operators to bind the pointer to a specific object. As a result,

unlike ordinary function pointers, a pointer to member is *not* a callable object; these pointers do not support the function-call operator (§ 10.3.2, p. 388).

Because a pointer to member is not a callable object, we cannot directly pass a pointer to a member function to an algorithm. As an example, if we wanted to find the first empty string in a vector of strings, the obvious call won't work:

```
auto fp = &string::empty; // fp points to the string empty function
// error: must use .* or ->* to call a pointer to member
find_if(svec.begin(), svec.end(), fp);
```

The `find_if` algorithm expects a callable object, but we've supplied `fp`, which is a pointer to a member function. This call won't compile, because the code inside `find_if` executes a statement something like

```
// check whether the given predicate applied to the current element yields true
if (fp(*it)) // error: must use ->* to call through a pointer to member
```

which attempts to call the object it was passed.

## Using function to Generate a Callable

One way to obtain a callable from a pointer to member function is by using the library function template (§ 14.8.3, p. 577):

```
function<bool (const string&)> fcn = &string::empty;
find_if(svec.begin(), svec.end(), fcn);
```

Here we tell `function` that `empty` is a function that can be called with a `string` and returns a `bool`. Ordinarily, the object on which a member function executes is passed to the implicit `this` parameter. When we want to use `function` to generate a callable for a member function, we have to "translate" the code to make that implicit parameter explicit.

When a `function` object holds a pointer to a member function, the `function` class knows that it must use the appropriate pointer-to-member operator to make the call. That is, we can imagine that `find_if` will have code something like

```
// assuming it is the iterator inside find_if, so *it is an object in the given range
if (fcn(*it)) // assuming fcn is the name of the callable inside find_if
```

which `function` will execute using the proper pointer-to-member operator. In essence, the `function` class will transform this call into something like

```
// assuming it is the iterator inside find_if, so *it is an object in the given range
if (((*it).*p)()) // assuming p is the pointer to member function inside fcn
```

When we define a `function` object, we must specify the function type that is the signature of the callable objects that object can represent. When the callable is a member function, the signature's first parameter must represent the (normally implicit) object on which the member will be run. The signature we give to `function` must specify whether the object will be passed as a pointer or a reference.

When we defined `fcn`, we knew that we wanted to call `find_if` on a sequence of `string` objects. Hence, we asked `function` to generate a callable that took `string` objects. Had our `vector` held pointers to `string`, we would have told `function` to expect a pointer:

```

vector<string*> pvec;
function<bool (const string*)> fp = &string::empty;
// fp takes a pointer to string and uses the ->* to call empty
find_if(pvec.begin(), pvec.end(), fp);

```

## Using `mem_fn` to Generate a Callable

To use `function`, we must supply the call signature of the member we want to call. We can, instead, let the compiler deduce the member's type by using another library facility, `mem_fn`, which, like `function`, is defined in the functional header. Like `function`, `mem_fn` generates a callable object from a pointer to member. Unlike `function`, `mem_fn` will deduce the type of the callable from the type of the pointer to member:

```
find_if(svec.begin(), svec.end(), mem_fn(&string::empty));
```

Here we used `mem_fn(&string::empty)` to generate a callable object that takes a `string` argument and returns a `bool`.

The callable generated by `mem_fn` can be called on either an object or a pointer:

```

auto f = mem_fn(&string::empty); // f takes a string or a string*
f(*svec.begin()); // ok: passes a string object; f uses .* to call empty
f(&svec[0]); // ok: passes a pointer to string; f uses .-> to call empty

```

C++  
11

Effectively, we can think of `mem_fn` as if it generates a callable with an overloaded function call operator—one that takes a `string*` and the other a `string&`.

## Using `bind` to Generate a Callable

For completeness, we can also use `bind` (§ 10.3.4, p. 397) to generate a callable from a member function:

```

// bind each string in the range to the implicit first argument to empty
auto it = find_if(svec.begin(), svec.end(),
                  bind(&string::empty, _1));

```

As with `function`, when we use `bind`, we must make explicit the member function's normally implicit parameter that represents the object on which the member function will operate. Like `mem_fn`, the first argument to the callable generated by `bind` can be either a pointer or a reference to a `string`:

```

auto f = bind(&string::empty, _1);
f(*svec.begin()); // ok: argument is a string f will use .* to call empty
f(&svec[0]); // ok: argument is a pointer to string f will use .-> to call empty

```

## 19.5 Nested Classes

A class can be defined within another class. Such a class is a **nested class**, also referred to as a **nested type**. Nested classes are most often used to define implementation classes, such as the `QueryResult` class we used in our text query example (§ 12.3, p. 484).

### EXERCISES SECTION 19.4.3

**Exercise 19.18:** Write a function that uses `count_if` to count how many empty strings there are in a given vector.

**Exercise 19.19:** Write a function that takes a `vector<Sales_data>` and finds the first element whose average price is greater than some given amount.

Nested classes are independent classes and are largely unrelated to their enclosing class. In particular, objects of the enclosing and nested classes are independent from each other. An object of the nested type does not have members defined by the enclosing class. Similarly, an object of the enclosing class does not have members defined by the nested class.

The name of a nested class is visible within its enclosing class scope but not outside the class. Like any other nested name, the name of a nested class will not collide with the use of that name in another scope.

A nested class can have the same kinds of members as a nonnested class. Just like any other class, a nested class controls access to its own members using access specifiers. The enclosing class has no special access to the members of a nested class, and the nested class has no special access to members of its enclosing class.

A nested class defines a type member in its enclosing class. As with any other member, the enclosing class determines access to this type. A nested class defined in the `public` part of the enclosing class defines a type that may be used anywhere. A nested class defined in the `protected` section defines a type that is accessible only by the enclosing class, its friends, and its derived classes. A `private` nested class defines a type that is accessible only to the members and friends of the enclosing class.

### Declaring a Nested Class

The `TextQuery` class from § 12.3.2 (p. 487) defined a companion class named `QueryResult`. The `QueryResult` class is tightly coupled to our `TextQuery` class. It would make little sense to use `QueryResult` for any other purpose than to represent the results of a query operation on a `TextQuery` object. To reflect this tight coupling, we'll make `QueryResult` a member of `TextQuery`.

```
class TextQuery {
public:
    class QueryResult; // nested class to be defined later
    // other members as in § 12.3.2 (p. 487)
};
```

We need to make only one change to our original `TextQuery` class—we declare our intention to define `QueryResult` as a nested class. Because `QueryResult` is a type member (§ 7.4.1, p. 284), we must declare `QueryResult` before we use it. In particular, we must declare `QueryResult` before we use it as the return type for the `query` member. The remaining members of our original class are unchanged.

## Defining a Nested Class outside of the Enclosing Class

Inside `TextQuery` we declared `QueryResult` but did not define it. As with member functions, nested classes must be declared inside the class but can be defined either inside or outside the class.

When we define a nested class outside its enclosing class, we must qualify the name of the nested class by the name of its enclosing class:

```
// we're defining the QueryResult class that is a member of class TextQuery
class TextQuery::QueryResult {
    // in class scope, we don't have to qualify the name of the QueryResult parameters
    friend std::ostream&
        print(std::ostream&, const QueryResult&);

public:
    // no need to define QueryResult::line_no; a nested class can use a member
    // of its enclosing class without needing to qualify the member's name
    QueryResult(std::string,
                std::shared_ptr<std::set<line_no>>,
                std::shared_ptr<std::vector<std::string>>);

    // other members as in § 12.3.2 (p. 487)
};
```

The only change we made compared to our original class is that we no longer define a `line_no` member in `QueryResult`. The members of `QueryResult` can access that name directly from `TextQuery`, so there is no need to define it again.



Until the actual definition of a nested class that is defined outside the class body is seen, that class is an incomplete type (§ 7.3.3, p. 278).

## Defining the Members of a Nested Class

In this version, we did not define the `QueryResult` constructor inside the class body. To define the constructor, we must indicate that `QueryResult` is nested within the scope of `TextQuery`. We do so by qualifying the nested class name with the name of its enclosing class:

```
// defining the member named QueryResult for the class named QueryResult
// that is nested inside the class TextQuery
TextQuery::QueryResult::QueryResult(string s,
                                    shared_ptr<set<line_no>> p,
                                    shared_ptr<vector<string>> f) :
    sought(s), lines(p), file(f) { }
```

Reading the name of the function from right to left, we see that we are defining the constructor for class `QueryResult`, which is nested in the scope of class `TextQuery`. The code itself just stores the given arguments in the data members and has no further work to do.

## Nested-Class static Member Definitions

If `QueryResult` had declared a `static` member, its definition would appear outside the scope of the `TextQuery`. For example, assuming `QueryResult` had a

`static` member, its definition would look something like

```
// defines an int static member of QueryResult
// which is a class nested inside TextQuery
int TextQuery::QueryResult::static_mem = 1024;
```

## Name Lookup in Nested Class Scope

Normal rules apply for name lookup (§ 7.4.1, p. 283) inside a nested class. Of course, because a nested class is a nested scope, the nested class has additional enclosing class scopes to search. This nesting of scopes explains why we didn't define `line_no` inside the nested version of `QueryResult`. Our original `QueryResult` class defined this member so that its own members could avoid having to write `TextQuery::line_no`. Having nested the definition of our results class inside `TextQuery`, we no longer need this `typedef`. The nested `QueryResult` class can access `line_no` without specifying that `line_no` is defined in `TextQuery`.

As we've seen, a nested class is a type member of its enclosing class. Members of the enclosing class can use the name of a nested class the same way it can use any other type member. Because `QueryResult` is nested inside `TextQuery`, the `query` member of `TextQuery` can refer to the name `QueryResult` directly:

```
// return type must indicate that QueryResult is now a nested class
TextQuery::QueryResult
TextQuery::query(const string &sought) const
{
    // we'll return a pointer to this set if we don't find sought
    static shared_ptr<set<line_no>> nodata(new set<line_no>);
    // use find and not a subscript to avoid adding words to wm!
    auto loc = wm.find(sought);
    if (loc == wm.end())
        return QueryResult(sought, nodata, file); // not found
    else
        return QueryResult(sought, loc->second, file);
}
```

As usual, the return type is not yet in the scope of the class (§ 7.4, p. 282), so we start by noting that our function returns a `TextQuery::QueryResult` value. However, inside the body of the function, we can refer to `QueryResult` directly, as we do in the `return` statements.

## The Nested and Enclosing Classes Are Independent

Although a nested class is defined in the scope of its enclosing class, it is important to understand that there is no connection between the objects of an enclosing class and objects of its nested classe(s). A nested-type object contains only the members defined inside the nested type. Similarly, an object of the enclosing class has only those members that are defined by the enclosing class. It does not contain the data members of any nested classes.

More concretely, the second `return` statement in `TextQuery::query`

```
return QueryResult(sought, loc->second, file);
```

uses data members of the `TextQuery` object on which `query` was run to initialize a `QueryResult` object. We have to use these members to construct the `QueryResult` object we return because a `QueryResult` object does not contain the members of its enclosing class.

## EXERCISES SECTION 19.5

**Exercise 19.20:** Nest your `QueryResult` class inside `TextQuery` and rerun the programs you wrote to use `TextQuery` in § 12.3.2 (p. 490).

## 19.6 union: A Space-Saving Class

A **union** is a special kind of class. A union may have multiple data members, but at any point in time, only one of the members may have a value. When a value is assigned to one member of the union, all other members become undefined. The amount of storage allocated for a union is at least as much as is needed to contain its largest data member. Like any class, a union defines a new type.

Some, but not all, class features apply equally to unions. A union cannot have a member that is a reference, but it can have members of most other types, including, under the new standard, class types that have constructors or destructors. A union can specify protection labels to make members `public`, `private`, or `protected`. By default, like `structs`, members of a union are `public`.

A union may define member functions, including constructors and destructors. However, a union may not inherit from another class, nor may a union be used as a base class. As a result, a union may not have virtual functions.

### Defining a union

unions offer a convenient way to represent a set of mutually exclusive values of different types. As an example, we might have a process that handles different kinds of numeric or character data. That process might define a union to hold these values:

```
// objects of type Token have a single member, which could be of any of the listed types
union Token {
    // members are public by default
    char    cval;
    int     ival;
    double  dval;
};
```

A union is defined starting with the keyword `union`, followed by an (optional) name for the union and a set of member declarations enclosed in curly braces. This code defines a union named `Token` that can hold a value that is either a `char`, an `int`, or a `double`.

## Using a union Type

The name of a union is a type name. Like the built-in types, by default unions are uninitialized. We can explicitly initialize a union in the same way that we can explicitly initialize aggregate classes (§ 7.5.5, p. 298) by enclosing the initializer in a pair of curly braces:

```
Token first_token = {'a'}; // initializes the cval member
Token last_token;         // uninitialized Token object
Token *pt = new Token;    // pointer to an uninitialized Token object
```

If an initializer is present, it is used to initialize the first member. Hence, the initialization of `first_token` gives a value to its `cval` member.

The members of an object of union type are accessed using the normal member access operators:

```
last_token.cval = 'z';
pt->ival = 42;
```

Assigning a value to a data member of a union object makes the other data members undefined. As a result, when we use a union, we must always know what type of value is currently stored in the union. Depending on the types of the members, retrieving or assigning to the value stored in the union through the wrong data member can lead to a crash or other incorrect program behavior.

## Anonymous unions

An **anonymous union** is an unnamed union that does not include any declarations between the close curly that ends its body and the semicolon that ends the union definition (§ 2.6.1, p. 73). When we define an anonymous union the compiler automatically creates an unnamed object of the newly defined union type:

```
union {                      // anonymous union
    char   cval;
    int    ival;
    double dval;
}; // defines an unnamed object, whose members we can access directly
cval = 'c'; // assigns a new value to the unnamed, anonymous union object
ival = 42;   // that object now holds the value 42
```

The members of an anonymous union are directly accessible in the scope where the anonymous union is defined.



An anonymous union cannot have private or protected members, nor can an anonymous union define member functions.

## unions with Members of Class Type

Under earlier versions of C++, unions could not have members of a class type that defined its own constructors or copy-control members. Under the new standard, this restriction is lifted. However, unions with members that define their

own constructors and /or copy-control members are more complicated to use than unions that have members of built-in type.

When a union has members of built-in type, we can use ordinary assignment to change the value that the union holds. Not so for unions that have members of nontrivial class types. When we switch the union's value to and from a member of class type, we must construct or destroy that member, respectively: When we switch the union to a member of class type, we must run a constructor for that member's type; when we switch from that member, we must run its destructor.

When a union has members of built-in type, the compiler will synthesize the memberwise versions of the default constructor or copy-control members. The same is not true for unions that have members of a class type that defines its own default constructor or one or more of the copy-control members. If a union member's type defines one of these members, the compiler synthesizes the corresponding member of the union as deleted (§ 13.1.6, p. 508).

For example, the `string` class defines all five copy-control members and the default constructor. If a union contains a `string` and does not define its own default constructor or one of the copy-control members, then the compiler will synthesize that missing member as deleted. If a class has a union member that has a deleted copy-control member, then that corresponding copy-control operation(s) of the class itself will be deleted as well.

## Using a Class to Manage union Members

Because of the complexities involved in constructing and destroying members of class type, unions with class-type members ordinarily are embedded inside another class. That way the class can manage the state transitions to and from the member of class type. As an example, we'll add a `string` member to our union. We'll define our union as an anonymous union and make it a member of a class named `Token`. The `Token` class will manage the union's members.

To keep track of what type of value the union holds, we usually define a separate object known as a **discriminant**. A discriminant lets us discriminate among the values that the union can hold. In order to keep the union and its discriminant in sync, we'll make the discriminant a member of `Token` as well. Our class will define a member of an enumeration type (§ 19.3, p. 832) to keep track of the state of its union member.

The only functions our class will define are the default constructor, the copy-control members, and a set of assignment operators that can assign a value of one of our union's types to the union member:

```
class Token {
public:
    // copy control needed because our class has a union with a string member
    // defining the move constructor and move-assignment operator is left as an exercise
    Token() : tok(INT), ival{0} { }
    Token(const Token& t) : tok(t.tok) { copyUnion(t); }
    Token &operator=(const Token& t);
    // if the union holds a string, we must destroy it; see § 19.1.2 (p. 824)
    ~Token() { if (tok == STR) sval~string(); }
```

```

// assignment operators to set the differing members of the union
Token &operator=(const std::string&);
Token &operator=(char);
Token &operator=(int);
Token &operator=(double);

private:
    enum {INT, CHAR, DBL, STR} tok; // discriminant
    union {                                // anonymous union
        char   cval;
        int    ival;
        double dval;
        std::string sval;
    }; // each Token object has an unnamed member of this unnamed union type
    // check the discriminant and copy the union member as appropriate
    void copyUnion(const Token&);
};

```

Our class defines a nested, unnamed, unscoped enumeration (§ 19.3, p. 832) that we use as the type for the member named `tok`. We defined `tok` following the close curly and before the semicolon that ends the definition of the `enum`, which defines `tok` to have this unnamed `enum` type (§ 2.6.1, p. 73).

We'll use `tok` as our discriminant. When the union holds an `int` value, `tok` will have the value `INT`; if the union has a `string`, `tok` will be `STR`; and so on.

The default constructor initializes the discriminant and the union member to hold an `int` value of 0.

Because our union has a member with a destructor, we must define our own destructor to (conditionally) destroy the `string` member. Unlike ordinary members of a class type, class members that are part of a union are not automatically destroyed. The destructor has no way to know which type the union holds, so it cannot know which member to destroy.

Our destructor checks whether the object being destroyed holds a `string`. If so, the destructor explicitly calls the `string` destructor (§ 19.1.2, p. 824) to free the memory used by that `string`. The destructor has no work to do if the union holds a member of any of the built-in types.

## Managing the Discriminant and Destroying the `string`

The assignment operators will set `tok` and assign the corresponding member of the union. Like the destructor, these members must conditionally destroy the `string` before assigning a new value to the union:

```

Token &Token::operator=(int i)
{
    if (tok == STR) sval ~string(); // if we have a string, free it
    ival = i;                      // assign to the appropriate member
    tok = INT;                     // update the discriminant
    return *this;
}

```

If the current value in the union is a `string`, we must destroy that `string` before assigning a new value to the union. We do so by calling the `string` destructor.

Once we've cleaned up the `string` member, we assign the given value to the member that corresponds to the parameter type of the operator. In this case, our parameter is an `int`, so we assign to `ival`. We update the discriminant and return.

The `double` and `char` assignment operators behave identically to the `int` version and are left as an exercise. The `string` version differs from the others because it must manage the transition to and from the `string` type:

```
Token &Token::operator=(const std::string &s)
{
    if (tok == STR) // if we already hold a string, just do an assignment
        sval = s;
    else
        new(&sval) string(s); // otherwise construct a string
    tok = STR;           // update the discriminant
    return *this;
}
```

In this case, if the `union` already holds a `string`, we can use the normal `string` assignment operator to give a new value to that `string`. Otherwise, there is no existing `string` object on which to invoke the `string` assignment operator. Instead, we must construct a `string` in the memory that holds the `union`. We do so using `placement new` (§ 19.1.2, p. 824) to construct a `string` at the location in which `sval` resides. We initialize that `string` as a copy of our `string` parameter. We next update the discriminant and return.

## Managing Union Members That Require Copy Control

Like the type-specific assignment operators, the copy constructor and assignment operators have to test the discriminant to know how to copy the given value. To do this common work, we'll define a member named `copyUnion`.

When we call `copyUnion` from the copy constructor, the `union` member will have been default-initialized, meaning that the first member of the `union` will have been initialized. Because our `string` is not the first member, we know that the `union` member doesn't hold a `string`. In the assignment operator, it is possible that the `union` already holds a `string`. We'll handle that case directly in the assignment operator. That way `copyUnion` can assume that if its parameter holds a `string`, `copyUnion` must construct its own `string`:

```
void Token::copyUnion(const Token &t)
{
    switch (t.tok) {
        case Token::INT: ival = t.ival; break;
        case Token::CHAR: cval = t.cval; break;
        case Token::DBL: dval = t.dval; break;
        // to copy a string, construct it using placement new; see (§ 19.1.2 (p. 824))
        case Token::STR: new(&sval) string(t.sval); break;
    }
}
```

This function uses a `switch` statement (§ 5.3.2, p. 178) to test the discriminant. For

the built-in types, we assign the value to the corresponding member; if the member we are copying is a `string`, we construct it.

The assignment operator must handle three possibilities for its `string` member: Both the left-hand and right-hand operands might be a `string`; neither operand might be a `string`; or one but not both operands might be a `string`:

```
Token &Token::operator=(const Token &t)
{
    // if this object holds a string and t doesn't, we have to free the old string
    if (tok == STR && t.tok != STR) sval ~string();
    if (tok == STR && t.tok == STR)
        sval = t.sval; // no need to construct a new string
    else
        copyUnion(t); // will construct a string if t.tok is STR
    tok = t.tok;
    return *this;
}
```

If the union in the left-hand operand holds a `string`, but the union in the right-hand does not, then we have to first free the old `string` before assigning a new value to the union member. If both unions hold a `string`, we can use the normal `string` assignment operator to do the copy. Otherwise, we call `copyUnion` to do the assignment. Inside `copyUnion`, if the right-hand operand is a `string`, we'll construct a new `string` in the union member of the left-hand operand. If neither operand is a `string`, then ordinary assignment will suffice.

## EXERCISES SECTION 19.6

**Exercise 19.21:** Write your own version of the `Token` class.

**Exercise 19.22:** Add a member of type `Sales_data` to your `Token` class.

**Exercise 19.23:** Add a move constructor and move assignment to `Token`.

**Exercise 19.24:** Explain what happens if we assign a `Token` object to itself.

**Exercise 19.25:** Write assignment operators that take values of each type in the union.

## 19.7 Local Classes

A class can be defined inside a function body. Such a class is called a **local class**. A local class defines a type that is visible only in the scope in which it is defined. Unlike nested classes, the members of a local class are severely restricted.



All members, including functions, of a local class must be completely defined inside the class body. As a result, local classes are much less useful than nested classes.

In practice, the requirement that members be fully defined within the class limits the complexity of the member functions of a local class. Functions in local classes are rarely more than a few lines of code. Beyond that, the code becomes difficult for the reader to understand.

Similarly, a local class is not permitted to declare `static` data members, there being no way to define them.

## Local Classes May Not Use Variables from the Function's Scope

The names from the enclosing scope that a local class can access are limited. A local class can access only type names, `static` variables (§ 6.1.1, p. 205), and enumerators defined within the enclosing local scopes. A local class may not use the ordinary local variables of the function in which the class is defined:

```
int a, val;
void foo(int val)
{
    static int si;
    enum Loc { a = 1024, b };
    // Bar is local to foo
    struct Bar {
        Loc locVal; // ok: uses a local type name
        int barVal;
        void fooBar(Loc l = a) // ok: default argument is Loc::a
        {
            barVal = val; // error: val is local to foo
            barVal = ::val; // ok: uses a global object
            barVal = si; // ok: uses a static local object
            locVal = b; // ok: uses an enumerator
        }
    };
    // ...
}
```

## Normal Protection Rules Apply to Local Classes

The enclosing function has no special access privileges to the `private` members of the local class. Of course, the local class could make the enclosing function a friend. More typically, a local class defines its members as `public`. The portion of a program that can access a local class is very limited. A local class is already encapsulated within the scope of the function. Further encapsulation through information hiding is often overkill.

## Name Lookup within a Local Class

Name lookup within the body of a local class happens in the same manner as for other classes. Names used in the declarations of the members of the class must be in scope before the use of the name. Names used in the definition of a member can appear anywhere in the class. If a name is not found as a class member, then

the search continues in the enclosing scope and then out to the scope enclosing the function itself.

## Nested Local Classes

It is possible to nest a class inside a local class. In this case, the nested class definition can appear outside the local-class body. However, the nested class must be defined in the same scope as that in which the local class is defined.

```
void foo()
{
    class Bar {
        public:
            // ...
            class Nested;      // declares class Nested
    };
    // definition of Nested
    class Bar::Nested {
        // ...
    };
}
```

As usual, when we define a member outside a class, we must indicate the scope of the name. Hence, we defined `Bar::Nested`, which says that `Nested` is a class defined in the scope of `Bar`.

A class nested in a local class is itself a local class, with all the attendant restrictions. All members of the nested class must be defined inside the body of the nested class itself.

## 19.8 Inherently Nonportable Features

To support low-level programming, C++ defines some features that are inherently **nonportable**. A nonportable feature is one that is machine specific. Programs that use nonportable features often require reprogramming when they are moved from one machine to another. The fact that the sizes of the arithmetic types vary across machines (§ 2.1.1, p. 32) is one such nonportable feature that we have already used.

In this section we'll cover two additional nonportable features that C++ inherits from C: bit-fields and the `volatile` qualifier. We'll also cover linkage directives, which is a nonportable feature that C++ adds to those that it inherits from C.

### 19.8.1 Bit-fields

A class can define a (nonstatic) data member as a **bit-field**. A bit-field holds a specified number of bits. Bit-fields are normally used when a program needs to pass binary data to another program or to a hardware device.



The memory layout of a bit-field is machine dependent.

A bit-field must have integral or enumeration type (§ 19.3, p. 832). Ordinarily, we use an unsigned type to hold a bit-field, because the behavior of a signed bit-field is implementation defined. We indicate that a member is a bit-field by following the member name with a colon and a constant expression specifying the number of bits:

```
typedef unsigned int Bit;
class File {
    Bit mode: 2;           // mode has 2 bits
    Bit modified: 1;       // modified has 1 bit
    Bit prot_owner: 3;     // prot_owner has 3 bits
    Bit prot_group: 3;     // prot_group has 3 bits
    Bit prot_world: 3;     // prot_world has 3 bits
    // operations and data members of File
public:
    // file modes specified as octal literals; see § 2.1.3 (p. 38)
    enum modes { READ = 01, WRITE = 02, EXECUTE = 03 } ;
    File &open(modes);
    void close();
    void write();
    bool isRead() const;
    void setWrite();
};
```

The mode bit-field has two bits, modified only one, and the other members each have three bits. Bit-fields defined in consecutive order within the class body are, if possible, packed within adjacent bits of the same integer, thereby providing for storage compaction. For example, in the preceding declaration, the five bit-fields will (probably) be stored in a single unsigned int. Whether and how the bits are packed into the integer is machine dependent.

The address-of operator (`&`) cannot be applied to a bit-field, so there can be no pointers referring to class bit-fields.



Ordinarily it is best to make a bit-field an unsigned type. The behavior of bit-fields stored in a signed type is implementation defined.

## Using Bit-fields

A bit-field is accessed in much the same way as the other data members of a class:

```
void File::write()
{
    modified = 1;
    // ...
}
void File::close()
{
    if (modified)
        // ... save contents
}
```

Bit-fields with more than one bit are usually manipulated using the built-in bitwise operators (§ 4.8, p. 152):

```
File &File::open(File::modes m)
{
    mode |= READ;      // set the READ bit by default
    // other processing
    if (m & WRITE) // if opening READ and WRITE
        // processing to open the file in read/write mode
    return *this;
}
```

Classes that define bit-field members also usually define a set of inline member functions to test and set the value of the bit-field:

```
inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }
```

## 19.8.2 **volatile** Qualifier



The precise meaning of **volatile** is inherently machine dependent and can be understood only by reading the compiler documentation. Programs that use **volatile** usually must be changed when they are moved to new machines or compilers.

Programs that deal directly with hardware often have data elements whose value is controlled by processes outside the direct control of the program itself. For example, a program might contain a variable updated by the system clock. An object should be declared **volatile** when its value might be changed in ways outside the control or detection of the program. The **volatile** keyword is a directive to the compiler that it should not perform optimizations on such objects.

The **volatile** qualifier is used in much the same way as the **const** qualifier. It is an additional modifier to a type:

```
volatile int display_register; // int value that might change
volatile Task *curr_task; // curr_task points to a volatile object
volatile int iax[max_size]; // each element in iax is volatile
volatile Screen bitmapBuf; // each member of bitmapBuf is volatile
```

There is no interaction between the **const** and **volatile** type qualifiers. A type can be both **const** and **volatile**, in which case it has the properties of both.

In the same way that a class may define **const** member functions, it can also define member functions as **volatile**. Only **volatile** member functions may be called on **volatile** objects.

§ 2.4.2 (p. 62) described the interactions between the **const** qualifier and pointers. The same interactions exist between the **volatile** qualifier and pointers. We can declare pointers that are **volatile**, pointers to **volatile** objects, and pointers that are **volatile** that point to **volatile** objects:

```
volatile int v;      // v is a volatile int
int *volatile vip; // vip is a volatile pointer to int
volatile int *ivp; // ivp is a pointer to volatile int
// vivp is a volatile pointer to volatile int
volatile int *volatile vivp;
int *ip = &v; // error: must use a pointer to volatile
*ivp = &v;    // ok: ivp is a pointer to volatile
vivp = &v;    // ok: vivp is a volatile pointer to volatile
```

As with `const`, we may assign the address of a `volatile` object (or copy a pointer to a `volatile` type) only to a pointer to `volatile`. We may use a `volatile` object to initialize a reference only if the reference is `volatile`.

### Synthesized Copy Does Not Apply to `volatile` Objects

One important difference between the treatment of `const` and `volatile` is that the synthesized copy/move and assignment operators cannot be used to initialize or assign from a `volatile` object. The synthesized members take parameters that are references to (nonvolatile) `const`, and we cannot bind a nonvolatile reference to a `volatile` object.

If a class wants to allow `volatile` objects to be copied, moved, or assigned, it must define its own versions of the copy or move operation. As one example, we might write the parameters as `const volatile` references, in which case we can copy or assign from any kind of `Foo`:

```
class Foo {
public:
    Foo(const volatile Foo&); // copy from a volatile object
    // assign from a volatile object to a nonvolatile object
    Foo& operator=(volatile const Foo&);
    // assign from a volatile object to a volatile object
    Foo& operator=(volatile const Foo&) volatile;
    // remainder of class Foo
};
```

Although we can define copy and assignment for `volatile` objects, a deeper question is whether it makes any sense to copy a `volatile` object. The answer to that question depends intimately on the reason for using `volatile` in any particular program.

#### 19.8.3 Linkage Directives: `extern "C"`

C++ programs sometimes need to call functions written in another programming language. Most often, that other language is C. Like any name, the name of a function written in another language must be declared. As with any function, that declaration must specify the return type and parameter list. The compiler checks calls to functions written in another language in the same way that it handles ordinary C++ functions. However, the compiler typically must generate different

code to call functions written in other languages. C++ uses **linkage directives** to indicate the language used for any non-C++ function.



Mixing C++ with code written in any other language, including C, requires access to a compiler for that language that is compatible with your C++ compiler.

## Declaring a Non-C++ Function

A linkage directive can have one of two forms: single or compound. Linkage directives may not appear inside a class or function definition. The same linkage directive must appear on every declaration of a function.

As an example, the following declarations shows how some of the C functions in the `cstring` header might be declared:

```
// illustrative linkage directives that might appear in the C++ header <cstring>
// single-statement linkage directive
extern "C" size_t strlen(const char *) ;
// compound-statement linkage directive
extern "C" {
    int strcmp(const char*, const char*) ;
    char *strcat(char*, const char*) ;
}
```

The first form of a linkage directive consists of the `extern` keyword followed by a string literal, followed by an “ordinary” function declaration.

The string literal indicates the language in which the function is written. A compiler is required to support linkage directives for C. A compiler may provide linkage specifications for other languages, for example, `extern "Ada"`, `extern "FORTRAN"`, and so on.

## Linkage Directives and Headers

We can give the same linkage to several functions at once by enclosing their declarations inside curly braces following the linkage directive. These braces serve to group the declarations to which the linkage directive applies. The braces are otherwise ignored, and the names of functions declared within the braces are visible as if the functions were declared outside the braces.

The multiple-declaration form can be applied to an entire header file. For example, the C++ `cstring` header might look like

```
// compound-statement linkage directive
extern "C" {
#include <string.h>      // C functions that manipulate C-style strings
}
```

When a `#include` directive is enclosed in the braces of a compound-linkage directive, all ordinary function declarations in the header file are assumed to be functions written in the language of the linkage directive. Linkage directives can be

nested, so if a header contains a function with its own linkage directive, the linkage of that function is unaffected.



The functions that C++ inherits from the C library are permitted to be defined as C functions but are not required to be C functions—it's up to each C++ implementation to decide whether to implement the C library functions in C or C++.

## Pointers to `extern "C"` Functions

The language in which a function is written is part of its type. Hence, every declaration of a function defined with a linkage directive must use the same linkage directive. Moreover, pointers to functions written in other languages must be declared with the same linkage directive as the function itself:

```
// pf points to a C function that returns void and takes an int
extern "C" void (*pf)(int);
```

When `pf` is used to call a function, the function call is compiled assuming that the call is to a C function.

A pointer to a C function does not have the same type as a pointer to a C++ function. A pointer to a C function cannot be initialized or be assigned to point to a C++ function (and vice versa). As with any other type mismatch, it is an error to try to assign two pointers with different linkage directives:

```
void (*pf1)(int);           // points to a C++ function
extern "C" void (*pf2)(int); // points to a C function
pf1 = pf2; // error: pf1 and pf2 have different types
```



Some C++ compilers may accept the preceding assignment as a language extension, even though, strictly speaking, it is illegal.

## Linkage Directives Apply to the Entire Declaration

When we use a linkage directive, it applies to the function and any function pointers used as the return type or as a parameter type:

```
// f1 is a C function; its parameter is a pointer to a C function
extern "C" void f1(void(*)(int));
```

This declaration says that `f1` is a C function that doesn't return a value. It has one parameter, which is a pointer to a function that returns nothing and takes a single `int` parameter. The linkage directive applies to the function pointer as well as to `f1`. When we call `f1`, we must pass it the name of a C function or a pointer to a C function.

Because a linkage directive applies to all the functions in a declaration, we must use a type alias (§ 2.5.1, p. 67) if we wish to pass a pointer to a C function to a C++ function:

```
// FC is a pointer to a C function
extern "C" typedef void FC(int);
// f2 is a C++ function with a parameter that is a pointer to a C function
void f2(FC *);
```

## Exporting Our C++ Functions to Other Languages

By using the linkage directive on a function definition, we can make a C++ function available to a program written in another language:

```
// the calc function can be called from C programs
extern "C" double calc(double dparm) { /* ... */ }
```

When the compiler generates code for this function, it will generate code appropriate to the indicated language.

It is worth noting that the parameter and return types in functions that are shared across languages are often constrained. For example, we can almost surely not write a function that passes objects of a (nontrivial) C++ class to a C program. The C program won't know about the constructors, destructors, or other class-specific operations.

### PREPROCESSOR SUPPORT FOR LINKING TO C

To allow the same source file to be compiled under either C or C++, the preprocessor defines `__cplusplus` (two underscores) when we compile C++. Using this variable, we can conditionally include code when we are compiling C++:

```
#ifdef __cplusplus
// ok: we're compiling C++
extern "C"
#endif
int strcmp(const char*, const char*);
```

## Overloaded Functions and Linkage Directives

The interaction between linkage directives and function overloading depends on the target language. If the language supports overloaded functions, then it is likely that a compiler that implements linkage directives for that language would also support overloading of these functions from C++.

The C language does not support function overloading, so it should not be a surprise that a C linkage directive can be specified for only one function in a set of overloaded functions:

```
// error: two extern "C" functions with the same name
extern "C" void print(const char*);
extern "C" void print(int);
```

If one function among a set of overloaded functions is a C function, the other functions must all be C++ functions:

```
class SmallInt { /* ... */ };
class BigNum { /* ... */ };

// the C function can be called from C and C++ programs
// the C++ functions overload that function and are callable from C++
extern "C" double calc(double);
extern SmallInt calc(const SmallInt&);
extern BigNum calc(const BigNum&);
```

The C version of `calc` can be called from C programs and from C++ programs. The additional functions are C++ functions with class parameters that can be called only from C++ programs. The order of the declarations is not significant.

### EXERCISES SECTION 19.8.3

**Exercise 19.26:** Explain these declarations and indicate whether they are legal:

```
extern "C" int compute(int *, int);
extern "C" double compute(double *, double);
```

## CHAPTER SUMMARY

---

C++ provides several specialized facilities that are tailored to particular kinds of problems.

Some applications need to take control of how memory is allocated. They can do so by defining their own versions—either class specific or global—of the library operator new and operator delete functions. If the application defines its own versions of these functions, new and delete expressions will use the application-defined version.

Some programs need to directly interrogate the dynamic type of an object at run time. Run-time type identification (RTTI) provides language-level support for this kind of programming. RTTI applies only to classes that define virtual functions; type information for types that do not define virtual functions is available but reflects the static type.

When we define a pointer to a class member, the pointer type also encapsulates the type of the class containing the member to which the pointer points. A pointer to member may be bound to any member of the class that has the appropriate type. When we dereference a pointer to member, we must supply an object from which to fetch the member.

C++ defines several additional aggregate types:

- Nested classes, which are classes defined in the scope of another class. Such classes are often defined as implementation classes of their enclosing class.
- unions are a special kind of class that may define several data members, but at any point in time, only one member may have a value. unions are most often nested inside another class type.
- Local classes, which are defined inside a function. All members of a local class must be defined in the class body. There are no `static` data members of a local class.

C++ also supports several inherently nonportable features, including bit-fields and `volatile`, which make it easier to interface to hardware, and linkage directives, which make it easier to interface to programs written in other languages.

## DEFINED TERMS

---

**anonymous union** Unnamed union that is not used to define an object. Members of an anonymous union become members of the surrounding scope. These unions may not have member functions and may not have private or protected members.

**bit-field** Class member with a integral type that specifies the number of bits to allocate to the member. Bit-fields defined in consecutive order in the class are, if possible, com-

pacted into a common integral value.

**discriminant** Programming technique that uses an object to determine which actual type is held in a union at any given time.

**dynamic\_cast** Operator that performs a checked cast from a base type to a derived type. When the base type has at least one virtual function, the operator checks the dynamic type of the object to which the refer-

ence or pointer is bound. If the object type is the same as the type of the cast (or a type derived from that type), then the cast is done. Otherwise, a zero pointer is returned for a pointer cast, or an exception is thrown for a cast to a reference type.

**enumeration** Type that groups a set of named integral constants.

**enumerator** Member of an enumeration. Enumerators are `const` and may be used where integral constant expressions are required.

**free** Low-level memory deallocation function defined in `cstdlib`. `free` may be used *only* to free memory allocated by `malloc`.

**linkage directive** Mechanism used to allow functions written in a different language to be called from a C++ program. All compilers must support calling C and C++ functions. It is compiler dependent whether any other languages are supported.

**local class** Class defined inside a function. A local class is visible only inside the function in which it is defined. All members of the class must be defined inside the class body. There can be no `static` members of a local class. Local class members may not access the `nonstatic` variables defined in the enclosing function. They may use type names, `static` variables, or enumerators defined in the enclosing function.

**malloc** Low-level memory allocation function defined in `cstdlib`. Memory allocated by `malloc` must be freed by `free`.

**mem\_fn** Library class template that generates a callable object from a given pointer to member function.

**nested class** Class defined inside another class. A nested class is defined inside its enclosing scope: Nested-class names must be unique within the class scope in which they are defined but can be reused in scopes outside the enclosing class. Access to the

nested class outside the enclosing class requires use of the scope operator to specify the scope(s) in which the class is nested.

**nested type** Synonym for nested class.

**nonportable** Features that are inherently machine specific and may require change when a program is ported to another machine or compiler.

**operator delete** Library function that frees untyped, unconstructed memory allocated by `operator new`. The library `operator delete []` frees memory used to hold an array that was allocated by `operator new []`.

**operator new** Library function that allocates untyped, unconstructed memory of a given size. The library function `operator new []` allocates raw memory for arrays. These library functions provide a more primitive allocation mechanism than the library `allocator` class. Modern C++ programs should use the `allocator` classes rather than these library functions.

**placement new expression** Form of `new` that constructs its object in specified memory. It does no allocation; instead, it takes an argument that specifies where the object should be constructed. It is a lower-level analog of the behavior provided by the `construct` member of the `allocator` class.

**pointer to member** Pointer that encapsulates the class type as well as the member type to which the pointer points. The definition of a pointer to member must specify the class name as well as the type of the member(s) to which the pointer may point:

```
T C::*pmem = &C::member;
```

This statement defines `pmem` as a pointer that can point to members of the class named `C` that have type `T` and initializes `pmem` to point to the member in `C` named `member`. To use the pointer, we must supply an object or pointer to type `C`:

```
classobj.*pmem;
classptr->*pmem;
```

fetches member from the object `classobj` of the object pointed to by `classptr`.

**run-time type identification** Language and library facilities that allow the dynamic type of a reference or pointer to be obtained at run time. The RTTI operators, `typeid` and `dynamic_cast`, provide the dynamic type only for references or pointers to class types with virtual functions. When applied to other types, the type returned is the static type of the reference or pointer.

**scoped enumeration** New-style enumeration in which the enumerator are not accessible directly in the surrounding scope.

**typeid operator** Unary operator that returns a reference to an object of the library type named `type_info` that describes the type of the given expression. When the expression is an object of a type that has virtual functions, then the dynamic type of the expression is returned; such expressions are evaluated at run time. If the type is a reference, pointer, or other type that does not define virtual functions, then the type returned is the static type of the reference, pointer, or object; such expressions are not evaluated.

**type\_info** Library type returned by the `typeid` operator. The `type_info` class is inherently machine dependent, but must provide a small set of operations, including a `name` function that returns a character string representing the type's name. `type_info` objects may not be copied, moved, or assigned.

**union** Classlike aggregate type that may define multiple data members, only one of which can have a value at any one point. Unions may have member functions, including constructors and destructors. A union may not serve as a base class. Under the new standard, unions can have members that are class types that define their own copy-control members. Such unions obtain deleted copy control if they do not themselves define the corresponding copy-control functions.

**unscoped enumeration** Enumeration in which the enumerators are accessible in the surrounding scope.

**volatile** Type qualifier that signifies to the compiler that a variable might be changed outside the direct control of the program. It is a signal to the compiler that it may not perform certain optimizations.

# A P P E N D I X

# A

## THE LIBRARY

### CONTENTS

---

|                                                      |     |
|------------------------------------------------------|-----|
| Section A.1 Library Names and Headers . . . . .      | 866 |
| Section A.2 A Brief Tour of the Algorithms . . . . . | 870 |
| Section A.3 Random Numbers . . . . .                 | 882 |

This Appendix contains additional details about the algorithms and random number parts of the library. We also provide a list of all the names we used from the standard library along with the name of the header that defines that name.

In Chapter 10 we used some of the more common algorithms and described the architecture that underlies the algorithms. In this Appendix, we list all the algorithms, organized by the kinds of operations they perform.

In § 17.4 (p. 745) we described the architecture of the random number library and used several of the library’s distribution types. The library defines a number or random number engines and 20 different distributions. In this Appendix, we list all the engines and distributions.

## A.1 Library Names and Headers

Our programs mostly did not show the actual #include directives needed to compile the program. As a convenience to readers, Table A.1 lists the library names our programs used and the header in which they may be found.

**Table A.1: Standard Library Names and Headers**

| Name                   | Header       |
|------------------------|--------------|
| abort                  | <cstdlib>    |
| accumulate             | <numeric>    |
| allocator              | <memory>     |
| array                  | <array>      |
| auto_ptr               | <memory>     |
| back_inserter          | <iterator>   |
| bad_alloc              | <new>        |
| bad_array_new_length   | <new>        |
| bad_cast               | <typeinfo>   |
| begin                  | <iterator>   |
| bernoulli_distribution | <random>     |
| bind                   | <functional> |
| bitset                 | <bitset>     |
| boolalpha              | <iostream>   |
| cerr                   | <iostream>   |
| cin                    | <iostream>   |
| cmatch                 | <regex>      |
| copy                   | <algorithm>  |
| count                  | <algorithm>  |
| count_if               | <algorithm>  |
| cout                   | <iostream>   |
| cref                   | <functional> |
| csub_match             | <regex>      |
| dec                    | <iostream>   |
| default_float_engine   | <iostream>   |
| default_random_engine  | <random>     |
| deque                  | <deque>      |
| domain_error           | <stdexcept>  |
| end                    | <iterator>   |
| endl                   | <iostream>   |
| ends                   | <iostream>   |
| equal_range            | <algorithm>  |
| exception              | <exception>  |
| fill                   | <algorithm>  |
| fill_n                 | <algorithm>  |
| find                   | <algorithm>  |
| find_end               | <algorithm>  |
| find_first_of          | <algorithm>  |

Table A.1: Standard Library Names and Headers (continued)

| Name               | Header             |
|--------------------|--------------------|
| find_if            | <algorithm>        |
| fixed              | <iostream>         |
| flush              | <iostream>         |
| for_each           | <algorithm>        |
| forward            | <utility>          |
| forward_list       | <forward_list>     |
| free               | cstdlib            |
| front_inserter     | <iterator>         |
| fstream            | <fstream>          |
| function           | <functional>       |
| get                | <tuple>            |
| getline            | <string>           |
| greater            | <functional>       |
| hash               | <functional>       |
| hex                | <iostream>         |
| hexfloat           | <iostream>         |
| ifstream           | <fstream>          |
| initializer_list   | <initializer_list> |
| inserter           | <iterator>         |
| internal           | <iostream>         |
| ios_base           | <ios_base>         |
| isalpha            | <cctype>           |
| islower            | <cctype>           |
| isprint            | <cctype>           |
| ispunct            | <cctype>           |
| isspace            | <cctype>           |
| istream            | <iostream>         |
| istream_iterator   | <iterator>         |
| istringstream      | <sstream>          |
| isupper            | <cctype>           |
| left               | <iostream>         |
| less               | <functional>       |
| less_equal         | <functional>       |
| list               | <list>             |
| logic_error        | <stdexcept>        |
| lower_bound        | <algorithm>        |
| lround             | <cmath>            |
| make_move_iterator | <iterator>         |
| make_pair          | <utility>          |
| make_shared        | <memory>           |
| make_tuple         | <tuple>            |
| malloc             | cstdlib            |
| map                | <map>              |

**Table A.1: Standard Library Names and Headers (continued)**

| Name                | Header       |
|---------------------|--------------|
| max                 | <algorithm>  |
| max_element         | <algorithm>  |
| mem_fn              | <functional> |
| min                 | <algorithm>  |
| move                | <utility>    |
| multimap            | <map>        |
| multiset            | <set>        |
| negate              | <functional> |
| noboolalpha         | <iostream>   |
| normal_distribution | <random>     |
| noshowbase          | <iostream>   |
| noshowpoint         | <iostream>   |
| noskipws            | <iostream>   |
| not1                | <functional> |
| nothrow             | <new>        |
| nothrow_t           | <new>        |
| nounitbuf           | <iostream>   |
| nouppercase         | <iostream>   |
| nth_element         | <algorithm>  |
| oct                 | <iostream>   |
| ofstream            | <fstream>    |
| ostream             | <iostream>   |
| ostream_iterator    | <iterator>   |
| ostringstream       | <sstream>    |
| out_of_range        | <stdexcept>  |
| pair                | <utility>    |
| partial_sort        | <algorithm>  |
| placeholders        | <functional> |
| placeholders::_1    | <functional> |
| plus                | <functional> |
| priority_queue      | <queue>      |
| ptrdiff_t           | <cstddef>    |
| queue               | <queue>      |
| rand                | <random>     |
| random_device       | <random>     |
| range_error         | <stdexcept>  |
| ref                 | <functional> |
| regex               | <regex>      |
| regex_constants     | <regex>      |
| regex_error         | <regex>      |
| regex_match         | <regex>      |
| regex_replace       | <regex>      |
| regex_search        | <regex>      |

Table A.1: Standard Library Names and Headers (continued)

| Name             | Header        |
|------------------|---------------|
| remove_pointer   | <type_traits> |
| remove_reference | <type_traits> |
| replace          | <algorithm>   |
| replace_copy     | <algorithm>   |
| reverse_iterator | <iterator>    |
| right            | <iostream>    |
| runtime_error    | <stdexcept>   |
| scientific       | <iostream>    |
| set              | <set>         |
| set_difference   | <algorithm>   |
| set_intersection | <algorithm>   |
| set_union        | <algorithm>   |
| setfill          | <iomanip>     |
| setprecision     | <iomanip>     |
| setw             | <iomanip>     |
| shared_ptr       | <memory>      |
| showbase         | <iostream>    |
| showpoint        | <iostream>    |
| size_t           | <cstddef>     |
| skipws           | <iostream>    |
| smatch           | <regex>       |
| sort             | <algorithm>   |
| sqrt             | <cmath>       |
| sregex_iterator  | <regex>       |
| ssub_match       | <regex>       |
| stable_sort      | <algorithm>   |
| stack            | <stack>       |
| stoi             | <string>      |
| strcmp           | <cstring>     |
| strcpy           | <cstring>     |
| string           | <string>      |
| stringstream     | <sstream>     |
| strlen           | <cstring>     |
| strncpy          | <cstring>     |
| strtod           | <string>      |
| swap             | <utility>     |
| terminate        | <exception>   |
| time             | <ctime>       |
| tolower          | <cctype>      |
| toupper          | <cctype>      |
| transform        | <algorithm>   |
| tuple            | <tuple>       |
| tuple_element    | <tuple>       |

**Table A.1: Standard Library Names and Headers (continued)**

| Name                      | Header          |
|---------------------------|-----------------|
| tuple_size                | <tuple>         |
| type_info                 | <typeinfo>      |
| unexpected                | <exception>     |
| uniform_int_distribution  | <random>        |
| uniform_real_distribution | <random>        |
| uninitialized_copy        | <memory>        |
| uninitialized_fill        | <memory>        |
| unique                    | <algorithm>     |
| unique_copy               | <algorithm>     |
| unique_ptr                | <memory>        |
| unitbuf                   | <iostream>      |
| unordered_map             | <unordered_map> |
| unordered_multimap        | <unordered_map> |
| unordered_multiset        | <unordered_set> |
| unordered_set             | <unordered_set> |
| upper_bound               | <algorithm>     |
| uppercase                 | <iostream>      |
| vector                    | <vector>        |
| weak_ptr                  | <memory>        |

## A.2 A Brief Tour of the Algorithms

The library defines more than 100 algorithms. Learning to use these algorithms effectively requires understanding their structure rather than memorizing the details of each algorithm. Accordingly, in Chapter 10 we concentrated on describing and understanding that architecture. In this section we'll briefly describe every algorithm. In the following descriptions,

- `beg` and `end` are iterators that denote a range of elements (§ 9.2.1, p. 331). Almost all of the algorithms operate on a sequence denoted by `beg` and `end`.
- `beg2` is an iterator denoting the beginning of a second input sequence. If present, `end2` denotes the end of the second sequence. When there is no `end2`, the sequence denoted by `beg2` is assumed to be as large as the input sequence denoted by `beg` and `end`. The types of `beg` and `beg2` need not match. However, it must be possible to apply the specified operation or given callable object to elements in the two sequences.
- `dest` is an iterator denoting a destination. The destination sequence must be able to hold as many elements as necessary given the input sequence.
- `unaryPred` and `binaryPred` are unary and binary predicates (§ 10.3.1, p. 386) that return a type that can be used as a condition and take one and two arguments, respectively, that are elements in the input range.

- `comp` is a binary predicate that meets the ordering requirements for `key` in an associative container (§ 11.2.2, p. 425).
- `unaryOp` and `binaryOp` are callable objects (§ 10.3.2, p. 388) that can be called with one and two arguments from the input range, respectively.

## A.2.1 Algorithms to Find an Object

These algorithms search an input range for a specific value or sequence of values.

Each algorithm provides two overloaded versions. The first version uses equality (`==`) operator of the underlying type to compare elements; the second version compares elements using the user-supplied `unaryPred` or `binaryPred`.

### Simple Find Algorithms

These algorithms look for specific values and require *input iterators*.

```
find(beg, end, val)
find_if(beg, end, unaryPred)
find_if_not(beg, end, unaryPred)
count(beg, end, val)
count_if(beg, end, unaryPred)
```

`find` returns an iterator to the first element in the input range equal to `val`.  
`find_if` returns an iterator to the first element for which `unaryPred` succeeds;  
`find_if_not` returns an iterator to the first element for which `unaryPred` is false. All three return `end` if no such element exists.

`count` returns a count of how many times `val` occurs; `count_if` counts elements for which `unaryPred` succeeds.

```
all_of(beg, end, unaryPred)
any_of(beg, end, unaryPred)
none_of(beg, end, unaryPred)
```

Returns a `bool` indicating whether the `unaryPred` succeeded for all of the elements, any element, or no element respectively. If the sequence is empty, `any_of` returns `false`; `all_of` and `none_of` return `true`.

### Algorithms to Find One of Many Values

These algorithms require *forward iterators*. They look for a repeated elements in the input sequence.

```
adjacent_find(beg, end)
adjacent_find(beg, end, binaryPred)
```

Returns an iterator to the first adjacent pair of duplicate elements. Returns `end` if there are no adjacent duplicate elements.

```
search_n(beg, end, count, val)
search_n(beg, end, count, val, binaryPred)
```

Returns an iterator to the beginning of a subsequence of `count` equal elements. Returns `end` if no such subsequence exists.

## Algorithms to Find Subsequences

With the exception of `find_first_of`, these algorithms require two pairs of *forward iterators*. `find_first_of` uses *input iterators* to denote its first sequence and *forward iterators* for its second. These algorithms search for subsequences rather than for a single element.

```
search(beg1, end1, beg2, end2)
search(beg1, end1, beg2, end2, binaryPred)
```

Returns an iterator to the first position in the input range at which the second range occurs as a subsequence. Returns `end1` if the subsequence is not found.

```
find_first_of(beg1, end1, beg2, end2)
find_first_of(beg1, end1, beg2, end2, binaryPred)
```

Returns an iterator to the first occurrence in the first range of any element from the second range. Returns `end1` if no match is found.

```
find_end(beg1, end1, beg2, end2)
find_end(beg1, end1, beg2, end2, binaryPred)
```

Like `search`, but returns an iterator to the last position in the input range at which the second range occurs as a subsequence. Returns `end1` if the second subsequence is empty or is not found.

### A.2.2 Other Read-Only Algorithms

These algorithms require *input iterators* for their first two arguments.

The `equal` and `mismatch` algorithms also take an additional *input iterator* that denotes the start of a second range. They also provide two overloaded versions. The first version uses equality (`==`) operator of the underlying type to compare elements; the second version compares elements using the user-supplied `unaryPred` or `binaryPred`.

```
for_each(beg, end, unaryOp)
```

Applies the callable object (§ 10.3.2, p. 388) `unaryOp` to each element in its input range. The return value from `unaryOp` (if any) is ignored. If the iterators allow writing to elements through the dereference operator, then `unaryOp` may modify the elements.

```
mismatch(beg1, end1, beg2)
mismatch(beg1, end1, beg2, binaryPred)
```

Compares the elements in two sequences. Returns a pair (§ 11.2.3, p. 426) of iterators denoting the first elements in each sequence that do not match. If all the elements match, then the pair returned is `end1`, and an iterator into `beg2` offset by the size of the first sequence.

```
equal(beg1, end1, beg2)
equal(beg1, end1, beg2, binaryPred)
```

Determines whether two sequences are equal. Returns `true` if each element in the input range equals the corresponding element in the sequence that begins at `beg2`.

### A.2.3 Binary Search Algorithms

These algorithms require *forward iterators* but are optimized so that they execute much more quickly if they are called with *random-access iterators*. Technically speaking, regardless of the iterator type, these algorithms execute a logarithmic number of comparisons. However, when used with forward iterators, they must make a linear number of iterator operations to move among the elements in the sequence.

These algorithms require that the elements in the input sequence are already in order. These algorithms behave similarly to the associative container members of the same name (§ 11.3.5, p. 438). The `equal_range`, `lower_bound`, and `upper_bound` algorithms return iterators that refer to positions in the sequence at which the given element can be inserted while still preserving the sequence's ordering. If the element is larger than any other in the sequence, then the iterator that is returned might be the off-the-end iterator.

Each algorithm provides two versions: The first uses the element type's less-than operator (`<`) to test elements; the second uses the given comparison operation. In the following algorithms, “ $x$  is less than  $y$ ” means  $x < y$  or that `comp(x, y)` succeeds.

`lower_bound(beg, end, val)`  
`lower_bound(beg, end, val, comp)`

Returns an iterator denoting the first element such that `val` is not less than that element, or `end` if no such element exists.

`upper_bound(beg, end, val)`  
`upper_bound(beg, end, val, comp)`

Returns an iterator denoting the first element such that `val` is less than that element, or `end` if no such element exists.

`equal_range(beg, end, val)`  
`equal_range(beg, end, val, comp)`

Returns a pair (§ 11.2.3, p. 426) in which the first member is the iterator that would be returned by `lower_bound`, and second is the iterator `upper_bound` would return.

`binary_search(beg, end, val)`  
`binary_search(beg, end, val, comp)`

Returns a `bool` indicating whether the sequence contains an element that is equal to `val`. Two values  $x$  and  $y$  are considered equal if  $x$  is not less than  $y$  and  $y$  is not less than  $x$ .

### A.2.4 Algorithms That Write Container Elements

Many algorithms write new values to the elements in the given sequence. These algorithms can be distinguished from one another both by the kinds of iterators they use to denote their input sequence and by whether they write elements in the input range or write to a given destination.

## Algorithms That Write but Do Not Read Elements

These algorithms require an *output iterator* that denotes a destination. The `_n` versions take a second argument that specifies a count and write the given number of elements to the destination.

```
fill(beg, end, val)
fill_n(dest, cnt, val)
generate(beg, end, Gen)
generate_n(dest, cnt, Gen)
```

Assigns a new value to each element in the input sequence. `fill` assigns the value `val`; `generate` executes the generator object `Gen()`. A generator is a callable object (§ 10.3.2, p. 388) that is expected to produce a different return value each time it is called. `fill` and `generate` return `void`. The `_n` versions return an iterator that refers to the position immediately following the last element written to the output sequence.

## Write Algorithms with Input Iterators

Each of these algorithms reads an input sequence and writes to an output sequence. They require `dest` to be an *output iterator*, and the iterators denoting the input range must be *input iterators*.

```
copy(beg, end, dest)
copy_if(beg, end, dest, unaryPred)
copy_n(beg, n, dest)
```

Copies from the input range to the sequence denoted by `dest`. `copy` copies all elements, `copy_if` copies those for which `unaryPred` succeeds, and `copy_n` copies the first `n` elements. The input sequence must have at least `n` elements.

```
move(beg, end, dest)
```

Calls `std::move` (§ 13.6.1, p. 533) on each element in the input sequence to move that element to the sequence beginning at iterator `dest`.

```
transform(beg, end, dest, unaryOp)
transform(beg, end, beg2, dest, binaryOp)
```

Calls the given operation and writes the result of that operation to `dest`. The first version applies a unary operation to each element in the input range. The second applies a binary operation to elements from the two input sequences.

```
replace_copy(beg, end, dest, old_val, new_val)
replace_copy_if(beg, end, dest, unaryPred, new_val)
```

Copies each element to `dest`, replacing the specified elements with `new_val`. The first version replaces those elements that are `== old_val`. The second version replaces those elements for which `unaryPred` succeeds.

```
merge(beg1, end1, beg2, end2, dest)
merge(beg1, end1, beg2, end2, dest, comp)
```

Both input sequences must be sorted. Writes a merged sequence to `dest`. The first version compares elements using the `<` operator; the second version uses the given comparison operation.

## Write Algorithms with Forward Iterators

These algorithms require *forward iterators* because they write to elements in their input sequence. The iterators must give write access to the elements.

```
iter_swap(iterator1, iterator2)
swap_ranges(iterator1, iterator2, iterator3)
```

Swaps the element denoted by `iterator1` with the one denoted by `iterator2`; or swaps all of the elements in the input range with those in the second sequence beginning at `iterator3`. The ranges must not overlap. `iter_swap` returns `void`; `swap_ranges` returns `iterator3` incremented to denote the element just after the last one swapped.

```
replace(iterator, iterator, value)
replace_if(iterator, iterator, unaryPredicate, value)
```

Replaces each matching element with `value`. The first version uses `==` to compare elements with `old_val`; the second version replaces those elements for which `unaryPred` succeeds.

## Write Algorithms with Bidirectional Iterators

These algorithms require the ability to go backward in the sequence, so they require *bidirectional iterators*.

```
copy_backward(iterator, iterator, destination)
move_backward(iterator, iterator, destination)
```

Copies or moves elements from the input range to the given destination. Unlike other algorithms, `destination` is the off-the-end iterator for the output sequence (i.e., the destination sequence will end immediately *before* `destination`). The last element in the input range is copied or moved to the last element in the destination, then the second-to-last element is copied/moved, and so on. Elements in the destination have the same order as those in the input range. If the range is empty, the return value is `destination`; otherwise, the return denotes the element that was copied or moved from `*iterator`.

```
inplace_merge(iterator, iterator, iterator)
inplace_merge(iterator, iterator, iterator, comparisonFunction)
```

Merges two sorted subsequences from the same sequence into a single, ordered sequence. The subsequences from `iterator` to `mid` and from `mid` to `end` are merged and written back into the original sequence. The first version uses `<` to compare elements; the second version uses a given comparison operation. Returns `void`.

### A.2.5 Partitioning and Sorting Algorithms

The sorting and partitioning algorithms provide various strategies for ordering the elements of a sequence.

Each of the sorting and partitioning algorithms provides stable and unstable versions (§ 10.3.1, p. 387). A stable algorithm maintains the relative order of equal elements. The stable algorithms do more work and so may run more slowly and use more memory than the unstable counterparts.

## Partitioning Algorithms

A partition divides elements in the input range into two groups. The first group consists of those elements that satisfy the specified predicate; the second, those that do not. For example, we can partition elements in a sequence based on whether the elements are odd, or on whether a word begins with a capital letter, and so forth. These algorithms require *bidirectional iterators*.

**is\_partitioned(beg, end, unaryPred)**

Returns true if all the elements for which unaryPred succeeds precede those for which unaryPred is false. Also returns true if the sequence is empty.

**partition\_copy(beg, end, dest1, dest2, unaryPred)**

Copies elements for which unaryPred succeeds to dest1 and copies those for which unaryPred fails to dest2. Returns a pair (§ 11.2.3, p. 426) of iterators. The first member denotes the end of the elements copied to dest1, and the second denotes the end of the elements copied to dest2. The input sequence may not overlap either of the destination sequences.

**partition\_point(beg, end, unaryPred)**

The input sequence must be partitioned by unaryPred. Returns an iterator one past the subrange for which unaryPred succeeds. If the returned iterator is not end, then unaryPred must be false for the returned iterator and for all elements that follow that point.

**stable\_partition(beg, end, unaryPred)**

**partition(beg, end, unaryPred)**

Uses unaryPred to partition the input sequence. Elements for which unaryPred succeeds are put at the beginning of the sequence; those for which the predicate is false are at the end. Returns an iterator just past the last element for which unaryPred succeeds, or beg if there are no such elements.

## Sorting Algorithms

These algorithms require *random-access iterators*. Each of the sorting algorithms provides two overloaded versions. One version uses the element's operator `<` to compare elements; the other takes an extra parameter that specifies an ordering relation (§ 11.2.2, p. 425). `partial_sort_copy` returns an iterator into the destination; the other sorting algorithms return `void`.

The `partial_sort` and `nth_element` algorithms do only part of the job of sorting the sequence. They are often used to solve problems that might otherwise be handled by sorting the entire sequence. Because these algorithms do less work, they typically are faster than sorting the entire input range.

**sort(beg, end)**

**stable\_sort(beg, end)**

**sort(beg, end, comp)**

**stable\_sort(beg, end, comp)**

Sorts the entire range.

```
is_sorted(beg, end)
is_sorted(beg, end, comp)
is_sorted_until(beg, end)
is_sorted_until(beg, end, comp)
```

`is_sorted` returns a bool indicating whether the entire input sequence is sorted. `is_sorted_until` finds the longest initial sorted subsequence in the input and returns an iterator just after the last element of that subsequence.

```
partial_sort(beg, mid, end)
partial_sort(beg, mid, end, comp)
```

`Sorts` a number of elements equal to `mid - beg`. That is, if `mid - beg` is equal to 42, then this function puts the lowest-valued elements in sorted order in the first 42 positions in the sequence. After `partial_sort` completes, the elements in the range from `beg` up to but not including `mid` are sorted. No element in the sorted range is larger than any element in the range after `mid`. The order among the unsorted elements is unspecified.

```
partial_sort_copy(beg, end, destBeg, destEnd)
partial_sort_copy(beg, end, destBeg, destEnd, comp)
```

`Sorts` elements from the input range and puts as much of the sorted sequence as fits into the sequence denoted by the iterators `destBeg` and `destEnd`. If the destination range is the same size or has more elements than the input range, then the entire input range is sorted and stored starting at `destBeg`. If the destination size is smaller, then only as many sorted elements as will fit are copied.

Returns an iterator into the destination that refers just past the last element that was sorted. The returned iterator will be `destEnd` if that destination sequence is smaller than or equal in size to the input range.

```
nth_element(beg, nth, end)
nth_element(beg, nth, end, comp)
```

The argument `nth` must be an iterator positioned on an element in the input sequence. After `nth_element`, the element denoted by that iterator has the value that would be there if the entire sequence were sorted. The elements in the sequence are partitioned around `nth`: Those before `nth` are all smaller than or equal to the value denoted by `nth`, and the ones after it are greater than or equal to it.

## A.2.6 General Reordering Operations

Several algorithms reorder the elements of the input sequence. The first two, `remove` and `unique`, reorder the sequence so that the elements in the first part of the sequence meet some criteria. They return an iterator marking the end of this subsequence. Others, such as `reverse`, `rotate`, and `random_shuffle`, rearrange the entire sequence.

The base versions of these algorithms operate “in place”; they rearrange the elements in the input sequence itself. Three of the reordering algorithms offer “copying” versions. These `_copy` versions perform the same reordering but write the reordered elements to a specified destination sequence rather than changing the input sequence. These algorithms require *output iterator* for the destination.

## Reordering Algorithms Using Forward Iterators

These algorithms reorder the input sequence. They require that the iterators be at least *forward iterators*.

```
remove(beg, end, val)
remove_if(beg, end, unaryPred)
remove_copy(beg, end, dest, val)
remove_copy_if(beg, end, dest, unaryPred)
```

“Removes” elements from the sequence by overwriting them with elements that are to be kept. The removed elements are those that are == val or for which unaryPred succeeds. Returns an iterator just past the last element that was not removed.

```
unique(beg, end)
unique(beg, end, binaryPred)
unique_copy(beg, end, dest)
unique_copy_if(beg, end, dest, binaryPred)
```

Reorders the sequence so that adjacent duplicate elements are “removed” by overwriting them. Returns an iterator just past the last unique element. The first version uses == to determine whether two elements are the same; the second version uses the predicate to test adjacent elements.

```
rotate(beg, mid, end)
rotate_copy(beg, mid, end, dest)
```

Rotates the elements around the element denoted by mid. The element at mid becomes the first element; elements from mid + 1 up to but not including end come next, followed by the range from beg up to but not including mid. Returns an iterator denoting the element that was originally at beg.

## Reordering Algorithms Using Bidirectional Iterators

Because these algorithms process the input sequence backward, they require *bidi-directional iterators*.

```
reverse(beg, end)
reverse_copy(beg, end, dest)
```

Reverses the elements in the sequence. reverse returns void; reverse\_copy returns an iterator just past the element copied to the destination.

## Reordering Algorithms Using Random-Access Iterators

Because these algorithms rearrange the elements in a random order, they require *random-access iterators*.

```
random_shuffle(beg, end)
random_shuffle(beg, end, rand)
shuffle(beg, end, Uniform_rand)
```

Shuffles the elements in the input sequence. The second version takes a callable that must take a positive integer value and produce a uniformly distributed random integer in the exclusive range from 0 to the given value. The third argument

to `shuffle` must meet the requirements of a uniform random number generator (§ 17.4, p. 745). All three versions return `void`.

### A.2.7 Permutation Algorithms

The permutation algorithms generate lexicographical permutations of a sequence. These algorithms reorder a permutation to produce the (lexicographically) next or previous permutation of the given sequence. They return a `bool` that indicates whether there was a next or previous permutation.

To understand what is meant by next or previous permutation, consider the following sequence of three characters: `abc`. There are six possible permutations on this sequence: `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`. These permutations are listed in lexicographical order based on the less-than operator. That is, `abc` is the first permutation because its first element is less than or equal to the first element in every other permutation, and its second element is smaller than any permutation sharing the same first element. Similarly, `acb` is the next permutation because it begins with `a`, which is smaller than the first element in any remaining permutation. Permutations that begin with `b` come before those that begin with `c`.

For any given permutation, we can say which permutation comes before it and which after it, assuming a particular ordering between individual elements. Given the permutation `bca`, we can say that its previous permutation is `bac` and that its next permutation is `cab`. There is no previous permutation of the sequence `abc`, nor is there a next permutation of `cba`.

These algorithms assume that the elements in the sequence are unique. That is, the algorithms assume that no two elements in the sequence have the same value.

To produce the permutation, the sequence must be processed both forward and backward, thus requiring *bidirectional iterators*.

```
is_permutation(beg1, end1, beg2)
is_permutation(beg1, end1, beg2, binaryPred)
```

Returns `true` if there is a permutation of the second sequence with the same number of elements as are in the first sequence and for which the elements in the permutation and in the input sequence are equal. The first version compares elements using `==`; the second uses the given `binaryPred`.

```
next_permutation(beg, end)
next_permutation(beg, end, comp)
```

If the sequence is already in its last permutation, then `next_permutation` reorders the sequence to be the lowest permutation and returns `false`. Otherwise, it transforms the input sequence into the lexicographically next ordered sequence, and returns `true`. The first version uses the element's `<` operator to compare elements; the second version uses the given comparison operation.

```
prev_permutation(beg, end)
prev_permutation(beg, end, comp)
```

Like `next_permutation`, but transforms the sequence to form the previous permutation. If this is the smallest permutation, then it reorders the sequence to be the largest permutation and returns `false`.

## A.2.8 Set Algorithms for Sorted Sequences

The set algorithms implement general set operations on a sequence that is in sorted order. These algorithms are distinct from the library `set` container and should not be confused with operations on `sets`. Instead, these algorithms provide setlike behavior on an ordinary sequential container (`vector`, `list`, etc.) or other sequence, such as an input stream.

These algorithms process elements sequentially, requiring *input iterators*. With the exception of `includes`, they also take an *output iterator* denoting a destination. These algorithms return their `dest` iterator incremented to denote the element just after the last one that was written to `dest`.

Each algorithm is overloaded. The first version uses the `<` operator for the element type. The second uses a given comparison operation.

`includes(beg, end, beg2, end2)`

`includes(beg, end, beg2, end2, comp)`

Returns `true` if every element in the second sequence is contained in the input sequence. Returns `false` otherwise.

`set_union(beg, end, beg2, end2, dest)`

`set_union(beg, end, beg2, end2, dest, comp)`

Creates a sorted sequence of the elements that are in either sequence. Elements that are in both sequences occur in the output sequence only once. Stores the sequence in `dest`.

`set_intersection(beg, end, beg2, end2, dest)`

`set_intersection(beg, end, beg2, end2, dest, comp)`

Creates a sorted sequence of elements present in both sequences. Stores the sequence in `dest`.

`set_difference(beg, end, beg2, end2, dest)`

`set_difference(beg, end, beg2, end2, dest, comp)`

Creates a sorted sequence of elements present in the first sequence but not in the second.

`set_symmetric_difference(beg, end, beg2, end2, dest)`

`set_symmetric_difference(beg, end, beg2, end2, dest, comp)`

Creates a sorted sequence of elements present in either sequence but not in both.

## A.2.9 Minimum and Maximum Values

These algorithms use either the `<` operator for the element type or the given comparison operation. The algorithms in the first group operate on values rather than sequences. The algorithms in the second set take a sequence that is denoted by *input iterators*.

`min(val1, val2)`

`min(val1, val2, comp)`

`min(init_list)`

`min(init_list, comp)`

```
max(val1, val2)
max(val1, val2, comp)
max(initializer_list)
max(initializer_list, comp)
```

Returns the minimum/maximum of val1 and val2 or the minimum/maximum value in the initializer\_list. The arguments must have exactly the same type as each other. Arguments and the return type are both references to const, meaning that objects are not copied.

```
minmax(val1, val2)
minmax(val1, val2, comp)
minmax(initializer_list)
minmax(initializer_list, comp)
```

Returns a pair (§ 11.2.3, p. 426) where the first member is the smaller of the supplied values and the second is the larger. The initializer\_list version returns a pair in which the first member is the smallest value in the list and the second member is the largest.

```
min_element(beg, end)
min_element(beg, end, comp)
max_element(beg, end)
max_element(beg, end, comp)
minmax_element(beg, end)
minmax_element(beg, end, comp)
```

min\_element and max\_element return iterators referring to the smallest and largest element in the input sequence, respectively. minmax\_element returns a pair whose first member is the smallest element and whose second member is the largest.

## Lexicographical Comparison

This algorithm compares two sequences based on the first unequal pair of elements. Uses either the < operator for the element type or the given comparison operation. Both sequences are denoted by *input iterators*.

```
lexicographical_compare(beg1, end1, beg2, end2)
lexicographical_compare(beg1, end1, beg2, end2, comp)
```

Returns true if the first sequence is lexicographically less than the second. Otherwise, returns false. If one sequence is shorter than the other and all its elements match the corresponding elements in the longer sequence, then the shorter sequence is lexicographically smaller. If the sequences are the same size and the corresponding elements match, then neither is lexicographically less than the other.

### A.2.10 Numeric Algorithms

The numeric algorithms are defined in the numeric header. These algorithms require *input iterators*; if the algorithm writes output, it uses an *output iterator* for the destination.

```
accumulate(beg, end, init)
accumulate(beg, end, init, binaryOp)
```

Returns the sum of all the values in the input range. The summation starts with the initial value specified by `init`. The return type is the same type as the type of `init`. The first version applies the `+` operator for the element type; the second version applies the specified binary operation.

```
inner_product(beg1, end1, beg2, init)
inner_product(beg1, end1, beg2, init, binOp1, binOp2)
```

Returns the sum of the elements generated as the product of two sequences. The two sequences are processed in tandem, and the elements from each sequence are multiplied. The product of that multiplication is summed. The initial value of the sum is specified by `init`. The type of `init` determines the return type.

The first version uses the element's multiplication (`*`) and addition (`+`) operators. The second version applies the specified binary operations, using the first operation in place of addition and the second in place of multiplication.

```
partial_sum(beg, end, dest)
partial_sum(beg, end, dest, binaryOp)
```

Writes a new sequence to `dest` in which the value of each new element represents the sum of all the previous elements up to and including its position within the input range. The first version uses the `+` operator for the element type; the second version applies the specified binary operation. Returns the `dest` iterator incremented to refer just past the last element written.

```
adjacent_difference(beg, end, dest)
adjacent_difference(beg, end, dest, binaryOp)
```

Writes a new sequence to `dest` in which the value of each new element other than the first represents the difference between the current and previous elements. The first version uses the element type's `-` operation; the second version applies the specified binary operation.

```
iota(beg, end, val)
```

Assigns `val` to the first element and increments `val`. Assigns the incremented value to the next element, and again increments `val`, and assigns the incremented value to the next element in the sequence. Continues incrementing `val` and assigning its new value to successive elements in the input sequence.

## A.3 Random Numbers

The library defines a collection of random number engine classes and adaptors that use differing mathematical approaches to generating pseudorandom numbers. The library also defines a collection of distribution templates that provide numbers according to various probability distributions. Both the engines and the distributions have names that correspond to their mathematical properties.

The specifics of how these classes generate numbers is well beyond the scope of this Primer. In this section, we'll list the engine and distribution types, but the reader will need to consult other resources to learn how to use these types.

### A.3.1 Random Number Distributions

With the exception of the `bernouilli_distribution`, which always generates type `bool`, the distribution types are templates. Each of these templates takes a single type parameter that names the result type that the distribution will generate.

The distribution classes differ from other class templates we've used in that the distribution types place restrictions on the types we can specify for the template type. Some distribution templates can be used to generate only floating-point numbers; others can be used to generate only integers.

In the following descriptions, we indicate whether a distribution generates floating-point numbers by specifying the type as `template_name<RealT>`. For these templates, we can use `float`, `double`, or `long double` in place of `RealT`. Similarly, `IntT` requires one of the built-in integral types, not including `bool` or any of the `char` types. The types that can be used in place of `IntT` are `short`, `int`, `long`, `long long`, `unsigned short`, `unsigned int`, `unsigned long`, or `unsigned long long`.

The distribution templates define a default template type parameter (§ 17.4.2, p. 750). The default for the integral distributions is `int`; the default for the classes that generate floating-point numbers is `double`.

The constructors for each distribution has parameters that are specific to the kind of distribution. Some of these parameters specify the range of the distribution. These ranges are always *inclusive*, unlike iterator ranges.

## Uniform Distributions

```
uniform_int_distribution<IntT> u(m, n);  
uniform_real_distribution<RealT> u(x, y);
```

Generates values of the specified type in the given inclusive range. `m` (or `x`) is the smallest number that can be returned; `n` (or `y`) is the largest. `m` defaults to 0; `n` defaults to the maximum value that can be represented in an object of type `IntT`. `x` defaults to 0.0 and `y` defaults to 1.0.

## Bernoulli Distributions

```
bernoulli_distribution b(p);
```

Yields `true` with given probability `p`; `p` defaults to 0.5.

```
binomial_distribution<IntT> b(t, p);
```

Distribution computed for a sample size that is the integral value `t`, with probability `p`; `t` defaults to 1 and `p` defaults to 0.5.

```
geometric_distribution<IntT> g(p);
```

Per-trial probability of success `p`; `p` defaults to 0.5.

```
negative_binomial_distribution<IntT> nb(k, p);
```

Integral value `k` trials with probability of success `p`; `k` defaults to 1 and `p` to 0.5.

## Poisson Distributions

```
poisson_distribution<IntT> p(x);
```

Distribution around double mean `x`.

```
exponential_distribution<RealT> e(lam);
Floating-point valued lambda lam; lam defaults to 1.0.

gamma_distribution<RealT> g(a, b);
With alpha (shape) a and beta (scale) b; both default to 1.0.

weibull_distribution<RealT> w(a, b);
With shape a and scale b; both default to 1.0.

extreme_value_distribution<RealT> e(a, b);
a defaults to 0.0 and b defaults to 1.0.
```

## Normal Distributions

```
normal_distribution<RealT> n(m, s);
Mean m and standard deviation s; m defaults to 0.0, s to 1.0.

lognormal_distribution<RealT> ln(m, s);
Mean m and standard deviation s; m defaults to 0.0, s to 1.0.

chi_squared_distribution<RealT> c(x);
x degrees of freedom; defaults to 1.0.

cauchy_distribution<RealT> c(a, b);
Location a and scale b default to 0.0 and 1.0, respectively.

fisher_f_distribution<RealT> f(m, n);
m and n degrees of freedom; both default to 1.

student_t_distribution<RealT> s(n);
n degrees of freedom; n defaults to 1.
```

## Sampling Distributions

```
discrete_distribution<IntT> d(i, j);
discrete_distribution<IntT> d{il};
i and j are input iterators to a sequence of weights; il is a braced list of weights.
The weights must be convertible to double.

piecewise_constant_distribution<RealT> pc(b, e, w);
b, e, and w are input iterators.

piecewise_linear_distribution<RealT> pl(b, e, w);
b, e, and w are input iterators.
```

## A.3.2 Random Number Engines

The library defines three classes that implement different algorithms for generating random numbers. The library also defines three adaptors that modify the sequences produced by a given engine. The engine and engine adaptor classes are templates. Unlike the parameters to the distributions, the parameters to these engines are complex and require detailed understanding of the math used by the

particular engine. We list the engines here so that the reader is aware of their existence, but describing how to generate these types is beyond the scope of this Primer.

The library also defines several types that are built from the engines or adaptors. The `default_random_engine` type is a type alias for one of the engine types parameterized by variables designed to yield good performance for casual use. The library also defines several classes that are fully specialized versions of an engine or adaptor. The engines and the specializations defined by the library are:

**default\_random\_engine**

Type alias for one of the other engines intended to be used for most purposes.

**linear\_congruential\_engine**

`minstd_rand0` Has a multiplier of 16807, a modulus of 2147483647, and an increment of 0.

`minstd_rand` Has a multiplier of 48271, a modulus of 2147483647, and an increment of 0.

**mersenne\_twister\_engine**

`mt19937` 32-bit unsigned Mersenne twister generator.

`mt19937_64` 64-bit unsigned Mersenne twister generator.

**subtract\_with\_carry\_engine**

`ranlux24_base` 32-bit unsigned subtract with carry generator.

`ranlux48_base` 64-bit unsigned subtract with carry generator.

**discard\_block\_engine**

Engine adaptor that discards results from its underlying engine. Parameterized by the underlying engine to use the block size, and size of the used blocks.

`ranlux24` Uses the `ranlux24_base` engine with a block size of 223 and a used block size of 23.

`ranlux48` Uses the `ranlux48_base` engine with a block size of 389 and a used block size of 11.

**independent\_bits\_engine**

Engine adaptor that generates numbers with a specified number of bits. Parameterized by the underlying engine to use, the number of bits to generate in its results, and an unsigned integral type to use to hold the generated bits. The number of bits specified must be less than the number of digits that the specified unsigned type can hold.

**shuffle\_order\_engine**

Engine adaptor that returns the same numbers as its underlying engine but delivers them in a different sequence. Parameterized by the underlying engine to use and the number of elements to shuffle.

`knuth_b` Uses the `minstd_rand0` engine with a table size of 256.

*This page intentionally left blank*

# Index

**Bold face** numbers refer to the page on which the term was first defined.  
Numbers in *italic* refer to the “Defined Terms” section in which the term is defined.

## What’s new in C++11

- = default, 265, 506
  - = delete, 507
  - allocator, construct forwards to any constructor, 482
  - array container, 327
  - auto, 68
    - for type abbreviation, 88, 129
    - not with dynamic array, 478
    - with dynamic object, 459
  - begin function, 118
  - bind function, 397
  - bitset enhancements, 726
  - constexpr
    - constructor, 299
    - function, 239
    - variable, 66
  - container
    - cbegin and cend, 109, 334
    - emplace members, 345
    - insert return type, 344
    - nonmember swap, 339
    - of container, 97, 329
    - shrink\_to\_fit, 357
  - decltype, 70
    - function return type, 250
  - delegating constructor, 291
  - deleted copy-control, 624
  - division rounding, 141
  - end function, 118
  - enumeration
    - controlling representation, 834
    - forward declaration, 834
    - scoped, 832
  - explicit conversion operator, 582
  - explicit instantiation, 675
  - final class, 600
- format control for floating-point, 757
  - forward function, 694
  - forward\_list container, 327
  - function interface to callable objects, 577
  - in-class initializer, 73, 274
  - inherited constructor, 628, 804
  - initializer\_list, 220
  - inline namespace, 790
  - lambda expression, 388
  - list initialization
    - = (assignment), 145
    - container, 336, 423
    - dynamic array, 478
    - dynamic object, 459
    - pair, 431
    - return value, 226, 427
    - variable, 43
    - vector, 98
  - long long, 33
  - mem\_fn function, 843
  - move function, 533
  - move avoids copies, 529
  - move constructor, 534
  - move iterator, 543
  - move-enabled this pointer, 546
  - noexcept
    - exception specification, 535, 779
    - operator, 780
  - nullptr, 54
  - random-number library, 745
  - range for statement, 91, 187
    - not with dynamic array, 477
  - regular expression-library, 728
  - rvalue reference, 532
    - cast from lvalue, 691
    - reference collapsing, 688
  - sizeof data member, 157
  - sizeof... operator, 700

smart pointer, 450  
     `shared_ptr`, 450  
     `unique_ptr`, 470  
     `weak_ptr`, 473  
**string**  
     numeric conversions, 367  
     parameter with IO types, 317  
**template**  
     function template default template  
         argument, 670  
     type alias, 666  
     type parameter as friend, 666  
     variadic, 699  
     variadics and forwarding, 704  
**trailing return type**, 229  
     in function template, 684  
     in lambda expression, 396  
**tuple**, 718  
**type alias declaration**, 68  
**union member of class type**, 848  
**unordered containers**, 443  
**virtual function**  
     `final`, 606  
     `override`, 596, 606

**Symbols**

. . . (ellipsis parameter), 222  
`/* */` (block comment), 9, 26  
`//` (single-line comment), 9, 26  
`= default`, 265, 306  
     copy-control members, 506  
     default constructor, 265  
`= delete`, 507  
     copy control, 507–508  
     default constructor, 507  
     function matching, 508  
     move operations, 538  
`--DATE--`, 242  
`--FILE--`, 242  
`--LINE--`, 242  
`--TIME--`, 242  
`--cplusplus`, 860  
`\0` (null character), 39  
`\Xnnn` (hexadecimal escape sequence), 39  
`\n` (newline character), 39  
`\t` (tab character), 39  
`\nnn` (octal escape sequence), 39  
`{ }` (curly brace), 2, 26  
`#include`, 6, 28  
     standard header, 6

user-defined header, 21  
`#define`, 77, 80  
`#endif`, 77, 80  
`#ifdef`, 77, 80  
`#ifndef`, 77, 80  
`~classname`, see destructor  
`;` (semicolon), 3  
     class definition, 73  
     null statement, 172  
`++` (increment), 12, 28, 147–149, 170  
     iterator, 107, 132  
     overloaded operator, 566–568  
     pointer, 118  
     precedence and associativity, 148  
     reverse iterator, 407  
     `StrBlobPtr`, 566  
`--` (decrement), 13, 28, 147–149, 170  
     iterator, 107  
     overloaded operator, 566–568  
     pointer, 118  
     precedence and associativity, 148  
     reverse iterator, 407, 408  
     `StrBlobPtr`, 566  
`*` (dereference), 53, 80, 448  
     iterator, 107  
     map iterators, 429  
     overloaded operator, 569  
     pointer, 53  
     precedence and associativity, 148  
     smart pointer, 451  
     `StrBlobPtr`, 569  
`&` (address-of), 52, 80  
     overloaded operator, 554  
`->` (arrow operator), 110, 132, 150  
     overloaded operator, 569  
     `StrBlobPtr`, 569  
`.` (dot), 23, 28, 150  
`->*` (pointer to member arrow), 837  
`.*` (pointer to member dot), 837  
`[ ]` (subscript), 93  
     array, 116, 132  
     array, 347  
     `bitset`, 727  
     `deque`, 347  
     does not add elements, 104  
     `map`, and `unordered_map`, 435, 448  
         adds element, 435  
     multidimensional array, 127  
     out-of-range index, 93  
     overloaded operator, 564  
     pointer, 121

- string, 93, 132, 347
- StrVec, 565
- subscript range, 95
- vector, 103, 132, 347
- () (call operator), 23, 28, 202, 252
  - absInt, 571
  - const member function, 573
  - execution flow, 203
  - overloaded operator, 571
  - PrintString, 571
  - ShorterString, 573
  - SizeComp, 573
- :: (scope operator), 8, 28, 82
  - base-class member, 607
  - class type member, 88, 282
  - container, type members, 333
  - global namespace, 789, 818
  - member function, definition, 259
  - overrides name lookup, 286
- = (assignment), 12, 28, 144–147
  - see also* copy assignment
  - see also* move assignment
  - associativity, 145
  - base from derived, 603
  - container, 89, 103, 337
  - conversion, 145, 159
  - derived class, 626
  - in condition, 146
  - initializer\_list, 563
  - list initialization, 145
  - low precedence, 146
  - multiple inheritance, 805
  - overloaded operator, 500, 563
  - pointer, 55
  - to signed, 35
  - to unsigned, 35
  - vs. == (equality), 146
  - vs. initialization, 42
- + = (compound assignment), 12, 28, 147
  - bitwise operators, 155
  - iterator, 111
  - overloaded operator, 555, 560
  - Sales\_data, 564
    - exception version, 784
    - string, 89
- + (addition), 6, 140
  - iterator, 111
  - pointer, 119
  - Sales\_data, 560
    - exception version, 784
  - Sales\_item, 22
- SmallInt, 588
- string, 89
- (subtraction), 140
  - iterator, 111
  - pointer, 119
- \* (multiplication), 140
- / (division), 140
  - rounding, 141
- % (modulus), 141
  - grading program, 176
- == (equality), 18, 28
  - arithmetic conversion, 144
  - container, 88, 102, 340, 341
  - iterator, 106, 107
  - overloaded operator, 561, 562
  - pointer, 55, 120
  - Sales\_data, 561
  - string, 88
  - tuple, 720
  - unordered container key\_type, 443
  - used in algorithms, 377, 385, 413
  - vs. != (assignment), 146
- != (inequality), 28
  - arithmetic conversion, 144
  - container, 88, 102, 340, 341
  - iterator, 106, 107
  - overloaded operator, 562
  - pointer, 55, 120
  - Sales\_data, 561
  - string, 88
  - tuple, 720
- < (less-than), 28, 143
  - container, 88, 340
  - ordered container key\_type, 425
  - overloaded operator, 562
    - strict weak ordering, 562
  - string, 88
  - tuple, 720
  - used in algorithms, 378, 385, 413
- <= (less-than-or-equal), 12, 28, 143
  - container, 88, 340
  - string, 88
- > (greater-than), 28, 143
  - container, 88, 340
  - string, 88
- >= (greater-than-or-equal), 28, 143
  - container, 88, 340
  - string, 88
- >> (input operator), 8, 28
  - as condition, 15, 86, 312
  - chained-input, 8

istream, 8  
 istream\_iterator, 403  
 overloaded operator, 558–559  
 precedence and associativity, 155  
 Sales\_data, 558  
 Sales\_item, 21  
 string, 85, 132  
 << (output operator), 7, 28  
     bitset, 727  
     chained output, 7  
     ostream, 7  
     ostream\_iterator, 405  
     overloaded operator, 557–558  
     precedence and associativity, 155  
     Query, 641  
     Sales\_data, 557  
     Sales\_item, 21  
     string, 85, 132  
 >> (right-shift), 153, 170  
 << (left-shift), 153, 170  
 && (logical AND), 94, 132, 142, 169  
     order of evaluation, 138  
     overloaded operator, 554  
     short-circuit evaluation, 142  
 || (logical OR), 142  
     order of evaluation, 138  
     overloaded operator, 554  
     short-circuit evaluation, 142  
& (bitwise AND), 154, 169  
     Query, 638, 644  
! (logical NOT), 87, 132, 143, 170  
|| (logical OR), 132, 170  
| (bitwise OR), 154, 170  
     Query, 638, 644  
^ (bitwise XOR), 154, 170  
~ (bitwise NOT), 154, 170  
     Query, 638, 643  
, (comma operator), 157, 169  
     order of evaluation, 138  
     overloaded operator, 554  
?: (conditional operator), 151, 169  
     order of evaluation, 138  
     precedence and associativity, 151  
+ (unary plus), 140  
- (unary minus), 140  
L'c' (wchar\_t literal), 38  
ddd.dddL or ddd.ddd1 (long double literal), 41  
numEnum or numenum (double literal), 39  
numF or numf (float literal), 41  
numL or numl (long literal), 41  
numLL or numl1 (long long literal), 41  
numU or numu (unsigned literal), 41  
class member : *constant expression*, see bit-field

## A

absInt, 571  
     () (call operator), 571  
 abstract base class, 610, 649  
     BinaryQuery, 643  
     Disc\_quote, 610  
     Query\_base, 636  
 abstract data type, 254, 305  
 access control, 611–616  
     class derivation list, 596  
     default inheritance access, 616  
     default member access, 268  
     derived class, 613  
     derived-to-base conversion, 613  
     design, 614  
     inherited members, 612  
     local class, 853  
     nested class, 844  
     private, 268  
     protected, 595, 611  
     public, 268  
     using declaration, 615  
 access specifier, 268, 305  
 accessible, 611, 649  
     derived-to-base conversion, 613  
 Account, 301  
 accumulate, 379, 882  
     bookstore program, 406  
 Action, 839  
 adaptor, 372  
     back\_inserter, 402  
     container, 368, 368–371  
     front\_inserter, 402  
      inserter, 402  
     make\_move\_iterator, 543  
 add, Sales\_data, 261  
 add\_item, Basket, 633  
 add\_to\_Folder, Message, 522  
 address, 33, 78  
 adjacent\_difference, 882  
 adjacent\_find, 871  
 advice  
     always initialize a pointer, 54  
     avoid casts, 165

- avoid undefined behavior, 36  
choosing a built-in type, 34  
define small utility functions, 277  
define variables near first use, 48  
don't create unnecessary regex objects, 733  
forwarding parameter pattern, 706  
keep lambda captures simple, 394  
managing iterators, 331, 354  
prefix vs. postfix operators, 148  
rule of five, 541  
use move sparingly, 544  
use constructor initializer lists, 289  
when to use overloading, 233  
writing compound expressions, 139
- aggregate class, 298, 305  
    initialization, 298
- algorithm header, 376
- algorithms, 376, 418  
    *see also* Appendix A
- architecture
- \_copy versions, 383, 414
  - \_if versions, 414
- naming convention, 413–414  
operate on iterators not containers, 378  
overloading pattern, 414  
parameter pattern, 412–413  
read-only, 379–380  
reorder elements, 383–385, 414  
write elements, 380–383
- associative container and, 430  
bind as argument, 397  
can't change container size, 385  
element type requirements, 377  
function object arguments, 572  
*istream\_iterator*, 404  
iterator category, 410–412  
iterator range, 376  
lambda as argument, 391, 396  
library function object, 575  
*ostream\_iterator*, 404  
sort comparison, requires strict weak ordering, 425  
supplying comparison operation, 386, 413
  - function, 386
  - lambda, 389, 390

two input ranges, 413  
type independence, 377  
use element's == (equality), 385, 413

use element's < (less-than), 385, 413  
accumulate, 379
  - bookstore program, 406

copy, 382  
count, 378  
equal\_range, 722  
equal, 380  
fill\_n, 381  
fill, 380  
find\_if, 388, 397, 414  
find, 376  
for\_each, 391  
replace\_copy, 383  
replace, 383  
set\_intersection, 647  
sort, 384  
stable\_sort, 387  
transform, 396  
unique, 384

alias declaration

  - namespace, 792, 817
  - template type, 666
  - type, 68

all\_of, 871

alloc\_n\_copy, StrVec, 527

allocate, allocator, 481

allocator, 481, 481–483, 491, 524–531
  - allocate, 481, 527
  - compared to operator new, 823

construct, 482
  - forwards to constructor, 527

deallocate, 483, 528
  - compared to operator delete, 823

destroy, 482, 528

alternative operator name, 46

alternative\_sum, program, 682

ambiguous

  - conversion, 583–589
  - multiple inheritance, 806

function call, 234, 245, 251
  - multiple inheritance, 808

overloaded operator, 588

AndQuery, 637

  - class definition, 644
  - eval function, 646

anonymous union, 848, 862

any\_bitset, 726

any\_of, 871

app (file mode), 319

append, string, 362

- argc, 219  
 argument, 23, 26, 202, 251  
     array, 214–219  
         buffer overflow, 215  
         to pointer conversion, 214  
     C-style string, 216  
     conversion, function matching, 234  
     default, 236  
     forwarding, 704  
     initializes parameter, 203  
     iterator, 216  
     low-level const, 213  
     main function, 218  
     multidimensional array, 218  
     nonreference parameter, 209  
     pass by reference, 210, 252  
     pass by value, 209, 252  
         uses copy constructor, 498  
         uses move constructor, 539, 541  
     passing, 208–212  
     pointer, 214  
     reference parameter, 210, 214  
     reference to const, 211  
     top-level const, 212  
 argument list, 202  
 argument-dependent lookup, 797  
     move and forward, 798  
 argv, 219  
 arithmetic  
     conversion, 35, 159, 168  
         in equality and relational operators, 144  
     integral promotion, 160, 169  
     signed to unsigned, 34  
     to bool, 162  
     operators, 139  
         compound assignment (e.g., +=), 147  
         function object, 574  
         overloaded, 560  
     type, 32, 78  
         machine-dependent, 32  
 arithmetic (addition and subtraction)  
     iterators, 111, 131  
     pointers, 119, 132  
 array, 113–130  
     [ ] (subscript), 116, 132  
     argument and parameter, 214–219  
     argument conversion, 214  
     auto returns pointer, 117  
     begin function, 118  
     compound type, 113  
     conversion to pointer, 117, 161  
     function arguments, 214  
     template argument deduction, 679  
 decltype returns array type, 118  
 definition, 113  
 dimension, constant expression, 113  
 dynamically allocated, 476, 476–484  
     allocator, 481  
     can't use begin and end, 477  
     can't use range for statement, 477  
     delete [], 478  
     empty array, 478  
     new [], 477  
         shared\_ptr, 480  
         unique\_ptr, 479  
 elements and destructor, 502  
 end function, 118  
 initialization, 114  
 initializer of vector, 125  
 multidimensional, 125–130  
 no copy or assign, 114  
 of char initialization, 114  
 parameter  
     buffer overflow, 215  
     converted to pointer, 215  
     function template, 654  
     pointer to, 218  
     reference to, 217  
 return type, 204  
     trailing, 229  
     type alias, 229  
     decltype, 230  
 sizeof, 157  
 subscript range, 116  
 subscript type, 116  
 understanding complicated declarations, 115  
 array  
     *see also* container  
     *see also* sequential container  
     [ ] (subscript), 347  
     = (assignment), 337  
     assign, 338  
     copy initialization, 337  
     default initialization, 336  
     definition, 336  
     header, 329  
     initialization, 334–337  
     list initialization, 337  
     overview, 327  
     random-access iterator, 412

- swap, 339  
assert preprocessor macro, 241, 251  
assign  
    array, 338  
    invalidates iterator, 338  
    sequential container, 338  
    string, 362  
assignment, vs. initialization, 42, 288  
assignment operators, 144–147  
associative array, *see* map  
associative container, 420, 447  
    and library algorithms, 430  
    initialization, 423, 424  
    key\_type requirements, 425, 445  
members  
    begin, 430  
    count, 437, 438  
    emplace, 432  
    end, 430  
    equal\_range, 439  
    erase, 434  
    find, 437, 438  
    insert, 432  
    key\_type, 428, 447  
    mapped\_type, 428, 448  
    value\_type, 428, 448  
override default comparison, 425  
override default hash, 446  
overview, 423  
associativity, 134, 136–137, 168  
    = (assignment), 145  
    ?: (conditional operator), 151  
dot and dereference, 150  
increment and dereference, 148  
IO operator, 155  
overloaded operator, 553  
at  
    deque, 348  
    map, 435  
    string, 348  
    unordered\_map, 435  
    vector, 348  
ate (file mode), 319  
auto, 68, 78  
    cbegin, 109, 379  
    cend, 109, 379  
for type abbreviation, 88, 129  
of array, 117  
of reference, 69  
pointer to function, 249  
with new, 459  
auto\_ptr deprecated, 471  
automatic object, 205, 251  
    *see also* local variable  
    *see also* parameter  
    and destructor, 502  
avg\_price, Sales\_data, 259
- ## B
- back  
    queue, 371  
    sequential container, 346  
    StrBlob, 457  
back\_inserter, 382, 402, 417  
    requires push\_back, 382, 402  
bad, 313  
bad\_alloc, 197, 460  
bad\_cast, 197, 826  
bad\_typeid, 828  
badbit, 312  
base, reverse iterator, 409  
base class, 592, 649  
    *see also* virtual function  
abstract, 610, 649  
base-to-derived conversion, not automatic, 602  
can be a derived class, 600  
definition, 594  
derived-to-base conversion, 597  
    accessibility, 613  
    key concepts, 604  
    multiple inheritance, 805  
final, 600  
friendship not inherited, 614  
initialized or assigned from derived, 603  
member hidden by derived, 619  
member new and delete, 822  
multiple, *see* multiple inheritance  
must be complete type, 600  
protected member, 611  
scope, 617  
    inheritance, 617–621  
    multiple inheritance, 807  
    virtual function, 620  
static members, 599  
user of, 614  
virtual, *see* virtual base class  
virtual destructor, 622  
Basket, 631  
    add\_item, 633

total, 632  
 Bear, 803  
     virtual base class, 812  
 before\_begin, forward\_list, 351  
 begin  
     associative container, 430  
     container, 106, 131, 333, 372  
     function, 118, 131  
         not with dynamic array, 477  
     multidimensional array, 129  
     StrBlob, 475  
     StrVec, 526  
 bernoulli\_distribution, 752  
 best match, 234, 251  
     *see also* function matching  
 bidirectional iterator, 412, 417  
 biggies program, 391  
 binary (file mode), 319  
 binary operators, 134, 168  
     overloaded operator, 552  
 binary predicate, 386, 417  
 binary\_function deprecated, 579  
 binary\_search, 873  
 BinaryQuery, 637  
     abstract base class, 643  
 bind, 397, 417  
     check\_size, 398  
     generates callable object, 397  
         from pointer to member, 843  
     placeholders, 399  
     reference parameter, 400  
 bind1st deprecated, 401  
 bind2nd deprecated, 401  
 binops desk calculator, 577  
 bit-field, 854, 862  
     access to, 855  
     constant expression, 854  
 bitset, 723, 723–728, 769  
     [ ] (subscript), 727  
     << (output operator), 727  
 any, 726  
 count, 727  
 flip, 727  
 grading program, 728  
 header, 723  
 initialization, 723–725  
     from string, 724  
     from unsigned, 723  
 none, 726  
 reset, 727  
 set, 727  
     test, 727  
     to\_ulong, 727  
 bitwise, bitset, operators, 725  
 bitwise operators, 152–156  
     += (compound assignment), 155  
     compound assignment (e.g., +=), 147  
     grading program, 154  
     operand requirements, 152  
 Blob  
     class template, 659  
     constructor, 662  
         initializer\_list, 662  
         iterator parameters, 673  
     instantiation, 660  
     member functions, 661–662  
 block, 2, 12, 26, 173, 199  
     function, 204  
     scope, 48, 80, 173  
     try, 193, 194, 200, 818  
 block /\* \*/, comment, 9, 26  
 book from author program, 438–440  
 bookstore program  
     Sales\_data, 255  
     using algorithms, 406  
     Sales\_item, 24  
 bool, 32  
     conversion, 35  
     literal, 41  
         in condition, 143  
 boolalpha, manipulator, 754  
 brace, curly, 2, 26  
 braced list, *see* list initialization  
 break statement, 190, 199  
     in switch, 179–181  
 bucket management, unordered container, 444  
 buffer, 7, 26  
     flushing, 314  
 buffer overflow, 105, 116, 131  
     array parameter, 215  
     C-style string, 123  
 buildMap program, 442  
 built-in type, 2, 26, 32–34  
     default initialization, 43  
 Bulk\_quote  
     class definition, 596  
     constructor, 598, 610  
     derived from Disc\_quote, 610  
     design, 592  
     synthesized copy control, 623  
 byte, 33, 78

# C

- .C file, 4
- .cc file, 4
- .cpp file, 4
- .cp file, 4
- C library header, 91
- C-style cast, 164
- C-style string, 114, **122**, 122–123, 131
  - buffer overflow, 123
  - initialization, 122
  - parameter, 216
  - string, 124
- c\_str, 124
- call by reference, 208, 210, 251
- call by value, **209**, 251
  - uses copy constructor, 498
  - uses move constructor, 539
- call signature, **576**, 590
- callable object, **388**, 417, 571–572
  - absInt, 571
  - bind, 397
  - call signature, 576
  - function and function pointers, 388
  - function objects, 572
- pointer to member
  - and bind, 843
  - and function, 842
  - and mem\_fn, 843
  - not callable, 842
- PrintString, 571
- ShorterString, 573
- SizeComp, 573
- with function, 576–579
- with algorithms, 390
- candidate function, **243**, 251
  - see also* function matching
  - function template, 695
  - namespace, 800
  - overloaded operator, 587
- capacity
  - string, 356
  - StrVec, 526
  - vector, 356
- capture list, *see lambda expression*
- case label, **179**, 179–182, 199
  - default, **181**
  - constant expression, 179
- case sensitive, string, 365
- cassert header, 241
- cast, *see also* named cast, 168
  - checked, *see dynamic\_cast*
  - old-style, 164
  - to rvalue reference, 691
- catch, **193**, 195, 199, **775**, 816
  - catch(...), 777, 816
  - exception declaration, 195, 200, 775, 816
- exception object, 775
- matching, 776
- ordering of, 776
- runtime\_error, 195
- catch all (catch(...)), 777, 816
- caution
  - ambiguous conversion operator, 581
  - conversions to unsigned, 37
  - dynamic memory pitfalls, 462
  - exception safety, 196
  - IO buffers, 315
  - overflow, 140
  - overloaded operator misuse, 555
  - overloaded operators and conversion operators, 586
  - smart pointer, pitfalls, 469
  - uninitialized variables, 45
  - using directives cause pollution, 795
- cbegin
  - auto, 109, 379
  - decltype, 109, 379
  - container, **109**, 333, 334, 372
- cctype
  - functions, 91–93
  - header, 91
- cend
  - auto, 109, 379
  - decltype, 109, 379
  - container, **109**, 333, 334, 372
- cerr, **6**, 26
- chained input, 8
- chained output, 7
- char, 32
  - signed, 34
  - unsigned, 34
  - array initialization, 114
  - literal, 39
  - representation, 34
- char16\_t, 33
- char32\_t, 33
- character
  - newline (\n), 39
  - nonprintable, **39**, 79
  - null (\0), 39

tab (\t), 39  
 character string literal, *see* string literal  
 check  
     StrBlob, 457  
     StrBlobPtr, 474  
 check\_size, 398  
     bind, 398  
 checked cast, *see* dynamic\_cast  
 children's story program, 383–391  
 chk\_n\_alloc, StrVec, 526  
 cin, 6, 26  
     tied to cout, 315  
 cl, 5  
 class, 19, 26, 72, 305  
     *see also* constructor  
     *see also* destructor  
     *see also* member function  
     *see also* static member  
 access specifier, 268  
     default, 268  
     private, 268, 306  
     public, 268, 306  
 aggregate, 298, 305  
 assignment operator  
     *see* copy assignment  
     *see* move assignment  
 base, *see* base class, 649  
 data member, 73, 78  
     const vs. mutable, 274  
     const, initialization, 289  
     in-class initializer, 274  
     initialization, 263, 274  
     must be complete type, 279  
     mutable, 274, 306  
     order of destruction, 502  
     order of initialization, 289  
     pointer, not deleted, 503  
     reference, initialization, 289  
     sizeof, 157  
 declaration, 278, 305  
 default inheritance specifier, 616  
 definition, 72, 256–267  
     ends with semicolon, 73  
 derived, *see* derived class, 649  
 exception, 193, 200  
 final specifier, 600  
 forward declaration, 279, 306  
 friend, 269, 280  
     class, 280  
     function, 269  
     member function, 280  
     overloaded function, 281  
     scope, 270, 281  
     template class or function, 664  
 implementation, 254  
 interface, 254  
 literal, 299  
 local, *see* local class  
 member, 73, 78  
 member access, 282  
 member new and delete, 822  
 member: *constant expression*, *see* bit-field  
 multiple base classes, *see* multiple inheritance  
 name lookup, 284  
 nested, *see* nested class  
 pointer to member, *see* pointer to member  
     preventing copies, 507  
     scope, 73, 282, 282–287, 305  
     synthesized, copy control, 267, 497,  
         500, 503, 537  
 template member, *see* member template  
 type member, 271  
     :: (scope operator), 282  
 user of, 255  
 valuelike, 512  
     without move constructor, 540  
 class  
     compared to typename, 654  
     default access specifier, 268  
     default inheritance specifier, 616  
     template parameter, 654  
 class derivation list, 596  
     access control, 612  
     default access specifier, 616  
     direct base class, 600  
     indirect base class, 600  
     multiple inheritance, 803  
     virtual base class, 812  
 class template, 96, 131, 658, 659, 658–667,  
     713  
     *see also* template parameter  
     *see also* instantiation  
     Blob, 659  
     declaration, 669  
     default template argument, 671  
     definition, 659  
     error detection, 657  
     explicit instantiation, 675, 675–676

explicit template argument, 660  
friend, 664  
    all instantiations, 665  
    declaration dependencies, 665  
    same instantiation, 664  
    specific instantiation, 665  
instantiation, 660  
member function  
    defined outside class body, 661  
    instantiation, 663  
member template, *see* member template  
specialization, 707, 709–712, 714  
    hash<key\_type>, 709, 788  
    member, 711  
    namespace, 788  
    partial, 711, 714  
static member, 667  
    accessed through an instantiation, 667  
    definition, 667  
template argument, 660  
template parameter, used in definition, 660  
type parameter as friend, 666  
type-dependent code, 658  
class type, 19, 26  
    conversion, 162, 305, 590  
    ambiguities, 587  
    conversion operator, 579  
    converting constructor, 294  
    impact on function matching, 584  
    overloaded function, 586  
    with standard conversion, 581  
default initialization, 44  
initialization, 73, 84, 262  
union member of, 848  
variable vs. function declaration, 294  
clear  
    sequential container, 350  
    stream, 313  
clog, 6, 26  
close, file stream, 318  
cmatch, 733  
cmath header, 751, 757  
collapsing rule, reference, 688  
combine, Sales\_data, 259  
comma (,) operator, 157  
comment, 9, 26  
    block /\* \*/, 9, 26  
    single-line (//), 9, 26  
compare  
    default template argument, 670  
function template, 652  
    default template argument, 670  
    explicit template argument, 683  
    specialization, 706  
    string literal version, 654  
    template argument deduction, 680  
    string, 366  
compareIsbn  
    and associative container, 426  
    Sales\_data, 387  
compilation  
    common errors, 16  
    compiler options, 207  
    conditional, 240  
    declaration vs. definition, 44  
    mixing C and C++, 860  
    needed when class changes, 270  
    templates, 656  
        error detection, 657  
        explicit instantiation, 675–676  
compiler  
    extension, 114, 131  
    GNU, 5  
    Microsoft, 5  
    options for separate compilation, 207  
composition vs. inheritance, 637  
compound assignment (e.g., +=)  
    arithmetic operators, 147  
    bitwise operators, 147  
compound expression, *see* expression  
compound statement, 173, 199  
compound type, 50, 50–58, 78  
    array, 113  
    declaration style, 57  
    understanding complicated declarations, 115  
concatenation  
    string, 89  
    string literal, 39  
condition, 12, 26  
    = (assignment) in, 146  
    conversion, 159  
    do while statement, 189  
    for statement, 13, 185  
    if statement, 18, 175  
    in IO expression, 156  
    logical operators, 141  
    smart pointer as, 451  
    stream type as, 15, 162, 312

while statement, 12, 183  
 condition state, IO classes, **312**, 324  
 conditional compilation, 240  
 conditional operator (? :), 151  
 connection, 468  
 console window, 6  
**const**, **59**, 78
 

- and `typedef`, 68
- conversion, 162
  - template argument deduction, 679
- dynamically allocated
  - destruction, 461
  - initialization, 460
- initialization, 59
  - class type object, 262
- low-level `const`, **64**
  - argument and parameter, 213
  - conversion from, 163
  - conversion to, 162
  - overloaded function, 232
  - template argument deduction, 693
- member function, **258**, 305
  - `()` (call operator), 573
  - not constructors, 262
  - overloaded function, 276
  - reference return, 276
- parameter, 212
  - function matching, 246
  - overloaded function, 232
- pointer, **63**, 78
  - conversion from `nonconst`, 162
  - initialization from `nonconst`, 62
  - overloaded parameter, 232
- reference, *see* reference to `const`
- top-level `const`, **64**
  - and `auto`, 69
  - argument and parameter, 212
  - `decltype`, 71
  - parameter, 232
  - template argument deduction, 679
- variable, 59
  - declared in header files, 76
  - `extern`, 60
  - local to file, 60

- `const_cast`, **163**, **163**
- `const_iterator`, container, **108**, 332
- `const_reference`, container, 333
- `const_reverse_iterator`, container, 332, 407
- constant expression, **65**, 78

array dimension, 113  
 bit-field, 854  
 case label, 179  
 enumerator, 833  
 integral, **65**  
 non-type template parameter, 655  
`sizeof`, 156  
 static data member, 303  
**constexpr**, **66**, 78
 

- constructor, **299**
- declared in header files, 76
- function, **239**, 251
  - non-constant return value, 239
- function template, 655
- pointer, 67
- variable, 66

- `construct`
- `allocator`, 482
- forwards to constructor, 527
- `constructor`, **262**, **264**, 262–266, 305
- see also* default constructor
- see also* copy constructor
- see also* move constructor
- calls to virtual function, 627
- `constexpr`, **299**
- converting, 294, 305
  - function matching, 585
  - `Sales_data`, 295
  - with standard conversion, 580
- default argument, 290
- delegating, **291**, 306
- derived class, 598
- initializes direct base class, 610
- initializes virtual base, 813
- `explicit`, **296**, 306
- function `try` block, 778, 817
- inherited, 628
- initializer list, **265**, 288–292, 305
- class member initialization, 274
- compared to assignment, 288
- derived class, 598
- function `try` block, 778, 817
- sometimes required, 288
- virtual base class, 814
- `initializer_list` parameter, 662
- `not const`, 262
- order of initialization, 289
- derived class object, 598, 623
- multiple inheritance, 804
- virtual base classes, 814
- overloaded, 262

StrBlob, 456  
StrBlobPtr, 474  
TextQuery, 488  
Blob, 662  
    initializer\_list, 662  
    iterator parameters, 673  
Bulk\_quote, 598, 610  
Disc\_quote, 609  
Sales\_data, 264–266  
container, 96, 131, 326, 372  
    *see also* sequential container  
    *see also* associative container  
adaptor, 368, 368–371  
    equality and relational operators, 370  
initialization, 369  
    requirements on container, 369  
and inheritance, 630  
as element type, 97, 329  
associative, 420, 447  
copy initialization, 334  
element type constraints, 329, 341  
elements and destructor, 502  
elements are copies, 342  
initialization from iterator range, 335  
list initialization, 336  
members  
    *see also* iterator  
    = (assignment), 337  
    == (equality), 341  
    != (inequality), 341  
begin, 106, 333, 372  
cbegin, 109, 333, 334, 372  
cend, 109, 333, 334, 372  
const\_iterator, 108, 332  
const\_reference, 333  
const\_reverse\_iterator, 332, 407  
crbegin, 333  
crend, 333  
difference\_type, 131, 332  
empty, 87, 102, 131, 340  
end, 106, 131, 333, 373  
equality and relational operators,  
    88, 102, 340  
iterator, 108, 332  
rbegin, 333, 407  
reference, 333  
relational operators, 341  
rend, 333, 407  
reverse\_iterator, 332, 407  
size, 88, 102, 132, 340  
size\_type, 88, 102, 132, 332  
swap, 339  
move operations, 529  
    moved-from object is valid but un-  
        specified, 537  
nonmember swap, 339  
of container, 97, 329  
overview, 328  
sequential, 326, 373  
type members, :: (scope operator),  
    333  
continue statement, 191, 199  
control, flow of, 11, 172, 200  
conversion, 78, 159, 168  
    = (assignment), 145, 159  
    ambiguous, 583–589  
    argument, 203  
    arithmetic, 35, 159, 168  
    array to pointer, 117  
        argument, 214  
        exception object, 774  
        multidimensional array, 128  
        template argument deduction, 679  
base-to-derived, not automatic, 602  
bool, 35  
class type, 162, 294, 305, 590  
    ambiguities, 587  
    conversion operator, 579  
    function matching, 584, 586  
    with standard conversion, 581  
condition, 159  
derived-to-base, 597, 649  
    accessibility, 613  
    key concepts, 604  
    shared\_ptr, 630  
floating-point, 35  
function to pointer, 248  
    exception object, 774  
    template argument deduction, 679  
integral promotion, 160, 169  
istream, 162  
multiple inheritance, 805  
    ambiguous, 806  
narrowing, 43  
operand, 159  
pointer to bool, 162  
rank, 245  
return value, 223  
Sales\_data, 295  
signed type, 160

- signed to unsigned, 34
- to const, 162
  - from pointer to nonconst, 62
  - from reference to nonconst, 61
  - template argument deduction, 679
- unscoped enumeration to integer, 834
- unsigned, 36
- virtual base class, 812
- conversion operator, 580, 580–587, 590
  - design, 581
  - explicit, 582, 590
    - bool, 583
  - function matching, 585, 586
  - SmallInt, 580
  - used implicitly, 580
  - with standard conversion, 580
- converting constructor, 294, 305
  - function matching, 585
  - with standard conversion, 580
- \_copy algorithms, 383, 414
- copy, 382, 874
  - copy and swap assignment, 518
    - move assignment, 540
    - self-assignment, 519
  - copy assignment, 500–501, 549
    - = default, 506
    - = delete, 507
  - base from derived, 603
  - copy and swap, 518, 549
  - derived class, 626
  - HasPtr
    - reference counted, 516
    - valuelike, 512
  - memberwise, 500
  - Message, 523
  - preventing copies, 507
  - private, 509
  - reference count, 514
  - rule of three/five, 505
    - virtual destructor exception, 622
  - self-assignment, 512
  - StrVec, 528
  - synthesized, 500, 550
    - deleted function, 508, 624
    - derived class, 623
    - multiple inheritance, 805
  - union with class type member, 852
  - valuelike class, 512
- copy constructor, 496, 496–499, 549
  - = default, 506
  - = delete, 507
- base from derived, 603
- derived class, 626
- HasPtr
  - reference counted, 515
  - valuelike, 512
- memberwise, 497
- Message, 522
- parameter, 496
- preventing copies, 507
- private, 509
- reference count, 514
- rule of three/five, 505
  - virtual destructor exception, 622
- StrVec, 528
- synthesized, 497, 550
  - deleted function, 508, 624
  - derived class, 623
  - multiple inheritance, 805
- union with class type member, 851
- used for copy-initialization, 498
- copy control, 267, 496, 549
  - = delete, 507–508
  - inheritance, 623–629
  - memberwise, 267, 550
    - copy assignment, 500
    - copy constructor, 497
    - move assignment, 538
    - move constructor, 538
  - multiple inheritance, 805
  - synthesized, 267
    - as deleted function, 508
    - as deleted in derived class, 624
    - move operations as deleted function, 538
- unions, 849
  - virtual base class, synthesized, 815
- copy initialization, 84, 131, 497, 497–499, 549
  - array, 337
  - container, 334
  - container elements, 342
  - explicit constructor, 498
  - invalid for arrays, 114
  - move vs. copy, 539
  - parameter and return value, 498
  - uses copy constructor, 497
  - uses move constructor, 541
- copy\_backward, 875
- copy\_if, 874
- copy\_n, 874
- copyUnion, Token, 851

count  
  algorithm, 378, 871  
  associative container, 437, 438  
  bitset, 727  
count\_calls, program, 206  
count\_if, 871  
cout, 6, 26  
  tied to cin, 315  
cplusplus\_primer, namespace, 787  
crbegin, container, 333  
cref, binds reference parameter, 400, 417  
cregex\_iterator, 733, 769  
crend, container, 333  
cstddef header, 116, 120  
cstdio header, 762  
cstdlib header, 54, 227, 778, 823  
cstring  
  functions, 122–123  
  header, 122  
csub\_match, 733, 769  
ctime header, 749  
curly brace, 2, 26

## D

dangling else, 177, 199  
dangling pointer, 225, 463, 491  
  undefined behavior, 463  
data abstraction, 254, 306  
data hiding, 270  
data member, *see* class data member  
data structure, 19, 26  
deallocate, allocator, 483, 528  
debug\_rep program  
  additional nontemplate versions, 698  
  general template version, 695  
  nontemplate version, 697  
  pointer template version, 696  
DebugDelete, member template, 673  
dec, manipulator, 754  
decimal, literal, 38  
declaration, 45, 78  
  class, 278, 305  
  class template, 669  
  class type, variable, 294  
  compound type, 57  
  dependencies  
    member function as friend, 281  
    overloaded templates, 698  
    template friends, 665  
    template instantiation, 657

  template specializations, 708  
  variadic templates, 702  
derived class, 600  
explicit instantiation, 675  
friend, 269  
function template, 669  
instantiation, 713  
member template, 673  
template, 669  
template specialization, 708  
type alias, 68  
using, 82, 132  
  access control, 615  
  overloaded inherited functions, 621  
variable, 45  
  const, 60  
declarator, 50, 79  
decltype, 70, 79  
  array return type, 230  
  cbegin, 109, 379  
  cend, 109, 379  
  depends on form, 71  
  for type abbreviation, 88, 106, 129  
  of array, 118  
  of function, 250  
  pointer to function, 249  
  top-level const, 71  
  yields lvalue, 71, 135  
decrement operators, 147–149  
default argument, 236, 251  
  adding default arguments, 237  
  and header file, 238  
  constructor, 290  
  default constructor, 291  
  function call, 236  
  function matching, 243  
  initializer, 238  
  static member, 304  
  virtual function, 607  
default case label, 181, 199  
default constructor, 263, 306  
  = default, 265  
  = delete, 507  
  default argument, 291  
  Sales\_data, 262  
  StrVec, 526  
  synthesized, 263, 306  
    deleted function, 508, 624  
    derived class, 623  
Token, 850  
used implicitly

default initialization, 293  
 value initialization, 293  
**default initialization**, 43  
 array, 336  
 built-in type, 43  
 class type, 44  
 string, 44, 84  
 uses default constructor, 293  
 vector, 97  
**default template argument**, 670  
 class template, 671  
 compare, 670  
 function template, 670  
 template<>, 671  
**default\_random\_engine**, 745, 769  
**defaultfloat manipulator**, 757  
**definition**, 79  
 array, 113  
 associative container, 423  
 base class, 594  
 class, 72, 256–267  
 class template, 659  
     member function, 661  
     static member, 667  
 class template partial specialization, 711  
 derived class, 596  
 dynamically allocated object, 459  
 explicit instantiation, 675  
 function, 577  
 in if condition, 175  
 in while condition, 183  
 instantiation, 713  
 member function, 256–260  
 multidimensional array, 126  
 namespace, 785  
     can be discontiguous, 786  
     member, 788  
 overloaded operator, 500, 552  
 pair, 426  
 pointer, 52  
 pointer to function, 247  
 pointer to member, 836  
 reference, 51  
 sequential container, 334  
 shared\_ptr, 450  
 static member, 302  
 string, 84  
 template specialization, 706–712  
 unique\_ptr, 470, 472  
 variable, 41, 45  
 const, 60  
 variable after case label, 182  
 vector, 97  
 weak\_ptr, 473  
**delegating constructor**, 291, 306  
**delete**, 460, 460–463, 491  
     const object, 461  
     execution flow, 820  
     memory leak, 462  
     null pointer, 461  
     pointer, 460  
     runs destructor, 502  
**delete []**, dynamically allocated array, 478  
**deleted function**, 507, 549  
**deleter**, 469, 491  
     shared\_ptr, 469, 480, 491  
     unique\_ptr, 472, 491  
**deprecated**, 401  
     auto\_ptr, 471  
     binary\_function, 579  
     bind1st, 401  
     bind2nd, 401  
     generalized exception specification, 780  
     ptr\_fun, 401  
     unary\_function, 579  
**deque**, 372  
     *see also* container, container member  
     *see also* sequential container  
     [ ] (subscript), 347  
     at, 348  
     header, 329  
     initialization, 334–337  
     list initialization, 336  
     overview, 327  
     push\_back, invalidates iterator, 354  
     push\_front, invalidates iterator, 354  
     random-access iterator, 412  
     value initialization, 336  
**deref**, StrBlobPtr, 475  
**derived class**, 592, 649  
     *see also* virtual function  
     :: (scope operator) to access base-class member, 607  
     = (assignment), 626  
     access control, 613  
     as base class, 600  
     assigned or copied to base object, 603  
     base-to-derived conversion, not automatic, 602

- constructor, 598
  - initializer list, 598
  - initializes direct base class, 610
  - initializes virtual base, 813
- copy assignment, 626
- copy constructor, 626
- declaration, 600
- default derivation specifier, 616
- definition, 596
- derivation list, **596**, 649
  - access control, 612
- derived object
  - contains base part, 597
  - multiple inheritance, 803
- derived-to-base conversion, 597
  - accessibility, 613
  - key concepts, 604
  - multiple inheritance, 805
- destructor, 627
- direct base class, **600**, 649
- final**, 600
- friendship not inherited, 615
- indirect base class, **600**, 650
- is user of base class, 614
- member new and delete, 822
- move assignment, 626
- move constructor, 626
- multiple inheritance, 803
- name lookup, 617
- order of destruction, 627
  - multiple inheritance, 805
- order of initialization, 598, 623
  - multiple inheritance, 804
  - virtual base classes, 814
- scope, 617
  - hidden base members, 619
  - inheritance, 617–621
  - multiple inheritance, 807
  - name lookup, 618
  - virtual function, 620
- static** members, 599
- synthesized
  - copy control members, 623
  - deleted copy control members, 624
- using declaration
  - access control, 615
  - overloaded inherited functions, 621
- virtual function, 596
- derived-to-base conversion, **597**, 649
  - accessible, 613
  - key concepts, 604
- multiple inheritance, 805
- not base-to-derived, 602
- shared\_ptr**, 630
- design
  - access control, 614
  - Bulk\_quote**, 592
  - conversion operator, 581
  - Disc\_quote**, 608
  - equality and relational operators, 562
  - generic programs, 655
  - inheritance, 637
  - Message class, 520
  - namespace, 786
  - overloaded operator, 554–556
  - Query classes, 636–639
  - Quote**, 592
  - reference count, 514
  - StrVec**, 525
- destination sequence, 381, 413
- destroy**, allocator, 482, 528
- destructor, **452**, 491, **501**, 501–503, 549
  - = default, 506
  - called during exception handling, 773
  - calls to virtual function, 627
  - container elements, 502
  - derived class, 627
  - doesn't delete pointer members, 503
  - explicit call to, 824
- HasPtr**
  - reference counted, 515
  - value-like, 512
- local variables, 502
- Message, 522
- not deleted function, 508
- not private**, 509
- order of destruction, 502
  - derived class, 627
  - multiple inheritance, 805
  - virtual base classes, 815
- reference count, 514
- rule of three/five, **505**
  - virtual destructor, exception, 622
- run by **delete**, 502
- shared\_ptr**, 453
- should not throw exception, 774
- StrVec**, 528
- synthesized, **503**, 550
  - deleted function, 508, 624
  - derived class, 623
  - multiple inheritance, 805
- Token**, 850

- valuelike class, 512
  - virtual function, 622
  - virtual in base class, 622
  - development environment, integrated, 3
  - difference\_type, 112**
    - vector, 112
    - container, 131, 332
    - string, 112
  - direct base class, 600
  - direct initialization, 84, 131
    - emplace members use, 345
  - Disc\_quote**
    - abstract base class, 610
    - class definition, 609
    - constructor, 609
    - design, 608
  - discriminant, 849, 862
    - Token, 850
  - distribution types
    - bernoulli\_distribution, 752
    - default template argument, 750
    - normal\_distribution, 751
    - random-number library, 745
    - uniform\_int\_distribution, 746
    - uniform\_real\_distribution, 750
  - divides<T>, 575
  - division rounding, 141
  - do while statement, 189, 200
  - domain\_error, 197
  - double, 33
    - literal (*numEnum or numenum*), 38
    - output format, 755
    - output notation, 757
  - dynamic binding, 593, 650
    - requirements for, 603
    - static vs. dynamic type, 605
  - dynamic type, 601, 650
  - dynamic\_cast, 163, 825, 825, 862
    - bad\_cast, 826
    - to pointer, 825
    - to reference, 826
  - dynamically allocated, 450, 491
    - array, 476, 476–484
    - allocator, 481
    - can't use begin and end, 477
    - can't use range for statement, 477
    - delete [], 478
    - empty array, 478
    - new [], 477
    - returns pointer to an element, 477
    - shared\_ptr, 480
  - unique\_ptr, 479
  - delete runs destructor, 502
  - lifetime, 450
  - new runs constructor, 458
  - object, 458–463
    - const object, 460
    - delete, 460
    - factory program, 461
    - initialization, 459
    - make\_shared, 451
    - new, 458
    - shared objects, 455, 486
    - shared\_ptr, 464
    - unique\_ptr, 470
- ## E
- echo command, 4
  - ECMAScript, 730, 739
    - regular expression library, 730
  - edit-compile-debug, 16, 26
    - errors at link time, 657
  - element type constraints, container, 329, 341
  - elimDups program, 383–391
  - ellipsis, parameter, 222
  - else, see if statement**
  - emplace**
    - associative container, 432
    - priority\_queue, 371
    - queue, 371
    - sequential container, 345
    - stack, 371
  - emplace\_back**
    - sequential container, 345
    - StrVec, 704
  - emplace\_front**, sequential container, 345
  - empty**
    - container, 87, 102, 131, 340
    - priority\_queue, 371
    - queue, 371
    - stack, 371
  - encapsulation, 254, 306
    - benefits of, 270
  - end**
    - associative container, 430
    - container, 106, 131, 333, 373
    - function, 118, 131
    - multidimensional array, 129
    - StrBlob, 475
    - StrVec, 526

end-of-file, 15, 26, 762  
    character, 15  
Endangered, 803  
endl, 7  
    manipulator, 314  
ends, manipulator, 315  
engine, random-number library, 745, 770  
    default\_random\_engine, 745  
    max, min, 747  
    retain state, 747  
    seed, 748, 770  
enum, unscoped enumeration, 832  
enum class, scoped enumeration, 832  
enumeration, 832, 863  
    as union discriminant, 850  
    function matching, 835  
    scoped, 832, 864  
    unscoped, 832, 864  
        conversion to integer, 834  
        unnamed, 832  
enumerator, 832, 863  
    constant expression, 833  
    conversion to integer, 834  
eof, 313  
eofbit, 312  
equal, 380, 872  
equal virtual function, 829  
equal\_range  
    algorithm, 722, 873  
    associative container, 439  
equal\_to<T>, 575  
equality operators, 141  
    arithmetic conversion, 144  
    container adaptor, 370  
    container member, 340  
    iterator, 106  
    overloaded operator, 561  
pointer, 120  
Sales\_data, 561  
string, 88  
vector, 102  
erase  
    associative container, 434  
    changes container size, 385  
    invalidates iterator, 349  
    sequential container, 349  
        string, 362  
error, standard, 6  
error\_type, 732  
error\_msg program, 221  
ERRORLEVEL, 4  
escape sequence, 39, 79  
    hexadecimal (`\xnnn`), 39  
    octal (`\nnn`), 39  
eval function  
    AndQuery, 646  
    NotQuery, 647  
    OrQuery, 645  
exception  
    class, 193, 200  
    class hierarchy, 783  
    deriving from, 782  
        Sales\_data, 783  
    header, 197  
    initialization, 197  
    what, 195, 782  
exception handling, 193–198, 772, 817  
    *see also* throw  
    *see also* catch  
exception declaration, 195, 775, 816  
    and inheritance, 775  
    must be complete type, 775  
exception in destructor, 773  
exception object, 774, 817  
finding a catch, 776  
function try block, 778, 817  
handler, *see* catch  
local variables destroyed, 773  
noexcept specification, 535, 779, 817  
nonthrowing function, 779, 818  
safe resource allocation, 467  
stack unwinding, 773, 818  
terminate function, 196, 200  
try block, 194, 773  
uncaught exception, 773  
unhandled exception, 196  
exception object, 774, 817  
    catch, 775  
    conversion to pointer, 774  
    initializes catch parameter, 775  
    pointer to local object, 774  
    rethrow, 777  
exception safety, 196, 200  
    smart pointers, 467  
exception specification  
    argument, 780  
    generalized, deprecated, 780  
    noexcept, 779  
    nonthrowing, 779  
    pointer to function, 779, 781  
    throw(), 780  
    violation, 779

- virtual function, 781  
 executable file, 5, 251  
 execution flow  
     () (call operator), 203  
     delete, 820  
     for statement, 186  
     new, 820  
     switch statement, 180  
     throw, 196, 773  
**EXIT\_FAILURE, 227**  
**EXIT\_SUCCESS, 227**  
 expansion  
     forward, 705  
     parameter pack, **702, 702–704, 714**  
         function parameter pack, 703  
         template parameter pack, 703  
     pattern, 702  
**explicit**  
     constructor, **296, 306**  
         copy initialization, 498  
     conversion operator, **582, 590**  
     conversion to `bool`, 583  
**explicit call to**  
     destructor, 824  
     overloaded operator, 553  
     postfix operators, 568  
**explicit instantiation, 675, 713**  
**explicit template argument, 660, 713**  
     class template, 660  
     forward, 694  
     function template, **682**  
         function pointer, 686  
     template argument deduction, 682  
**exporting C++ to C, 860**  
**expression, 7, 27, 134, 168**  
     callable, *see* callable object  
     constant, **65, 78**  
     lambda, *see* lambda expression  
     operand conversion, 159  
     order of evaluation, 137  
     parenthesized, 136  
     precedence and associativity, 136–137  
     regular, *see* regular expression  
**expression statement, 172, 200**  
**extension, compiler, 114, 131**  
**extern**  
     and `const` variables, 60  
     explicit instantiation, 675  
     variable declaration, 45  
**extern 'C', *see* linkage directive**
- F**
- fact program, 202**  
**factorial program, 227**  
**factory program**  
     new, 461  
     `shared_ptr`, 453  
**fail, 313**  
**failbit, 312**  
**failure, new, 460**  
**file, source, 4**  
**file extension, program, 730**  
     version 2, 738  
**file marker, stream, 765**  
**file mode, 319, 324**  
**file redirection, 22**  
**file static, 792, 817**  
**file stream, *see* `fstream`**  
**fill, 380, 874**  
**fill\_n, 381, 874**  
**final specifier, 600**  
     class, 600  
     virtual function, 607  
**find**  
     algorithm, 376, 871  
     associative container, 437, 438  
     string, 364  
**find last word program, 408**  
**find\_char program, 211**  
**find\_first\_of, 872**  
**find\_first\_not\_of, string, 365**  
**find\_first\_of, 872**  
     string, 365  
**find\_if, 388, 397, 414, 871**  
**find\_if\_not, 871**  
**find\_if\_not\_of, 871**  
**find\_last\_not\_of, string, 366**  
**find\_last\_of, string, 366**  
**findBook, program, 721**  
**fixed manipulator, 757**  
**flip**  
     `bitset`, 727  
     program, 694  
**flip1, program, 692**  
**flip2, program, 693**  
**float, 33**  
     literal (`numF` or `numf`), 41  
**floating-point, 32**  
     conversion, 35  
     literal, 38  
     output format, 755

output notation, 757  
flow of control, 11, **172**, 200  
`flush`, manipulator, 315  
`Folder`, *see Message*  
`for` statement, **13**, **27**, **185**, 185–187, 200  
    condition, 13  
    execution flow, 186  
    `for` header, 185  
    range, **91**, **187**, 187–189, 200  
        can't add elements, 101, 188  
        multidimensional array, 128  
`for_each`, 391, 872  
`format` state, stream, 753  
formatted IO, **761**, 769  
`forward`, **694**  
    argument-dependent lookup, 798  
    explicit template argument, 694  
    pack expansion, 705  
    passes argument type unchanged, 694, 705  
    usage pattern, 706  
`forward declaration`, class, **279**, 306  
`forward iterator`, **411**, 417  
`forward_list`  
    *see also* container  
    *see also* sequential container  
    `before_begin`, 351  
    forward iterator, 411  
    header, 329  
    initialization, 334–337  
    list initialization, 336  
    `merge`, 415  
    overview, 327  
    `remove`, 415  
    `remove_if`, 415  
    `reverse`, 415  
    `splice_after`, 416  
    `unique`, 415  
    value initialization, 336  
`forwarding`, 692–694  
    passes argument type unchanged, 694  
    preserving type information, 692  
    rvalue reference parameters, 693, 705  
    typical implementation, 706  
    variadic template, 704  
`free`, `StrVec`, 528  
`free` library function, **823**, 863  
`free store`, **450**, 491  
`friend`, **269**, 306  
    class, 280  
    class template type parameter, 666  
declaration, **269**  
declaration dependencies  
    member function as friend, 281  
    template friends, 665  
function, 269  
inheritance, 614  
member function, 280, 281  
overloaded function, 281  
scope, 270, 281  
    namespace, 799  
template as, 664  
`front`  
    queue, 371  
sequential container, 346  
`StrBlob`, 457  
`front_inserter`, **402**, 417  
    compared to `inserter`, 402  
    requires `push_front`, 402  
`fstream`, 316–320  
    close, 318  
    file marker, 765  
    file mode, **319**  
    header, 310, 316  
    initialization, 317  
    `off_type`, 766  
    open, 318  
    `pos_type`, 766  
    random access, 765  
    random IO program, 766  
    `seek` and `tell`, 763–768  
function, **2**, **27**, **202**, 251  
    *see also* return type  
    *see also* return value  
block, 204  
body, **2**, **27**, **202**, 251  
callable object, 388  
candidate, 251  
candidate function, **243**  
`constexpr`, **239**, 251  
    nonconstant return value, 239  
declaration, 206  
declaration and header file, 207  
`decltype` returns function type, 250  
default argument, **236**, 251  
    adding default arguments, 237  
    and header file, 238  
    initializer, 238  
deleted, **507**, 549  
    function matching, 508  
exception specification  
    `noexcept`, 779

- `throw()`, 780
- `friend`, 269
- `function` to pointer conversion, 248
- `inline`, **238**, 252
  - and header, 240
  - linkage directive, 859
  - member, *see* member function
  - name, **2**, 27
  - nonthrowing, **779**, 818
  - overloaded
    - compared to redeclaration, 231
    - friend declaration, 281
    - scope, 234
  - parameter, *see* parameter
  - parameter list, **2**, 27, **202**, 204
  - prototype, **207**, 251
  - recursive, **227**
    - variadic template, 701
  - scope, 204
  - viable, 252
  - viable function, **243**
  - virtual, *see* virtual function
- `function`, **577**, 576–579, 590
  - and pointer to member, 842
  - definition, 577
  - desk calculator, 577
- `function call`
  - ambiguous, **234**, 245, 251
  - default argument, 236
  - execution flow, 203
  - overhead, 238
  - through pointer to function, 248
  - through pointer to member, 839
  - to overloaded operator, 553
  - to overloaded postfix operator, 568
- `function matching`, **233**, 251
  - `= delete`, 508
  - argument, conversion, 234
  - candidate function, **243**
    - overloaded operator, 587
  - `const` arguments, 246
  - conversion, class type, 583–587
  - conversion operator, 585, 586
  - conversion rank, 245
    - class type conversions, 586
  - default argument, 243
  - enumeration, 835
  - `function template`, 694–699
    - specialization, 708
  - integral promotions, 246
  - member function, 273
- multiple parameters, 244
- `namespace`, 800
- `overloaded operator`, 587–589
- `prefers more specialized function`, 695
- `rvalue reference`, 539
- `variadic template`, 702
- `viable function`, **243**
- `function object`, **571**, 590
  - argument to algorithms, 572
  - arithmetic operators, 574
  - is callable object, 571
- `function parameter`, *see* parameter
- `function parameter pack`, 700
  - expansion, 703
  - pattern, 704
- `function pointer`, 247–250
  - callable object, 388
  - definition, 247
  - exception specification, 779, 781
  - function template instantiation, 686
  - overloaded function, 248
  - parameter, 249
  - return type, 204, 249
    - using `decltype`, 250
  - template argument deduction, 686
  - type alias declaration, 249
  - `typedef`, 249
- `function table`, **577**, **577**, 590, 840
- `function template`, **652**, 713
  - see also* template parameter
  - see also* template argument deduction
  - see also* instantiation
  - argument conversion, 680
  - array function parameters, 654
  - candidate function, 695
  - `compare`, 652
    - string literal version, 654
  - `constexpr`, 655
  - declaration, 669
  - default template argument, 670
  - error detection, 657
  - explicit instantiation, **675**, 675–676
  - explicit template argument, **682**
    - `compare`, 683
  - function matching, 694–699
  - `inline` function, 655
  - nontype parameter, 654
  - overloaded function, 694–699
  - parameter pack, 713
  - specialization, 707, 714
    - `compare`, 706

function matching, 708  
is an instantiation, 708  
namespace, 788  
scope, 708  
trailing return type, 684  
type-dependent code, 658  
function try block, 778, 817  
functional header, 397, 399, 400, 575,  
    577, 843

## G

g++, 5  
gcount, istream, 763  
generate, 874  
generate\_n, 874  
generic algorithms, *see* algorithms  
generic programming, 108  
    type-independent code, 655  
get  
    istream, 761  
    multi-byte version, istream, 762  
    returns int, istream, 762, 764  
get<n>, 719, 770  
getline, 87, 131  
    istream, 762  
    istringstream, 321  
    TextQuery constructor, 488  
global function  
    operator delete, 863  
    operator new, 863  
global namespace, 789, 817  
    :: (scope operator), 789, 818  
global scope, 48, 80  
global variable, lifetime, 204  
GNU compiler, 5  
good, 313  
goto statement, 192, 200  
grade clusters program, 103  
greater<T>, 575  
greater\_equal<T>, 575

## H

.h file header, 19  
handler, *see* catch  
has-a relationship, 637  
hash<key\_type>, 445, 447  
    override, 446  
    specialization, 709, 788  
        compatible with == (equality), 710

hash function, 443, 447  
HasPtr  
    reference counted, 514–516  
    copy assignment, 516  
    destructor, 515  
    valuelike, 512  
        copy assignment, 512  
        move assignment, 540  
        move constructor, 540  
        swap, 516  
header, 6, 27  
    iostream, 27  
C library, 91  
const and constexpr, 76  
default argument, 238  
function declaration, 207  
.h file, 19  
#include, 6, 21  
inline function, 240  
inline member function definition,  
    273  
namespace members, 786  
standard, 6  
table of library names, 866  
template definition, 656  
template specialization, 708  
user-defined, 21, 76–77, 207, 240  
using declaration, 83  
Sales\_data.h, 76  
Sales\_item.h, 19  
algorithm, 376  
array, 329  
bitset, 723  
cassert, 241  
cctype, 91  
cmath, 751, 757  
cstddef, 116, 120  
cstdio, 762  
cstdlib, 54, 227, 778, 823  
cstring, 122  
ctime, 749  
deque, 329  
exception, 197  
forward\_list, 329  
fstream, 310, 316  
functional, 397, 399, 400, 575, 577,  
    843  
initializer\_list, 220  
iomanip, 756  
iostream, 6, 310  
iterator, 119, 382, 401

- list, 329
  - map, 420
  - memory, 450, 451, 481, 483
  - new, 197, 460, 478, 821
  - numeric, 376, 881
  - queue, 371
  - random, 745
  - regex, 728
  - set, 420
  - sstream, 310, 321
  - stack, 370
  - stdexcept, 194, 197
  - string, 74, 76, 84
  - tuple, 718
  - type\_info, 197
  - type\_traits, 684
  - typeinfo, 826, 827, 831
  - unordered\_map, 420
  - unordered\_set, 420
  - utility, 426, 530, 533, 694
  - vector, 96, 329
  - header guard, 77, 79
    - preprocessor, 77
  - heap, 450, 491
  - hex, manipulator, 754
  - hexadecimal
    - escape sequence (\Xnm), 39
    - literal (0Xnum or 0xnum), 38
  - hexfloat manipulator, 757
  - high-order bits, 723, 770
- 
- ## I
- i before e, program, 729
    - version 2, 734
  - IDE, 3
  - identifier, 46, 79
    - reserved, 46
  - \_if algorithms, 414
  - if statement, 17, 27, 175, 175–178, 200
    - compared to switch, 178
    - condition, 18, 175
    - dangling else, 177
    - else branch, 18, 175, 200
  - ifstream, 311, 316–320, 324
    - see also* istream
    - close, 318
    - file marker, 765
    - file mode, 319
    - initialization, 317
    - off\_type, 766
  - open, 318
  - pos\_type, 766
  - random access, 765
  - random IO program, 766
  - seek and tell, 763–768
  - ignore, istream, 763
  - implementation, 254, 254, 306
    - in (file mode), 319
    - in scope, 49, 79
  - in-class initializer, 73, 73, 79, 263, 265, 274
  - #include
    - standard header, 6, 21
    - user-defined header, 21
  - includes, 880
  - incomplete type, 279, 306
    - can't be base class, 600
    - not in exception declaration, 775
    - restrictions on use, 279
  - incr, StrBlobPtr, 475
  - increment operators, 147–149
  - indentation, 19, 177
  - index, 94, 131
    - see also* [ ] (subscript)
  - indirect base class, 600, 650
  - inferred return type, lambda expression, 396
  - inheritance, 650
    - and container, 630
    - conversions, 604
    - copy control, 623–629
    - friend, 614
    - hierarchy, 592, 600
    - interface class, 637
    - IO classes, 311, 324
    - name collisions, 618
    - private, 612, 650
    - protected, 612, 650
    - public, 612, 650
    - vs. composition, 637
  - inherited, constructor, 628
  - initialization
    - aggregate class, 298
    - array, 114
    - associative container, 423, 424
    - bitset, 723–725
    - C-style string, 122
    - class type objects, 73, 262
    - const
      - static data member, 302
      - class type object, 262
      - data member, 289

- object, 59
- copy, 84, 131, 497, 497–499, 549
- default, 43, 293
- direct, 84, 131
- dynamically allocated object, 459
- exception, 197
- istream\_iterator, 405
- list, *see* list initialization
- lvalue reference, 532
- multidimensional array, 126
- new [], 477
- ostream\_iterator, 405
- pair, 426
- parameter, 203, 208
- pointer, 52–54
  - to const, 62
- queue, 369
- reference, 51
  - data member, 289
  - to const, 61
- return value, 224
- rvalue reference, 532
- sequential container, 334–337
- shared\_ptr, 464
- stack, 369
- string, 84–85, 360–361
- string streams, 321
- tuple, 718
- unique\_ptr, 470
- value, 98, 132, 293
- variable, 42, 43, 79
- vector, 97–101
  - vs. assignment, 42, 288
- weak\_ptr, 473
- initializer\_list, 220, 220–222, 252
  - = (assignment), 563
  - constructor, 662
  - header, 220
- inline function, 238, 252
  - and header, 240
  - function template, 655
  - member function, 257, 273
    - and header, 273
- inline namespace, 790, 817
- inner scope, 48, 79
- inner\_product, 882
- inplace\_merge, 875
- input, standard, 6
- input iterator, 411, 418
- insert
  - associative container, 432
  - multiple key container, 433
  - sequential container, 343
    - string, 362
  - insert iterator, 382, 401, 402, 418
    - back\_inserter, 402
    - front\_inserter, 402
    - inserter, 402
    - inserter, 402, 418
      - compared to front\_inserter, 402
  - instantiation, 96, 131, 653, 656, 713
    - Blob, 660
    - class template, 660
      - member function, 663
    - declaration, 713
    - definition, 713
    - error detection, 657
    - explicit, 675–676
    - function template from function pointer, 686
    - member template, 674
    - static member, 667
  - int, 33
    - literal, 38
  - integral
    - constant expression, 65
    - promotion, 134, 160, 169
      - function matching, 246
    - type, 32, 79
  - integrated development environment, 3
  - interface, 254, 306
  - internal, manipulator, 759
  - interval, left-inclusive, 373
  - invalid pointer, 52
  - invalid\_argument, 197
  - invalidated iterator
    - and container operations, 354
    - undefined behavior, 353
  - invalidates iterator
    - assign, 338
    - erase, 349
    - resize, 352
  - IO
    - formatted, 761, 769
    - unformatted, 761, 770
  - IO classes
    - condition state, 312, 324
    - inheritance, 324
  - IO stream, *see* stream
  - iomanip header, 756
  - iostate, 312
    - machine-dependent, 313

`iostream`, 5  
  file marker, 765  
  header, 6, 27, 310  
  `off_type`, 766  
  `pos_type`, 766  
  random access, 765  
  random IO program, 766  
  seek and `tell`, 763–768  
  virtual base class, 810  
`iota`, 882  
`is-a` relationship, 637  
`is_partitioned`, 876  
`is_permutation`, 879  
`is_sorted`, 877  
`is_sorted_until`, 877  
`isalnum`, 92  
`isalpha`, 92  
`isbn`  
  `Sales_data`, 257  
  `Sales_item`, 23  
`ISBN`, 2  
`isbn_mismatch`, 783  
`iscntrl`, 92  
`isdigit`, 92  
`isgraph`, 92  
`islower`, 92  
`isprint`, 92  
`ispunct`, 92  
`isShorter` program, 211  
`isspace`, 92  
`istream`, 5, 27, 311  
  *see also* manipulator  
  `>>` (input operator), 8  
    precedence and associativity, 155  
  as condition, 15  
  chained input, 8  
  condition state, 312  
  conversion, 162  
  explicit conversion to `bool`, 583  
  file marker, 765  
  flushing input buffer, 314  
  format state, 753  
  `gcount`, 763  
  `get`, 761  
    multi-byte version, 762  
    returns `int`, 762, 764  
  `getline`, 87, 321, 762  
  `ignore`, 763  
  no copy or assign, 311  
  `off_type`, 766  
  `peek`, 761  
  `pos_type`, 766  
  `put`, 761  
  `putback`, 761  
  random access, 765  
  random IO program, 766  
  `read`, 763  
  `seek` and `tell`, 763–768  
  unformatted IO, 761  
    multi-byte, 763  
    single-byte, 761  
  `unget`, 761  
`istream_iterator`, 403, 418  
  `>>` (input operator), 403  
  algorithms, 404  
  initialization, 405  
  off-the-end iterator, 403  
  operations, 404  
  type requirements, 406  
`istringstream`, 311, 321, 321–323  
  *see also* `istream`  
  word per line processing, 442  
  file marker, 765  
  `getline`, 321  
  initialization, 321  
  `off_type`, 766  
  phone number program, 321  
  `pos_type`, 766  
  random access, 765  
  random IO program, 766  
  seek and `tell`, 763–768  
  `TextQuery` constructor, 488  
`isupper`, 92  
`isxdigit`, 92  
`iter_swap`, 875  
`iterator`, 106, 106–112, 131  
  `++` (increment), 107, 132  
  `--` (decrement), 107  
  `*` (dereference), 107  
  `+=` (compound assignment), 111  
  `+` (addition), 111  
  `-` (subtraction), 111  
  `==` (equality), 106, 107  
  `!=` (inequality), 106, 107  
  algorithm type independence, 377  
  arithmetic, 111, 131  
  compared to reverse iterator, 409  
  destination, 413  
  `insert`, 401, 418  
  `move`, 401, 418, 543  
    `uninitialized_copy`, 543  
  off-the-beginning

- before\_begin, 351
  - forward\_list, 351
  - off-the-end, **106**, 132, 373
    - istream\_iterator, 403
  - parameter, 216
  - regex, 734
  - relational operators, 111
  - reverse, **401**, 407–409, 418
  - stream, **401**, 403–406, 418
    - used as destination, 382
  - iterator
    - compared to reverse\_iterator, 408
    - container, **108**, 332
    - header, 119, 382, 401
    - set iterators are const, 429
  - iterator category, **410**, 410–412, 418
    - bidirectional iterator, **412**, 417
    - forward iterator, **411**, 417
    - input iterator, **411**, 418
    - output iterator, **411**, 418
    - random-access iterator, **412**, 418
  - iterator range, **331**, 331–332, 373
    - algorithms, 376
    - as initializer of container, 335
    - container erase member, 349
    - container insert member, 344
    - left-inclusive, **331**
    - off-the-end, 331
  - ## K
  - key concept
    - algorithms
      - and containers, 378
      - iterator arguments, 381
    - class user, 255
    - classes define behavior, 20
    - defining an assignment operator, 512
    - dynamic binding in C++, 605
    - elements are copies, 342
    - encapsulation, 270
    - headers for template code, 657
    - indentation, 19
    - inheritance and conversions, 604
    - isA and hasA relationships, 637
    - name lookup and inheritance, 619
    - protected members, 614
    - refactoring, 611
    - respecting base class interface, 599
    - specialization declarations, 708
  - type checking, 46
  - types define behavior, 3
  - use concise expressions, 149
- key\_type
  - associative container, 428, 447
  - requirements
    - ordered container, 425
    - unordered container, 445
- keyword table, 47
- Koenig lookup, 797
- ## L
- L'c' (wchar\_t literal), 38
- label
  - case, **179**, 199
  - statement, 192
- labeled statement, **192**, 200
- lambda expression, **388**, 418
  - arguments, 389
  - biggies program, 391
    - reference capture, 393
  - capture list, **388**, 417
    - capture by reference, 393
    - capture by value, 390, 392
    - implicit capture, 394
  - inferred return type, 389, 396
  - mutable, 395
  - parameters, 389
  - passed to find\_if, 390
  - passed to stable\_sort, 389
  - synthesized class type, 572–574
  - trailing return type, 396
- left, manipulator, 758
- left-inclusive interval, **331**, 373
- length\_error, 197
- less<T>, 575
- less\_equal<T>, 575
- letter grade, program, 175
- lexicographical\_compare, 881
- library function objects, 574
  - as arguments to algorithms, 575
- library names to header table, 866
- library type, **5**, 27, 82
- lifetime, **204**, 252
  - compared to scope, 204
  - dynamically allocated objects, 450, 461
  - global variable, 204
  - local variable, 204
  - parameter, 205
- linkage directive, **858**, 863

- C++ to C, 860
- compound, 858
- overloaded function, 860
- parameter or return type, 859
- pointer to function, 859
- return type, 859
- single, 858
- linker, **208**, 252
  - template errors at link time, 657
- list**, 373
  - see also* container
  - see also* sequential container
  - bidirectional iterator, 412
  - header, 329
  - initialization, 334–337
  - list initialization, 336
  - merge, 415
  - overview, 327
  - remove, 415
  - remove\_if, 415
  - reverse, 415
  - splice, 416
  - unique, 415
  - value initialization, 336
- list initialization, **43**, 79
  - = (assignment), 145
  - array, 337
  - associative container, 423
  - container, 336
  - dynamically allocated, object, 459
  - pair, 427, 431, 527
  - preferred, 99
  - prevents narrowing, 43
  - return value, **226**, 427, 527
  - sequential container, 336
  - vector, 98
- literal, **38**, 38–41, 79
  - bool, 41
    - in condition, 143
  - char, 39
  - decimal, 38
  - double (*numEnum* or *numenum*), 38
  - float (*numF* or *numf*), 41
  - floating-point, 38
  - hexadecimal (0X*num* or 0x*num*), 38
  - int, 38
  - long (*numL* or *numl*), 38
  - long double (*ddd.dddL* or *ddd.ddd1*), 41
  - long long (*numLL* or *numl1*), 38
  - octal (0*num*), 38
- string, **7**, 28, 39
- unsigned (*numU* or *numu*), 41
- wchar\_t, 40
- literal type, 66
  - class type, 299
- local class, **852**, 863
  - access control, 853
  - name lookup, 853
  - nested class in, 854
  - restrictions, 852
- local scope, *see* block scope
- local static object, **205**, 252
- local variable, **204**, 252
  - destroyed during exception handling, 467, 773
  - destructor, 502
  - lifetime, 204
  - pointer, return value, 225
  - reference, return value, 225
  - return statement, 224
- lock, weak\_ptr, 473
- logic\_error, 197
- logical operators, 141, 142
  - condition, 141
  - function object, 574
- logical\_and<T>, 575
- logical\_not<T>, 575
- logical\_or<T>, 575
- long, 33
  - literal (*numL* or *numl*), 38
- long double, 33
  - literal (*ddd.dddL* or *ddd.ddd1*), 41
- long long, 33
  - literal (*numLL* or *numl1*), 38
- lookup, name, *see* name lookup
- low-level const, **64**, 79
  - argument and parameter, 213
  - conversion from, 163
  - conversion to, 162
  - overloaded function, 232
  - template argument deduction, 693
- low-order bits, **723**, 770
- lower\_bound
  - algorithm, 873
  - ordered container, 438
- lround, 751
- lvalue, **135**, 169
  - cast to rvalue reference, 691
  - copy initialization, uses copy constructor, 539
- decltype, 135

reference collapsing rule, 688  
result  
  - > (arrow operator), 150  
  ++ (increment) prefix, 148  
  -- (decrement) prefix, 148  
  \* (dereference), 135  
  [ ] (subscript), 135  
  = (assignment), 145  
  , (comma operator), 158  
  ?: (conditional operator), 151  
  cast, 163  
  decltype, 71  
  function reference return type, 226  
  variable, 533  
lvalue reference, *see also* reference, 532, 549  
  collapsing rule, 688  
  compared to rvalue reference, 533  
  function matching, 539  
  initialization, 532  
  member function, 546  
    overloaded, 547  
  move, 533  
  template argument deduction, 687

## M

machine-dependent  
  bit-field layout, 854  
  char representation, 34  
  end-of-file character, 15  
  enum representation, 835  
  iostate, 313  
  linkage directive language, 861  
  nonzero return from main, 227  
  random IO, 763  
  reinterpret\_cast, 164  
  return from exception what, 198  
  signed out-of-range value, 35  
  signed types and bitwise operators,  
    153  
  size of arithmetic types, 32  
  terminate function, 196  
  type\_info members, 831  
  vector, memory management, 355  
  volatile implementation, 856  
main, 2, 27  
  not recursive, 228  
  parameters, 218  
  return type, 2  
  return value, 2–4, 227  
make\_move\_iterator, 543

make\_pair, 428  
make\_plural program, 224  
make\_shared, 451  
make\_tuple, 718  
malloc library function, 823, 863  
manipulator, 7, 27, 753, 770  
  boolalpha, 754  
  change format state, 753  
  dec, 754  
  defaultfloat, 757  
  endl, 314  
  ends, 315  
  fixed, 757  
  flush, 315  
  hex, 754  
  hexfloat, 757  
  internal, 759  
  left, 758  
  noboolalpha, 754  
  noshowbase, 755  
  noshowpoint, 758  
  noskipws, 760  
  nouppercase, 755  
  oct, 754  
  right, 758  
  scientific, 757  
  setfill, 759  
  setprecision, 756  
  setw, 758  
  showbase, 755  
  showpoint, 758  
  skipws, 760  
  unitbuf, 315  
  uppercase, 755  
map, 420, 447  
  *see also* ordered container  
  \* (dereference), 429  
  [ ] (subscript), 435, 448  
    adds element, 435  
  at, 435  
  definition, 423  
  header, 420  
  insert, 431  
  key\_type requirements, 425  
  list initialization, 423  
  lower\_bound, 438  
  map, initialization, 424  
  TextQuery class, 485  
  upper\_bound, 438  
word\_count program, 421

**mapped\_type**, associative container, 428, 448  
**match**  
  best, 251  
  no, 252  
**match\_flag\_type**, *regex\_constants*, 743  
**max**, 881  
**max\_element**, 881  
**mem\_fn**, 843, 863  
  generates callable, 843  
**member**, *see* class data member  
**member access operators**, 150  
**member function**, 23, 27, 306  
  as friend, 281  
  base member hidden by derived, 619  
  class template  
    defined outside class body, 661  
    instantiation, 663  
  **const**, 258, 305  
  () (call operator), 573  
  reference return, 276  
  declared but not defined, 509  
  defined outside class, 259  
  definition, 256–260  
    :: (scope operator), 259  
    name lookup, 285  
    parameter list, 282  
    return type, 283  
  explicitly *inline*, 273  
  function matching, 273  
  implicit *this* parameter, 257  
  implicitly *inline*, 257  
  *inline* and header, 273  
  move-enabled, 545  
  name lookup, 287  
  overloaded, 273  
    on **const**, 276  
    on lvalue or rvalue reference, 547  
  overloaded operator, 500, 552  
  reference qualified, 546, 550  
  returning *\*this*, 260, 275  
  rvalue reference parameters, 544  
  scope, 282  
  template, *see* member template  
**member template**, 672, 714  
  **Blob**, iterator constructor, 673  
  **DebugDelete**, 673  
  declaration, 673  
  defined outside class body, 674  
  instantiation, 674  
**template parameters**, 673, 674  
**memberwise**  
  copy assignment, 500  
  copy constructor, 497  
  copy control, 267, 550  
  destruction is implicit, 503  
  move assignment, 538  
  move constructor, 538  
**memory**  
  *see also* dynamically allocated  
  exhaustion, 460  
  leak, 462  
**memory header**, 450, 451, 481, 483  
**merge**, 874  
  list and *forward\_list*, 415  
**Message**, 519–524  
  **add\_to\_Folder**, 522  
  class definition, 521  
  copy assignment, 523  
  copy constructor, 522  
  design, 520  
  destructor, 522  
  move assignment, 542  
  move constructor, 542  
  **move\_Folders**, 542  
  **remove\_from\_Folders**, 523  
**method**, *see* member function  
**Microsoft compiler**, 5  
**min**, 881  
**min\_element**, 881  
**minmax**, 881  
**minus**<T>, 575  
**mismatch**, 872  
**mode**, *file*, 324  
**modulus**<T>, 575  
**move**, 530, 533, 874  
  argument-dependent lookup, 798  
  binds rvalue reference to lvalue, 533  
  explained, 690–692  
  inherently dangerous, 544  
  **Message**, move operations, 541  
  moved from object has unspecified  
    value, 533  
  reference collapsing rule, 691  
  **StrVec** *reallocate*, 530  
  **remove\_reference**, 691  
**move assignment**, 536, 550  
  copy and swap, 540  
  derived class, 626  
  **HasPtr**, valuelike, 540  
  memberwise, 538

- Message, 542  
moved-from object destructible, 537  
noexcept, 535  
rule of three/five, virtual destructor  
    exception, 622  
self-assignment, 537  
StrVec, 536  
synthesized  
    deleted function, 538, 624  
    derived class, 623  
    multiple inheritance, 805  
    sometimes omitted, 538  
move constructor, 529, 534, 534–536, 550  
    and copy initialization, 541  
    derived class, 626  
    HasPtr, valuelike, 540  
    memberwise, 538  
    Message, 542  
    moved-from object destructible, 534,  
        537  
    noexcept, 535  
    rule of three/five, virtual destructor  
        exception, 622  
    string, 529  
    StrVec, 535  
    synthesized  
        deleted function, 624  
        derived class, 623  
        multiple inheritance, 805  
        sometimes omitted, 538  
move iterator, 401, 418, 543, 550  
    make\_move\_iterator, 543  
    StrVec, reallocate, 543  
    uninitialized\_copy, 543  
move operations, 531–548  
    function matching, 539  
    move, 533  
    noexcept, 535  
    rvalue references, 532  
    valid but unspecified, 537  
move\_backward, 875  
move\_Folders, Message, 542  
multidimensional array, 125–130  
    [ ] (subscript), 127  
argument and parameter, 218  
begin, 129  
conversion to pointer, 128  
definition, 126  
end, 129  
initialization, 126  
pointer, 128  
range for statement and, 128  
multimap, 448  
    *see also* ordered container  
    \* (dereference), 429  
    definition, 423  
    has no subscript operator, 435  
    insert, 431, 433  
    key\_type requirements, 425  
    list initialization, 423  
    lower\_bound, 438  
    map, initialization, 424  
    upper\_bound, 438  
multiple inheritance, 802, 817  
    *see also* virtual base class  
    = (assignment), 805  
ambiguous conversion, 806  
ambiguous names, 808  
avoiding ambiguities, 809  
class derivation list, 803  
conversion, 805  
copy control, 805  
name lookup, 807  
object composition, 803  
order of initialization, 804  
scope, 807  
virtual function, 807  
multiplies<T>, 575  
multiset, 448  
    *see also* ordered container  
    insert, 433  
    iterator, 429  
    key\_type requirements, 425  
    list initialization, 423  
    lower\_bound, 438  
override comparison  
    Basket class, 631  
    using compareISBN, 426  
    upper\_bound, 438  
        used in Basket, 632  
mutable  
    data member, 274  
    lambda expression, 395

## N

- \n (newline character), 39  
name lookup, 283, 306  
    :: (scope operator), overrides, 286  
argument-dependent lookup, 797  
before type checking, 619  
multiple inheritance, 809

- block scope, 48
- class, 284
- class member
  - declaration, 284
  - definition, 285, 287
  - function, 284
- depends on static type, 617, 619
  - multiple inheritance, 806
- derived class, 617
  - name collisions, 618
- local class, 853
- multiple inheritance, 807
  - ambiguous names, 808
- namespace, 796
- nested class, 846
- overloaded virtual functions, 621
- templates, 657
- type checking, 235
- virtual base class, 812
- named cast, **162**
  - `const_cast`, **163**, **163**
  - `dynamic_cast`, **163**, **825**
  - `reinterpret_cast`, **163**, **164**
  - `static_cast`, **163**, **163**
- namespace, **7**, **27**, **785**, **817**
  - alias, **792**, **817**
  - argument-dependent lookup, 797
  - candidate function, 800
  - `cplusplus_primer`, 787
  - definition, 785
  - design, 786
  - discontiguous definition, 786
  - friend declaration scope, 799
  - function matching, 800
  - global, **789**, **817**
  - inline, **790**, **817**
  - member, 786
  - member definition, 788
    - outside namespace, 788
  - name lookup, 796
  - nested, 789
  - overloaded function, 800
  - placeholders, 399
  - scope, 785–790
  - `std`, **7**
  - template specialization, 709, 788
  - unnamed, **791**, **818**
    - local to file, 791
    - replace file `static`, 792
- namespace pollution, **785**, **817**
- narrowing conversion, 43
- NDEBUG, 241
- `negate<T>`, 575
- nested class, **843**, **863**
  - access control, 844
  - class defined outside enclosing class, 845
  - constructor, `QueryResult`, 845
  - in local class, 854
  - member defined outside class body, 845
  - name lookup, 846
  - `QueryResult`, 844
  - relationship to enclosing class, 844, 846
  - scope, 844
  - static member, 845
- nested namespace, 789
- nested type, *see* nested class
- `new`, **458**, **458**–**460**, **491**
  - execution flow, 820
  - failure, 460
  - header, 197, 460, 478, 821
  - initialization, 458
  - placement, 460, 491, **824**, **863**
    - union with class type member, 851
  - `shared_ptr`, 464
  - `unique_ptr`, 470
  - with `auto`, 459
- `new []`, **477**, **477**–**478**
  - initialization, 477
  - returns pointer to an element, 477
  - value initialization, 478
- `newline (\n)`, character, 39
- `next_permutation`, 879
- no match, **234**, **252**
  - see also* function matching
- `noboolalpha`, manipulator, 754
- `NoDefault`, 293
- `noexcept`
  - exception specification, **779**, **817**
    - argument, 779–781
    - violation, 779
  - move operations, 535
  - operator, **780**, **817**
- `nonconst` reference, *see* reference
- `none`, `bitset`, 726
- `none_of`, 871
- nonportable, 36, 863
- nonprintable character, **39**, **79**
- nonthrowing function, **779**, **818**
- `nontype` parameter, **654**, **714**

compare, 654  
must be constant expression, 655  
type requirements, 655  
`normal_distribution`, 751  
`noshowbase`, manipulator, 755  
`noshowpoint`, manipulator, 758  
`noskipws`, manipulator, 760  
`not_equal_to<T>`, 575  
`NotQuery`, 637  
    class definition, 642  
    eval function, 647  
`nouppercase`, manipulator, 755  
`nth_element`, 877  
`NULL`, 54  
`null (\0)`, character, 39  
null pointer, 53, 79  
    delete of, 461  
null statement, 172, 200  
null-terminated character string, *see C-style string*  
`nullptr`, 54, 79  
numeric header, 376, 881  
numeric conversion, to and from `string`, 367  
numeric literal  
    `float (numF or numf)`, 41  
    `long (numL or numl)`, 41  
    `long double (ddd.dddL or ddd.ddd1)`, 41  
    `long long (numLL or numl1)`, 41  
    `unsigned (numU or numu)`, 41

## O

object, 42, 79  
    automatic, 205, 251  
    dynamically allocated, 458–463  
        `const object`, 460  
        `delete`, 460  
        factory program, 461  
        initialization, 459  
        lifetime, 450  
        new, 458  
    lifetime, 204, 252  
    local static, 205, 252  
    order of destruction  
        class type object, 502  
        derived class object, 627  
        multiple inheritance, 805  
        virtual base classes, 815  
    order of initialization

class type object, 289  
derived class object, 598, 623  
multiple inheritance, 804  
virtual base classes, 814  
object code, 252  
object file, 208, 252  
object-oriented programming, 650  
`oct`, manipulator, 754  
octal, literal (`0num`), 38  
octal escape sequence (`\nnn`), 39  
off-the-beginning iterator, 351, 373  
    `before_begin`, 351  
    `forward_list`, 351  
off-the-end  
    iterator, 106, 132, 373  
    iterator range, 331  
    pointer, 118  
`ofstream`, 311, 316–320, 324  
    *see also ostream*  
    `close`, 318  
    file marker, 765  
    file mode, 319  
    initialization, 317  
    `off_type`, 766  
    `open`, 318  
    `pos_type`, 766  
    random access, 765  
    random IO program, 766  
    `seek and tell`, 763–768  
old-style, cast, 164  
open, file stream, 318  
operand, 134, 169  
    conversion, 159  
operator, 134, 169  
operator alternative name, 46  
operator `delete`  
    class member, 822  
    global function, 820, 863  
operator `delete []`  
    class member, 822  
    compared to `deallocate`, 823  
    global function, 820  
operator `new`  
    class member, 822  
    global function, 820, 863  
operator `new []`  
    class member, 822  
    compared to `allocate`, 823  
    global function, 820  
operator overloading, *see overloaded operator*

- operators
- arithmetic, 139
  - assignment, 12, 144–147
  - binary, 134, 168
  - bitwise, 152–156
    - `bitset`, 725
  - comma (,), 157
  - compound assignment, 12
  - conditional (? :), 151
  - decrement, 147–149
  - equality, 18, 141
  - increment, 12, 147–149
  - input, 8
  - iterator
    - addition and subtraction, 111
    - arrow, 110
    - dereference, 107
    - equality, 106, 108
    - increment and decrement, 107
    - relational, 111
  - logical, 141
  - member access, 150
  - `noexcept`, 780
  - output, 7
  - overloaded, arithmetic, 560
  - pointer
    - addition and subtraction, 119
    - equality, 120
    - increment and decrement, 118
    - relational, 120, 123
    - subscript, 121
  - relational, 12, 141, 143
  - `Sales_data`
    - `+=` (compound assignment), 564
    - `+` (addition), 560
    - `==` (equality), 561
    - `!=` (inequality), 561
    - `>>` (input operator), 558
    - `<<` (output operator), 557
  - `Sales_item`, 20
  - scope, 82
  - `sizeof`, 156
  - `sizeof...`, 700
  - `string`
    - addition, 89
    - equality and relational, 88
    - IO, 85
    - subscript, 93–95
  - subscript, 116
  - `typeid`, 826, 864
  - unary, 134, 169
- `vector`
  - equality and relational, 102
  - subscript, 103–105
- options to `main`, 218
- order of destruction
  - class type object, 502
  - derived class object, 627
  - multiple inheritance, 805
  - virtual base classes, 815
- order of evaluation, 134, 169
  - `&&` (logical AND), 138
  - `||` (logical OR), 138
  - `,` (comma operator), 138
  - `? :` (conditional operator), 138
  - expression, 137
  - pitfalls, 149
- order of initialization
  - class type object, 289
  - derived class object, 598
  - multiple base classes, 816
  - multiple inheritance, 804
  - virtual base classes, 814
- ordered container
  - see also* container
  - see also* associative container
  - `key_type` requirements, 425
  - `lower_bound`, 438
  - override default comparison, 425
  - `upper_bound`, 438
- ordering, strict weak, 425, 448
- `OrQuery`, 637
  - class definition, 644
  - `eval` function, 645
- `ostream`, 5, 27, 311
  - see also* manipulator
  - `<<` (output operator), 7
    - precedence and associativity, 155
  - chained output, 7
  - condition state, 312
  - explicit conversion to `bool`, 583
  - file marker, 765
  - floating-point notation, 757
  - flushing output buffer, 314
  - format state, 753
  - no copy or assign, 311
  - not flushed if program crashes, 315
  - `off_type`, 766
  - output format, floating-point, 755
  - `pos_type`, 766
  - precision member, 756
  - random access, 765

random IO program, 766  
seek and tell, 763–768  
tie member, 315  
virtual base class, 810  
write, 763  
`ostream_iterator`, 403, 418  
  << (output operator), 405  
algorithms, 404  
initialization, 405  
operations, 405  
type requirements, 406  
`ostringstream`, 311, 321, 321–323  
  *see also* `ostream`  
file marker, 765  
initialization, 321  
`off_type`, 766  
phone number program, 323  
`pos_type`, 766  
random access, 765  
random IO program, 766  
seek and tell, 763–768  
str, 323  
`out` (file mode), 319  
out-of-range value, signed, 35  
`out_of_range`, 197  
  at function, 348  
`out_of_stock`, 783  
outer scope, 48, 79  
output, standard, 6  
output iterator, 411, 418  
overflow, 140  
`overflow_error`, 197  
overhead, function call, 238  
overload resolution, *see* function matching  
  ing  
overloaded function, 230, 230–235, 252  
  *see also* function matching  
  as friend, 281  
  compared to redeclaration, 231  
  compared to template specialization,  
    708  
const parameters, 232  
constructor, 262  
function template, 694–699  
linkage directive, 860  
member function, 273  
  const, 276  
  move-enabled, 545  
  reference qualified, 547  
  virtual, 621  
move-enabled, 545  
namespace, 800  
pointer to, 248  
scope, 234  
  derived hides base, 619  
using declaration, 800  
  in derived class, 621  
using directive, 801  
overloaded operator, 135, 169, 500, 550,  
  552, 590  
  ++ (increment), 566–568  
  -- (decrement), 566–568  
  \* (dereference), 569  
    `StrBlobPtr`, 569  
  & (address-of), 554  
  -> (arrow operator), 569  
    `StrBlobPtr`, 569  
  [ ] (subscript), 564  
    `StrVec`, 565  
  () (call operator), 571  
    `absInt`, 571  
    `PrintString`, 571  
  = (assignment), 500, 563  
    `StrVec initializer_list`, 563  
  += (compound assignment), 555, 560  
    `Sales_data`, 564  
  + (addition), `Sales_data`, 560  
  == (equality), 561  
    `Sales_data`, 561  
  != (inequality), 562  
    `Sales_data`, 561  
  < (less-than), strict weak ordering,  
    562  
  >> (input operator), 558–559  
    `Sales_data`, 558  
  << (output operator), 557–558  
    `Sales_data`, 557  
  && (logical AND), 554  
  || (logical OR), 554  
  & (bitwise AND), `Query`, 644  
  | (bitwise OR), `Query`, 644  
  ~ (bitwise NOT), `Query`, 643  
  , (comma operator), 554  
ambiguous, 588  
arithmetic operators, 560  
associativity, 553  
binary operators, 552  
candidate function, 587  
consistency between relational and  
  equality operators, 562  
definition, 500, 552  
design, 554–556

- equality operators, 561  
 explicit call to, 553  
     postfix operators, 568  
 function matching, 587–589  
 member function, 500, 552  
 member vs. nonmember function,  
     552, 555  
 precedence, 553  
 relational operators, 562  
 requires class-type parameter, 552  
 short-circuit evaluation lost, 553  
 unary operators, 552  
 override, virtual function, 595, 650  
 override specifier, 593, 596, 606
- P**
- pair, **426**, 448  
 default initialization, 427  
 definition, 426  
 initialization, 426  
 list initialization, 427, 431, 527  
 make\_pair, 428  
 map, \* (dereference), 429  
 operations, 427  
 public data members, 427  
 return value, 527
- Panda, 803
- parameter, **202**, 208, 252  
 array, 214–219  
     buffer overflow, 215  
     to pointer conversion, 214  
 C-style string, 216  
 const, 212  
 copy constructor, 496  
 ellipsis, 222  
 forwarding, 693  
 function pointer, linkage directive, 859  
 implicit this, **257**  
 initialization, 203, 208  
 iterator, 216  
 lifetime, 205  
 low-level const, 213  
 main function, 218  
 multidimensional array, 218  
 nonreference, 209  
     uses copy constructor, 498  
     uses move constructor, 539  
 pass by reference, **210**, 252  
 pass by value, **209**, 252  
 passing, 208–212
- pointer, 209, 214  
 pointer to const, 246  
 pointer to array, 218  
 pointer to function, 249  
     linkage directive, 859  
 reference, 210–214  
     to const, 213, 246  
     to array, 217  
 reference to const, 211  
 template, *see* template parameter  
 top-level const, 212
- parameter list  
 function, 2, 27, **202**  
 template, 653, 714
- parameter pack, 714  
 expansion, **702**, 702–704, 714  
 function template, 713  
 sizeof..., 700  
 variadic template, 699
- parentheses, override precedence, 136
- partial\_sort, 877  
 partial\_sort\_copy, 877  
 partial\_sum, 882  
 partition, 876  
 partition\_copy, 876  
 partition\_point, 876  
 pass by reference, **208**, 210, 252  
 pass by value, **209**, 252  
     uses copy constructor, 498  
     uses move constructor, 539
- pattern, **702**, 714  
 function parameter pack, 704  
 regular expression, phone number, 739  
 template parameter pack, 703
- peek, istream, 761
- PersonInfo, 321
- phone number, regular expression  
 program, 738  
 reformat program, 742  
 valid, 740
- pitfalls  
 dynamic memory, 462  
 order of evaluation, 149  
 self-assignment, 512  
 smart pointer, 469  
     using directive, 795
- placeholders, 399
- placement new, 460, 491, **824**, 863  
     union, class type member, 851
- plus<T>, 575

- pointer, 52, 52–58, 79  
  `++` (increment), 118  
  `--` (decrement), 118  
  `*` (dereference), 53  
  `[ ]` (subscript), 121  
  `=` (assignment), 55  
  `+` (addition), 119  
  `-` (subtraction), 119  
  `==` (equality), 55, 120  
  `!=` (inequality), 55, 120  
  and array, 117  
  arithmetic, 119, 132  
  `const`, 63, 78  
  const pointer to const, 63  
  `constexpr`, 67  
  conversion  
    from array, 161  
    to `bool`, 162  
    to `const`, 62, 162  
    to `void*`, 161  
  dangling, 463, 491  
  declaration style, 57  
  definition, 52  
  `delete`, 460  
  derived-to-base conversion, 597  
    under multiple inheritance, 805  
  `dynamic_cast`, 825  
  `implicit this`, 257, 306  
  initialization, 52–54  
  invalid, 52  
  multidimensional array, 128  
  null, 53, 79  
  off-the-end, 118  
  parameter, 209, 214  
  relational operators, 123  
  return type, 204  
  return value, local variable, 225  
  smart, 450, 491  
  to `const`, 62  
    and `typedef`, 68  
  to array  
    parameter, 218  
    return type, 204  
    return type declaration, 229  
  to `const`, 79  
    overloaded parameter, 232, 246  
  to pointer, 58  
  `typeid` operator, 828  
  valid, 52  
    `volatile`, 856  
pointer to function, 247–250  
auto, 249  
callable object, 388  
`decltype`, 249  
exception specification, 779, 781  
explicit template argument, 686  
function template instantiation, 686  
linkage directive, 859  
overloaded function, 248  
parameter, 249  
return type, 204, 249  
  using `decltype`, 250  
template argument deduction, 686  
trailing return type, 250  
type alias, 249  
`typedef`, 249  
pointer to member, 835, 863  
arrow (`->*`), 837  
definition, 836  
dot (`.*`), 837  
function, 838  
  and `bind`, 843  
  and function, 842  
  and `mem_fn`, 843  
  not callable object, 842  
function call, 839  
function table, 840  
precedence, 838  
polymorphism, 605, 650  
pop  
  `priority_queue`, 371  
  queue, 371  
  stack, 371  
pop\_back  
  sequential container, 348  
  `StrBlob`, 457  
pop\_front, sequential container, 348  
portable, 854  
precedence, 134, 136–137, 169  
  `=` (assignment), 146  
  `? :` (conditional operator), 151  
  assignment and relational operators,  
    146  
  dot and dereference, 150  
  increment and dereference, 148  
  of IO operator, 156  
  overloaded operator, 553  
  parentheses overrides, 136  
  pointer to member and call operator,  
    838  
precedence table, 166  
precision member, `ostream`, 756

predicate, 386, 418  
     binary, 386, 417  
     unary, 386, 418  
 prefix, smatch, 736  
 preprocessor, 76, 79  
     #include, 7  
     assert macro, 241, 251  
     header guard, 77  
     variable, 54, 79  
 prev\_permutation, 879  
 print, Sales\_data, 261  
 print program  
     array parameter, 215  
     array reference parameter, 217  
     pointer and size parameters, 217  
     pointer parameter, 216  
     two pointer parameters, 216  
     variadic template, 701  
 print\_total  
     explained, 604  
     program, 593  
 PrintString, 571  
     () (call operator), 571  
 priority\_queue, 371, 373  
     emplace, 371  
     empty, 371  
     equality and relational operators, 370  
     initialization, 369  
     pop, 371  
     push, 371  
     sequential container, 371  
     size, 371  
     swap, 371  
     top, 371  
 private  
     access specifier, 268, 306  
     copy constructor and assignment, 509  
     inheritance, 612, 650  
 program  
     addition  
         Sales\_data, 74  
         Sales\_item, 21, 23  
     alternative\_sum, 682  
     biggies, 391  
     binops desk calculator, 577  
     book from author version 1, 438  
     book from author version 2, 439  
     book from author version 3, 440  
     bookstore  
         Sales\_data, 255  
         Sales\_data using algorithms, 406  
     Sales\_item, 24  
     buildMap, 442  
     children's story, 383–391  
     compare, 652  
     count\_calls, 206  
     debug\_rep  
         additional nontemplate versions, 698  
         general template version, 695  
         nontemplate version, 697  
         pointer template version, 696  
     elimDups, 383–391  
     error\_msg, 221  
     fact, 202  
     factorial, 227  
     factory  
         new, 461  
         shared\_ptr, 453  
     file extension, 730  
         version 2, 738  
     find last word, 408  
     find\_char, 211  
     findBook, 721  
     flip, 694  
     flip1, 692  
     flip2, 693  
     grade clusters, 103  
     grading  
         bitset, 728  
         bitwise operators, 154  
     i before e, 729  
         version 2, 734  
     isShorter, 211  
     letter grade, 175  
     make\_plural, 224  
     message handling, 519  
     phone number  
         istringstream, 321  
         ostringstream, 323  
         reformat, 742  
         regular expression version, 738  
         valid, 740  
     print  
         array parameter, 215  
         array reference parameter, 217  
         pointer and size parameters, 217  
         pointer parameter, 216  
         two pointer parameters, 216  
         variadic template, 701  
     print\_total, 593  
     Query, 635

class design, 636–639  
    random IO, 766  
    reset  
        pointer parameters, 209  
        reference parameters, 210  
    restricted word\_count, 422  
    sum, 682  
    swap, 223  
    TextQuery, 486  
        design, 485  
    transform, 442  
    valid, 740  
    vector capacity, 357  
    vowel counting, 179  
    word\_count  
        map, 421  
        unordered\_map, 444  
    word\_transform, 441  
    ZooAnimal, 802  
promotion, *see* integral promotion  
protected  
    access specifier, 595, 611, 650  
    inheritance, 612, 650  
    member, 611  
ptr\_fun deprecated, 401  
ptrdiff\_t, 120, 132  
public  
    access specifier, 268, 306  
    inheritance, 612, 650  
pure virtual function, 609, 650  
    Disc\_quote, 609  
    Query\_base, 636  
push  
    priority\_queue, 371  
    queue, 371  
    stack, 371  
push\_back  
    back\_inserter, 382, 402  
    sequential container, 100, 132, 342  
        move-enabled, 545  
    StrVec, 527  
        move-enabled, 545  
push\_front  
    front\_inserter, 402  
    sequential container, 342  
put, istream, 761  
putback, istream, 761

**Q**

Query, 638

<< (output operator), 641  
& (bitwise AND), 638  
    definition, 644  
| (bitwise OR), 638  
    definition, 644  
~ (bitwise NOT), 638  
    definition, 643  
classes, 636–639  
definition, 640  
interface class, 637  
operations, 635  
program, 635  
recap, 640  
Query\_base, 636  
    abstract base class, 636  
    definition, 639  
    member function, 636  
QueryResult, 485  
    class definition, 489  
    nested class, 844  
        constructor, 845  
    print, 490  
queue, 371, 373  
    back, 371  
    emplace, 371  
    empty, 371  
    equality and relational operators, 370  
    front, 371  
    header, 371  
    initialization, 369  
    pop, 371  
    push, 371  
    sequential container, 371  
    size, 371  
    swap, 371  
Quote  
    class definition, 594  
    design, 592

**R**

Raccoon, virtual base class, 812  
raise exception, *see* throw  
rand function, drawbacks, 745  
random header, 745  
random IO, 765  
    machine-dependent, 763  
    program, 766  
random-access iterator, 412, 418  
random-number library, 745  
    compared to rand function, 745



smatch, provides context for a match, 735  
subexpression, 738  
  file extension program version 2, 738  
  types, 733  
  valid, program, 740  
reinterpret\_cast, 163, 164  
  machine-dependent, 164  
relational operators, 141, 143  
  arithmetic conversion, 144  
  container adaptor, 370  
  container member, 340  
  function object, 574  
  iterator, 111  
  overloaded operator, 562  
  pointer, 120, 123  
  Sales\_data, 563  
  string, 88  
  tuple, 720  
  vector, 102  
release, unique\_ptr, 470  
remove, 878  
  list and forward\_list, 415  
remove\_copy, 878  
remove\_copy\_if, 878  
remove\_from\_Folders, Message, 523  
remove\_if, 878  
  list and forward\_list, 415  
remove\_pointer, 685  
remove\_reference, 684  
  move, 691  
rend, container, 333, 407  
replace, 383, 875  
  string, 362  
replace\_copy, 383, 874  
replace\_copy\_if, 874  
replace\_if, 875  
reserve  
  string, 356  
  vector, 356  
reserved identifiers, 46  
reset  
  bitset, 727  
  shared\_ptr, 466  
  unique\_ptr, 470  
reset program  
  pointer parameters, 209  
  reference parameters, 210  
resize  
  invalidates iterator, 352  
sequential container, 352  
value initialization, 352  
restricted word\_count program, 422  
result, 134, 169  
  \* (dereference), lvalue, 135  
  [ ] (subscript), lvalue, 135  
  , (comma operator), lvalue, 158  
  ?: (conditional operator), lvalue, 151  
  cast, lvalue, 163  
rethrow, 776  
  exception object, 777  
  throw, 776, 818  
return statement, 222, 222–228  
  from main, 227  
  implicit return from main, 223  
  local variable, 224, 225  
return type, 2, 27, 202, 204, 252  
  array, 204  
  array using decltype, 230  
  function, 204  
  function pointer, 249  
    using decltype, 250  
linkage directive, 859  
main, 2  
member function, 283  
nonreference, 224  
  copy initialized, 498  
pointer, 204  
  pointer to function, 204  
reference, 224  
  reference to const, 226  
  reference yields lvalue, 226  
trailing, 229, 252, 396, 684  
virtual function, 606  
void, 223  
return value  
  conversion, 223  
  copy initialized, 498  
  initialization, 224  
  list initialization, 226, 427, 527  
  local variable, pointer, 225  
  main, 2–4, 227  
  pair, 427, 527  
  reference, local variable, 225  
  \*this, 260, 275  
tuple, 721  
type checking, 223  
unique\_ptr, 471  
reverse, 878  
  list and forward\_list, 415  
reverse iterator, 401, 407–409, 418

++ (increment), 407  
 -- (decrement), 407, 408  
 base, 409  
     compared to iterator, 409  
 reverse\_copy, 414, 878  
 reverse\_copy\_if, 414  
 reverse\_iterator  
     compared to iterator, 408  
     container, 332, 407  
 rfind, string, 366  
 right, manipulator, 758  
 rotate, 878  
 rotate\_copy, 878  
 rule of three/five, 505, 541  
     virtual destructor exception, 622  
 run-time type identification, 825–831, 864  
     compared to virtual functions, 829  
 dynamic\_cast, 825, 825  
     bad\_cast, 826  
     to pointer, 825  
     to reference, 826  
 type-sensitive equality, 829  
 typeid, 826, 827  
     returns type\_info, 827  
 runtime binding, 594, 650  
 runtime\_error, 194, 197  
     initialization from string, 196  
 rvalue, 135, 169  
     copy initialization, uses move constructor, 539  
 result  
     ++ (increment) postfix, 148  
     -- (decrement) postfix, 148  
     function nonreference return type, 224  
 rvalue reference, 532, 550  
     cast from lvalue, 691  
     collapsing rule, 688  
     compared to lvalue reference, 533  
     function matching, 539  
     initialization, 532  
     member function, 546  
         overloaded, 547  
     move, 533  
     parameter  
         forwarding, 693, 705  
         member function, 544  
     preserves argument type information, 693  
     template argument deduction, 687  
     variable, 533

## S

Sales\_data  
     compareIsbn, 387  
     += (compound assignment), 564  
     + (addition), 560  
     == (equality), 561  
     != (inequality), 561  
     >> (input operator), 558  
     << (output operator), 557  
     add, 261  
     addition program, 74  
     avg\_price, 259  
     bookstore program, 255  
         using algorithms, 406  
     class definition, 72, 268  
     combine, 259  
     compareIsbn, 425  
         with associative container, 426  
     constructors, 264–266  
     converting constructor, 295  
     default constructor, 262  
     exception classes, 783  
     exception version  
         += (compound assignment), 784  
         + (addition), 784  
     explicit constructor, 296  
     isbn, 257  
     operations, 254  
     print, 261  
     read, 261  
     relational operators, 563  
 Sales\_data.h header, 76  
 Sales\_item, 20  
     + (addition), 22  
     >> (input operator), 21  
     << (output operator), 21  
     addition program, 21, 23  
     bookstore program, 24  
     isbn, 23  
     operations, 20  
 Sales\_item.h header, 19  
 scientific manipulator, 757  
 scope, 48, 80  
     base class, 617  
     block, 48, 80, 173  
     class, 73, 282, 282–287, 305  
     static member, 302  
     compared to object lifetime, 204  
     derived class, 617  
     friend, 270, 281

function, 204  
global, **48**, 80  
inheritance, 617–621  
member function, 282  
    parameters and return type, 283  
multiple inheritance, 807  
name collisions, using directive, 795  
namespace, 785–790  
nested class, 844  
overloaded function, 234  
statement, 174  
template parameter, 668  
template specialization, 708  
using directive, 794  
virtual function, 620  
scoped enumeration, **832**, 864  
    enum class, 832  
Screen, 271  
    pos member, 272  
    concatenating operations, 275  
    do\_display, 276  
    friends, 279  
    get, 273, 282  
    get\_cursor, 283  
    Menu function table, 840  
    move, 841  
    move members, 275  
    set, 275  
search, 872  
search\_n, 871  
seed, random-number engine, 748  
seekp, seekg, 763–768  
self-assignment  
    copy and swap assignment, 519  
    copy assignment, 512  
    explicit check, 542  
    HasPtr  
        reference counted, 515  
        valuelike, 512  
Message, 523  
move assignment, 537  
pitfalls, 512  
StrVec, 528  
semicolon (;), 3  
    class definition, 73  
    null statement, 172  
separate compilation, **44**, 80, 252  
    compiler options, 207  
    declaration vs. definition, 44  
    templates, 656  
sequential container, **326**, 373  
array, 326  
deque, 326  
forward\_list, 326  
initialization, 334–337  
list, 326  
list initialization, 336  
members  
    assign, 338  
    back, 346  
    clear, 350  
    emplace, **345**  
    emplace\_back, **345**  
    emplace\_front, **345**  
    erase, 349  
    front, 346  
    insert, 343  
    pop\_back, 348  
    pop\_front, 348  
    push\_back, 132  
    push\_back, **100**, 342, 545  
    push\_front, 342  
    resize, 352  
    value\_type, 333  
performance characteristics, 327  
priority\_queue, 371  
queue, 371  
stack, 370  
value initialization, 336  
vector, 326  
set, **420**, 448  
    *see also* ordered container  
    bitset, 727  
    header, 420  
    insert, 431  
    iterator, 429  
    key\_type requirements, 425  
    list initialization, 423  
    lower\_bound, 438  
    TextQuery class, 485  
    upper\_bound, 438  
        word\_count program, 422  
set\_difference, 880  
set\_intersection, 647, 880  
set\_symmetric\_difference, 880  
set\_union, 880  
setfill, manipulator, 759  
setprecision, manipulator, 756  
setstate, stream, 313  
setw, manipulator, 758  
shared\_ptr, **450**, 450–457, 464–469, 491  
    \* (dereference), 451

copy and assignment, 451  
 definition, 450  
 deleter, 469, 491  
     bound at run time, 677  
 derived-to-base conversion, 630  
 destructor, 453  
 dynamically allocated array, 480  
 exception safety, 467  
 factory program, 453  
 initialization, 464  
 make\_shared, 451  
 pitfalls, 469  
 reset, 466  
 StrBlob, 455  
 TextQuery class, 485  
 with new, 464  
**short**, 33  
 short-circuit evaluation, 142, 169  
     && (logical AND), 142  
     || (logical OR), 142  
     not in overloaded operator, 553  
**ShorterString**, 573  
     () (call operator), 573  
**shorterString**, 224  
**showbase**, manipulator, 755  
**showpoint**, manipulator, 758  
**shrink\_to\_fit**  
     deque, 357  
     string, 357  
     vector, 357  
**shuffle**, 878  
**signed**, 34, 80  
     char, 34  
     conversion to unsigned, 34, 160  
     out-of-range value, 35  
**signed type**, 34  
**single-line (//), comment**, 9, 26  
**size**  
     container, 88, 102, 132, 340  
     priority\_queue, 371  
     queue, 371  
     returns unsigned, 88  
     stack, 371  
     StrVec, 526  
**size\_t**, 116, 132, 727  
     array subscript, 116  
**size\_type**, container, 88, 102, 132, 332  
**SizeComp**, 573  
     () (call operator), 573  
**sizeof**, 156, 169  
     array, 157  
     data member, 157  
**sizeof...**, parameter pack, 700  
**skipws**, manipulator, 760  
**sliced**, 603, 650  
**SmallInt**  
     + (addition), 588  
     conversion operator, 580  
**smart pointer**, 450, 491  
     exception safety, 467  
     pitfalls, 469  
**smatch**, 729, 733, 769, 770  
     prefix, 736  
     provide context for a match, 735  
     suffix, 736  
**sort**, 384, 876  
**source file**, 4, 27  
**specialization**, *see* template specialization  
**splice**, list, 416  
**splice\_after**, forward\_list, 416  
**sregex\_iterator**, 733, 770  
     i before e program, 734  
**sstream**  
     file marker, 765  
     header, 310, 321  
     off\_type, 766  
     pos\_type, 766  
     random access, 765  
     random IO program, 766  
     seek and tell, 763–768  
**ssub\_match**, 733, 736, 770  
     example, 740  
**stable\_partition**, 876  
**stable\_sort**, 387, 876  
**stack**, 370, 373  
     emplace, 371  
     empty, 371  
     equality and relational operators, 370  
     header, 370  
     initialization, 369  
     pop, 371  
     push, 371  
     sequential container, 370  
     size, 371  
     swap, 371  
     top, 371  
**stack unwinding**, exception handling, 773, 818  
**standard error**, 6, 27  
**standard header**, #include, 6, 21  
**standard input**, 6, 27  
**standard library**, 5, 27

- standard output, 6, 27  
statement, 2, 27  
  block, *see* block  
  break, 190, 199  
  compound, 173, 199  
  continue, 191, 199  
  do while, 189, 200  
  expression, 172, 200  
  for, 13, 27, 185, 185–187, 200  
  goto, 192, 200  
  if, 17, 27, 175, 175–178, 200  
  labeled, 192, 200  
  null, 172, 200  
  range for, 91, 187, 187–189, 200  
  return, 222, 222–228  
  scope, 174  
  switch, 178, 178–182, 200  
  while, 11, 28, 183, 183–185, 200  
statement label, 192  
static (file static), 792, 817  
static member  
  Account, 301  
  class template, 667  
  accessed through an instantiation, 667  
  definition, 667  
  const data member, initialization, 302  
  data member, 300  
  definition, 302  
  default argument, 304  
  definition, 302  
  inheritance, 599  
  instantiation, 667  
  member function, 301  
  nested class, 845  
  scope, 302  
static object, local, 205, 252  
static type, 601, 650  
  determines name lookup, 617, 619  
  multiple inheritance, 806  
static type checking, 46  
static\_cast, 163, 163  
  lvalue to rvalue, 691  
std, 7, 28  
std::forward, *see* forward  
std::move, *see* move  
stdexcept header, 194, 197  
stod, 368  
stof, 368  
stoi, 368  
  stol, 368  
  stold, 368  
  stoll, 368  
  store, free, 450, 491  
  stoul, 368  
  stoull, 368  
  str, string streams, 323  
StrBlob, 456  
  back, 457  
  begin, 475  
  check, 457  
  constructor, 456  
  end, 475  
  front, 457  
  pop\_back, 457  
  shared\_ptr, 455  
StrBlobPtr, 474  
  ++ (increment), 566  
  -- (decrement), 566  
  \* (dereference), 569  
  -> (arrow operator), 569  
  check, 474  
  constructor, 474  
  deref, 475  
  incr, 475  
  weak\_ptr, 474  
strcat, 123  
strcmp, 123  
strcpy, 123  
stream  
  as condition, 15, 162, 312  
  clear, 313  
  explicit conversion to bool, 583  
  file marker, 765  
  flushing buffer, 314  
  format state, 753  
  istream\_iterator, 403  
  iterator, 401, 403–406, 418  
    type requirements, 406  
  not flushed if program crashes, 315  
  ostream\_iterator, 403  
  random IO, 765  
  rdstate, 313  
  setstate, 313  
  strict weak ordering, 425, 448  
  string, 80, 84–93, 132  
    *see also* container  
    *see also* sequential container  
    *see also* iterator  
  [ ] (subscript), 93, 132, 347  
  += (compound assignment), 89

+ (addition), 89  
 >> (input operator), 85, 132  
 >> (input operator) as condition, 86  
 << (output operator), 85, 132  
 and string literal, 89–90  
 append, 362  
 assign, 362  
 at, 348  
 C-style string, 124  
 c\_str, 124  
 capacity, 356  
 case sensitive, 365  
 compare, 366  
 concatenation, 89  
 default initialization, 44  
**difference\_type, 112**  
 equality and relational operators, 88  
 erase, 362  
 find, 364  
 find\_first\_not\_of, 365  
 find\_last\_not\_of, 366  
 find\_last\_of, 366  
 getline, 87, 321  
 header, 74, 76, 84  
 initialization, 84–85, 360–361  
 initialization from string literal, 84  
 insert, 362  
 move constructor, 529  
 numeric conversions, 367  
 random-access iterator, 412  
 replace, 362  
 reserve, 356  
 rfind, 366  
 subscript range, 95  
 substr, 361  
 TextQuery class, 485  
 string literal, 7, 28, 39  
*see also* C-style string  
 and string, 89–90  
 concatenation, 39  
**stringstream, 321**, 321–323, 324  
 initialization, 321  
**strlen, 122**  
**struct**  
*see also* class  
 default access specifier, 268  
 default inheritance specifier, 616  
**StrVec, 525**  
 [ ] (subscript), 565  
 = (assignment), **initializer\_list**, 563  
**alloc\_n\_copy, 527**  
**begin, 526**  
**capacity, 526**  
**chk\_n\_alloc, 526**  
**copy assignment, 528**  
**copy constructor, 528**  
**default constructor, 526**  
**design, 525**  
**destructor, 528**  
**emplace\_back, 704**  
**end, 526**  
**free, 528**  
**memory allocation strategy, 525**  
**move assignment, 536**  
**move constructor, 535**  
**push\_back, 527**  
 move-enabled, 545  
**reallocate, 530**  
 move iterator version, 543  
**size, 526**  
**subexpression, 770**  
**subscript range, 93**  
 array, 116  
 string, 95  
 validating, 104  
 vector, 105  
**substr, string, 361**  
**suffix, smatch, 736**  
**sum, program, 682**  
**swap, 516**  
 array, 339  
 container, 339  
 container nonmember version, 339  
 copy and swap assignment operator, 518  
**priority\_queue, 371**  
**queue, 371**  
**stack, 371**  
 typical implementation, 517–518  
**swap program, 223**  
**swap\_ranges, 875**  
**switch statement, 178**, 178–182, 200  
 default case label, 181  
 break, 179–181, 190  
 compared to if, 178  
 execution flow, 180  
 variable definition, 182  
**syntax\_option\_type, regex, 730**  
**synthesized**  
 copy assignment, 500, 550  
 copy constructor, 497, 550

- copy control, 267  
as deleted function, 508  
as deleted in derived class, 624  
`Bulk_quote`, 623  
multiple inheritance, 805  
virtual base class, 815  
virtual base classes, 815  
`volatile`, 857  
default constructor, 263, 306  
derived class, 623  
members of built-in type, 263  
destructor, 503, 550  
move operations  
deleted function, 538  
not always defined, 538
- T**
- `\t` (tab character), 39  
`tellp`, `tellg`, 763–768  
template  
  *see also* class template  
  *see also* function template  
  *see also* instantiation  
  declaration, 669  
link time errors, 657  
overview, 652  
parameter, *see* template parameter  
parameter list, 714  
template argument, 653, 714  
  explicit, 660, 713  
template member, *see* member template  
type alias, 666  
type transformation templates, 684, 714  
type-dependencies, 658  
variadic, *see* variadic template  
template argument deduction, 678, 714  
  compare, 680  
  explicit template argument, 682  
  function pointer, 686  
  limited conversions, 679  
  low-level const, 693  
  lvalue reference parameter, 687  
  multiple function parameters, 680  
  parameter with nontemplate type, 680  
  reference parameters, 687–689  
  rvalue reference parameter, 687  
  top-level const, 679  
template class, *see* class template  
template function, *see* function template  
template parameter, 653, 714  
  default template argument, 670  
  class template, 671  
  function template, 671  
  name, 668  
  restrictions on use, 669  
nontype parameter, 654, 714  
  must be constant expression, 655  
  type requirements, 655  
scope, 668  
template argument deduction, 680  
type parameter, 654, 654, 714  
  as friend, 666  
  used in template class, 660  
template parameter pack, 699, 714  
  expansion, 703  
  pattern, 703  
template specialization, 707, 706–712, 714  
  class template, 709–712  
  class template member, 711  
  compare function template, 706  
  compared to overloading, 708  
  declaration dependencies, 708  
  function template, 707  
  `hash<key_type>`, 709, 788  
  headers, 708  
  of namespace member, 709, 788  
  partial, class template, 711, 714  
  scope, 708  
  `template<>`, 707  
`template<>`  
  default template argument, 671  
  template specialization, 707  
temporary, 62, 80  
terminate function, 773, 818  
  exception handling, 196, 200  
  machine-dependent, 196  
terminology  
  const reference, 61  
  iterator, 109  
  object, 42  
  overloaded new and delete, 822  
test, `bitset`, 727  
`TextQuery`, 485  
  class definition, 487  
  constructor, 488  
  main program, 486  
  program design, 485  
  query, 489  
  revisited, 635

- this pointer, 257, 306
  - static members, 301
  - as argument, 266
  - in return, 260
  - overloaded
    - on const, 276
    - on lvalue or rvalue reference, 546
- throw, 193, 193, 200, 772, 818
  - execution flow, 196, 773
  - pointer to local object, 774
  - rethrow, 776, 818
  - runtime\_error, 194
- throw(), exception specification, 780
- tie member, ostream, 315
- to\_string, 368
- Token, 849
  - assignment operators, 850
  - copy control, 851
  - copyUnion, 851
  - default constructor, 850
  - discriminant, 850
- tolower, 92
- top
  - priority\_queue, 371
  - stack, 371
- top-level const, 64, 80
  - and auto, 69
  - argument and parameter, 212
  - decltype, 71
  - parameter, 232
  - template argument deduction, 679
- toupper, 92
- ToyAnimal, virtual base class, 815
- trailing return type, 229, 252
  - function template, 684
  - lambda expression, 396
  - pointer to array, 229
  - pointer to function, 250
- transform
  - algorithm, 396, 874
  - program, 442
- translation unit, 4
- trunc (file mode), 319
- try block, 193, 194, 200, 773, 818
- tuple, 718, 770
  - findBook, program, 721
  - equality and relational operators, 720
  - header, 718
  - initialization, 718
  - make\_tuple, 718
  - return value, 721
- value initialization, 718
- type
  - alias, 67, 80
    - template, 666
  - alias declaration, 68
  - arithmetic, 32, 78
  - built-in, 2, 26, 32–34
  - checking, 46, 80
    - argument and parameter, 203
    - array reference parameter, 217
    - function return value, 223
    - name lookup, 235
  - class, 19, 26
  - compound, 50, 50–58, 78
  - conversion, *see* conversion
  - dynamic, 601, 650
  - incomplete, 279, 306
  - integral, 32, 79
  - literal, 66
    - class type, 299
  - specifier, 41, 80
  - static, 601, 650
- type alias declaration, 68, 78, 80
  - pointer, to array, 229
  - pointer to function, 249
  - pointer to member, 839
  - template type, 666
- type independence, algorithms, 377
- type member, class, 271
- type parameter, *see* template parameter
- type transformation templates, 684, 714
  - type\_traits, 685
- type\_info, 864
  - header, 197
  - name, 831
  - no copy or assign, 831
  - operations, 831
  - returned from typeid, 827
- type\_traits
  - header, 684
  - remove\_pointer, 685
  - remove\_reference, 684
    - and move, 691
- type transformation templates, 685
- typedef, 67, 80
  - const, 68
  - and pointer, to const, 68
  - pointer, to array, 229
  - pointer to function, 249
- typeid operator, 826, 827, 864
  - returns type\_info, 827

`typeinfo` header, 826, 827, 831

`typename`

compared to class, 654

required for type member, 670

template parameter, 654

## U

unary operators, 134, 169

overloaded operator, 552

unary predicate, 386, 418

`unary_function` deprecated, 579

uncaught exception, 773

undefined behavior, 35, 80

base class destructor not virtual, 622

bitwise operators and signed values,  
153

caching `end()` iterator, 355

`cstring` functions, 122

dangling pointer, 463

default initialized members of built-in  
type, 263

`delete` of invalid pointer, 460

destination sequence too small, 382

element access empty container, 346

invalidated iterator, 107, 353

missing return statement, 224

misuse of smart pointer `get`, 466

omitting [] when deleting array, 479

operand order of evaluation, 138, 149

out-of-range subscript, 93

out-of-range value assigned to signed  
type, 35

overflow and underflow, 140

pointer casts, 163

pointer comparisons, 123

return reference or pointer to local  
variable, 225

`string` invalid initializer, 361

uninitialized

dynamic object, 458

local variable, 205

pointer, 54

variable, 45

using unconstructed memory, 482

using unmatched match object, 737

writing to a `const` object, 163

wrong deleter with smart pointer, 480

`underflow_error`, 197

unformatted IO, 761, 770

`istream`, 761

multi-byte, `istream`, 763

single-byte, `istream`, 761

`unget`, `istream`, 761

`uniform_int_distribution`, 746

`uniform_real_distribution`, 750

uninitialized, 8, 28, 44, 80

pointer, undefined behavior, 54

variable, undefined behavior, 45

uninitialized\_copy, 483

move iterator, 543

uninitialized\_fill, 483

`union`, 847, 864

anonymous, 848, 862

class type member, 848

assignment operators, 850

copy control, 851

default constructor, 850

deleted copy control, 849

placement new, 851

definition, 848

discriminant, 850

restrictions, 847

`unique`, 384, 878

list and `forward_list`, 415

`unique_copy`, 403, 878

`unique_ptr`, 450, 470–472, 491

\* (dereference), 451

copy and assignment, 470

definition, 470, 472

deleter, 472, 491

bound at compile time, 678

dynamically allocated array, 479

initialization, 470

pitfalls, 469

`release`, 470

`reset`, 470

return value, 471

transfer ownership, 470

with new, 470

`unitbuf`, manipulator, 315

unnamed namespace, 791, 818

local to file, 791

`replace_file static`, 792

unordered container, 443, 448

*see also* container

*see also* associative container

bucket management, 444

`hash<key_type>` specialization, 709,  
788

compatible with == (equality), 710

`key_type` requirements, 445

- override default hash, 446
- `unordered_map`, 448  
*see also* unordered container
  - \* (dereference), 429
  - [ ] (subscript), 435, 448
    - adds element, 435
    - at, 435
    - definition, 423
    - header, 420
    - list initialization, 423
    - word\_count program, 444
  - `unordered_multimap`, 448  
*see also* unordered container
    - \* (dereference), 429
    - definition, 423
    - has no subscript operator, 435
    - insert, 433
    - list initialization, 423
  - `unordered_multiset`, 448  
*see also* unordered container
    - insert, 433
    - iterator, 429
    - list initialization, 423
    - override default hash, 446
  - `unordered_set`, 448  
*see also* unordered container
    - header, 420
    - iterator, 429
    - list initialization, 423
  - unscoped enumeration, 832, 864
    - as union discriminant, 850
    - conversion to integer, 834
    - enum, 832
  - `unsigned`, 34, 80
    - char, 34
    - conversion, 36
    - conversion from `signed`, 34
    - conversion to `signed`, 160
    - literal (`numU` or `numu`), 41
    - size return type, 88
  - `unsigned type`, 34
  - unwinding, stack, 773, 818
  - `upper_bound`
    - algorithm, 873
    - ordered container, 438
    - used in `Basket`, 632
  - `uppercase`, manipulator, 755
  - use count, *see* reference count
  - user-defined conversion, *see* class type conversion
  - user-defined header, 76–77

## V

- `valid`, program, 740
- valid but unspecified, 537
- valid pointer, 52
- value initialization, 98, 132
  - dynamically allocated, object, 459
  - map subscript operator, 435
  - `new []`, 478
  - `resize`, 352
  - sequential container, 336
  - `tuple`, 718
  - uses default constructor, 293
  - `vector`, 98
- `value_type`
  - associative container, 428, 448
  - sequential container, 333
- valuelike class, copy control, 512
- `varargs`, 222
- variable, 8, 28, 41, 41–49, 80
  - `const`, 59
  - `constexpr`, 66
  - declaration, 45
    - class type, 294
    - define before use, 46
    - defined after label, 182, 192
    - definition, 41, 45
  - `extern`, 45

- extern and `const`, 60  
initialization, 42, 43, 79  
is lvalue, 533  
lifetime, 204  
local, 204, 252  
preprocessor, 79  
variadic template, 699, 714  
declaration dependencies, 702  
forwarding, 704  
    usage pattern, 706  
function matching, 702  
pack expansion, 702–704  
parameter pack, 699  
print program, 701  
recursive function, 701  
`sizeof . . .`, 700  
`vector`, 96–105, 132, 373  
    *see also* container  
    *see also* sequential container  
    *see also* iterator  
    `[ ]` (subscript), 103, 132, 347  
    `=` (assignment), list initialization, 145  
    `at`, 348  
    capacity, 356  
    capacity program, 357  
    definition, 97  
    `difference_type`, 112  
    `erase`, changes container size, 385  
    header, 96, 329  
initialization, 97–101, 334–337  
initialization from array, 125  
list initialization, 98, 336  
memory management, 355  
overview, 326  
`push_back`, invalidates iterator, 354  
random-access iterator, 412  
`reserve`, 356  
subscript range, 105  
`TextQuery` class, 485  
value initialization, 98, 336  
viable function, 243, 252  
    *see also* function matching  
virtual base class, 811, 818  
    ambiguities, 812  
    `Bear`, 812  
    class derivation list, 812  
    conversion, 812  
    derived class constructor, 813  
    `iostream`, 810  
    name lookup, 812  
    order of destruction, 815  
order of initialization, 814  
`ostream`, 810  
    `Raccoon`, 812  
    `ToyAnimal`, 815  
    `ZooAnimal`, 811  
virtual function, 592, 595, 603–610, 650  
    compared to run-time type identification, 829  
    default argument, 607  
    derived class, 596  
    destructor, 622  
    exception specification, 781  
    final specifier, 607  
    in constructor, destructor, 627  
    multiple inheritance, 807  
    overloaded function, 621  
    override, 595, 650  
    override specifier, 593, 596, 606  
    overriding run-time binding, 607  
    overview, 595  
    pure, 609  
    resolved at run time, 604, 605  
    return type, 606  
    scope, 620  
    type-sensitive equality, 829  
virtual inheritance, *see* virtual base class  
Visual Studio, 5  
`void`, 32, 80  
    return type, 223  
`void*`, 56, 80  
    conversion from pointer, 161  
`volatile`, 856, 864  
    pointer, 856  
    synthesized copy-control members, 857  
vowel counting, program, 179

## W

- `wcerr`, 311  
`wchar_t`, 33  
    literal, 40  
`wchar_t` streams, 311  
`wcin`, 311  
`wcout`, 311  
weak ordering, strict, 448  
`weak_ptr`, 450, 473–475, 491  
    definition, 473  
    initialization, 473  
    `lock`, 473  
`StrBlobPtr`, 474

wfstream, 311  
what\_exception, 195, 782  
while statement, 11, 28, 183, 183–185, 200  
    condition, 12, 183  
wide character streams, 311  
wifstream, 311  
window, console, 6  
Window\_mgr, 279  
wiostream, 311  
wistream, 311  
wistringstream, 311  
wofstream, 311  
word, 33, 80  
word\_count program  
    map, 421  
    set, 422  
    unordered\_map, 444  
word\_transform program, 441  
WordQuery, 637, 642  
wostream, 311  
wostringstream, 311  
wregex, 733  
write, ostream, 763  
wstringstream, 311

## X

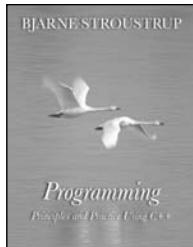
\xnnn (hexadecimal escape sequence), 39

## Z

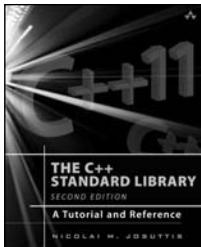
ZooAnimal  
    program, 802  
    virtual base class, 811

*This page intentionally left blank*

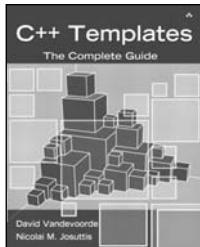
# Take the Next Step to Mastering C++



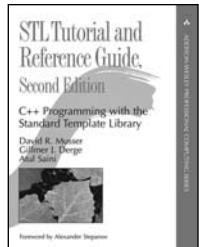
978-0-321-54372-1



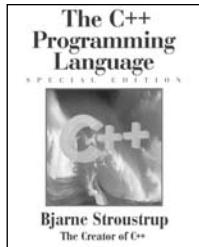
978-0-321-62321-8



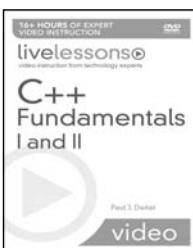
978-0-201-73484-3



978-0-321-38384-8



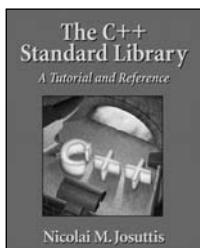
978-0-201-70073-2



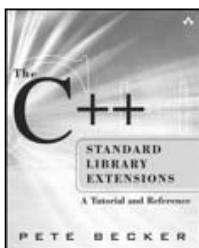
978-0-13-704483-2



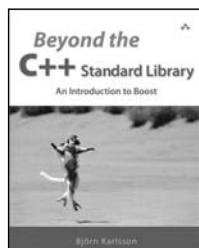
978-0-13-700130-9



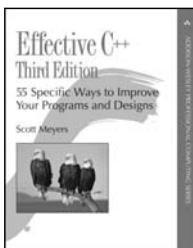
978-0-201-37926-6



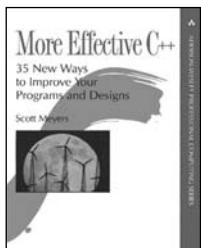
978-0-321-41299-7



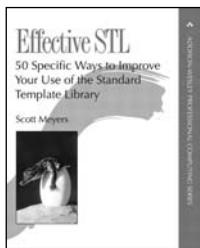
978-0-321-13354-0



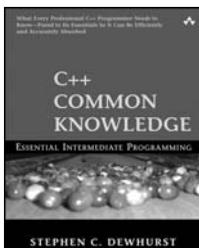
978-0-321-33487-9



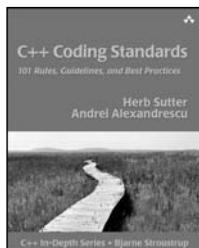
978-0-201-63371-9



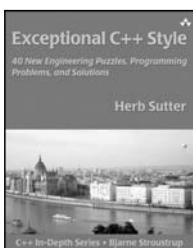
978-0-201-74962-5



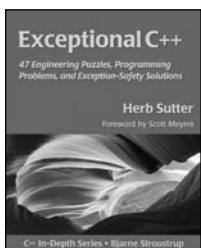
978-0-321-32192-3



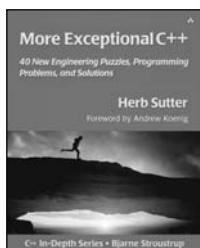
978-0-321-11358-0



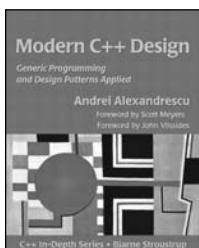
978-0-201-76042-2



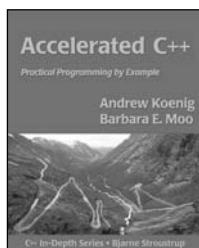
978-0-201-61562-3



978-0-201-70434-1



978-0-201-70431-0



978-0-201-70353-5



For more information on these titles  
visit [informit.com](http://informit.com)