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.

Chapter 17. Specialized Library Facilities

Contents

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

C++ 11

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. 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.

Table 17.1. Operations on `tuples`

`tuple<T1, T2, ..., Tn> t;`	`t` is a `tuple` with as many members as there are types `T1` ... `Tn`. The members are value initialized (§ 3.3.1, p. 98).
`tuple<T1, T2, ..., Tn> t(v1, v2, ..., vn);`	`t` is a `tuple` with types `T1` ... `Tn` in which each member is initialized from the corresponding initializer, v_i. This constructor is `explicit` (§ 7.5.4, p. 296).
`make_tuple(v1, v2, ..., vn)`	Returns a `tuple` initialized from the given initializers. The type of the `tuple` is inferred from the types of the initializers.
`t1 == t2` `t1 != t2`	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.
`t1` *relop* `t2`	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
`get<i>(t)`	Returns a reference to the `i`th 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`.
`tuple_size<`*tupleType*`>::value`	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.
`tuple_element<i, `*tupleType*`>::type`	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.

Note

A `tuple` can be thought of as a "quick and dirty" data structure.

17.1.1. Defining and Initializing `tuple`s

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

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
// 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>`.

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:

Click here to view code image

```
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:

Click here to view code image

```
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 or 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:

Click here to view code image

```
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
```

Note

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`:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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->cend(),
                                  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:

Click here to view code image

```
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:

Click here to view code image

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

Table 17.2. Ways to Initialize a bitset

`bitset<n> b;`	`b` has `n` bits; each bit is 0. This constructor is a `constexpr` (§ 7.5.6, p. 299).
`bitset<n> b(u);`	`b` is a copy of the `n` low-order bits of `unsigned long long` value `u`. If `n` is greater than the size of an `unsigned long long`, the high-order bits beyond those in the `unsigned long long` are set to zero. This constructor is a `constexpr` (§ 7.5.6, p. 299).
`bitset<n> b(s, pos, m, zero, one);`	`b` is a copy of the `m` characters from the `string` `s` starting at position `pos`. `s` may contain only the characters `zero` and `one`; if `s` contains any other character, throws `invalid_argument`. The characters are stored in `b` as `zeor` and `one`, respectively. `pos` defaults to 0, `m` defaults to `string::npos`, `zero` defaults to `'0'`, and `one` defaults to `'1'`.
`bitset<n> b(cp, pos, m, zero, one);`	Same as the previous constructor, but copies from the character array to which `cp` points. If `m` is not supplied, then `cp` must point to a C-style string. If `m` is supplied, there must be at least `m` characters that are `zero` or `one` starting at `cp`.

The constructors that take a `string` or character pointer are `explicit` (§ 7.5.4, p. 296). The ability to specify alternate characters for 0 and 1 was added in the new standard.

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 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:

Click here to view code image

```
// 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 00001011111011101111
// on machines with 64-bit long long 0ULL is 64 bits of 0, so ~0ULL is 64 ones
bitset<128> bitvec3(~0ULL); // 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:

Click here to view code image

```
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.

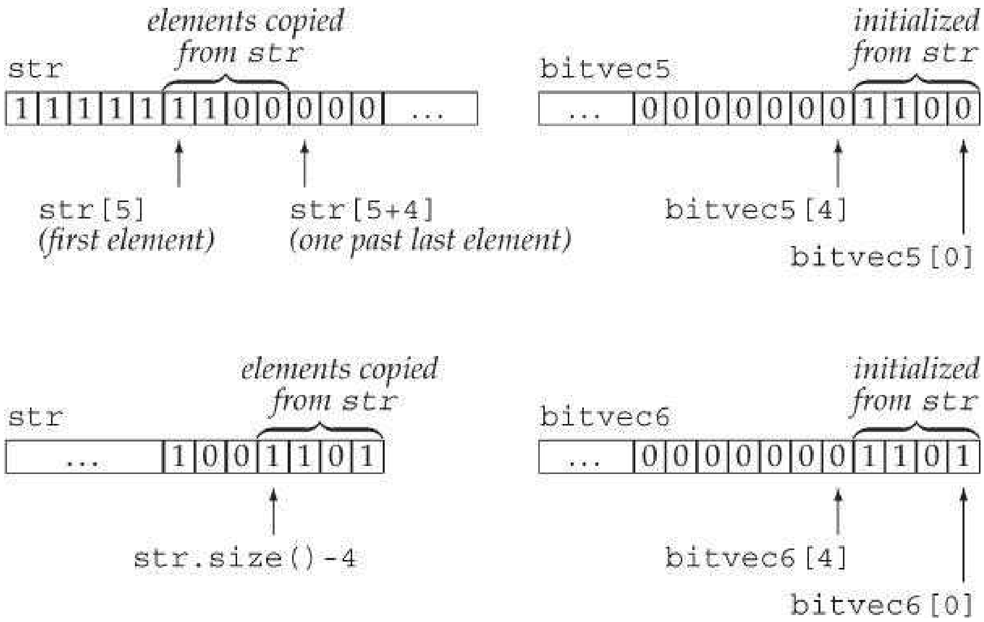
> **Note**
>
> The indexing conventions of `strings` and `bitsets` are inversely related: The character in the `string` with the highest subscript (the rightmost character) is used to initialize the low-order bit in the `bitset` (the bit with subscript 0). When you initialize a `bitset` from a `string`, 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:

Click here to view code image

```
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:

(a) `bitset<64> bitvec(32);`

(b) `bitset<32> bv(1010101);`

(c) `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

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

Click here to view code image

```
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
```

C++ 11

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:

Click here to view code image

```
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:

Click here to view code image

```
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`:

Click here to view code image

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

Note

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:

Click here to view code image

```
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:

Click here to view code image

```
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.

Table 17.4. Regular Expression Library Components

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

C++ 11

Tip

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.

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.

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) (*seq*, 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.

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*":

Click here to view code image

```
// 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.

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.

Table 17.6. regex (and wregex) Operations

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

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`

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

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

Click here to view code image

```
// 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.

> **Note**
>
> 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.

Table 17.7. Regular Expression Error Conditions

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

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

Click here to view code image

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

When run on our system, this program generates

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

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.

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:

Click here to view code image

```
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;
```

Table 17.8. Regular Expression Library Classes

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

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:

Click here to view code image

```
cmatch results;  // will match character array input sequences
if (regex_search("myfile.cc", results, r))
    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).

Table 17.9. sregex_iterator Operations

These operations also apply to `cregex_iterator`, `wsregex_iterator`, and `wcregex_iterator`	
`sregex_iterator it(b, e, r);`	`it` is an `sregx_iterator` that iterates through the `string` denoted by iterators `b` and `e`. Calls `regex_search(b, e, r)` to position `it` on the first match in the input.
`sregex_iterator end;`	Off-the-end iterator for `sregex_iterator`.
`*it` `it->`	Returns a reference to the `smatch` object or a pointer to the `smatch` object from the most recent call to `regex_search`.
`++it` `it++`	Calls `regex_search` 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.
`it1 == it2` `it1 != it2`	Two `sregex_iterators` are equal if they are both the off-the-end iterator. Two non-end iterators are equal if they are constructed from the same input sequence and `regex` object.

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:

Click here to view code image

```
// 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.

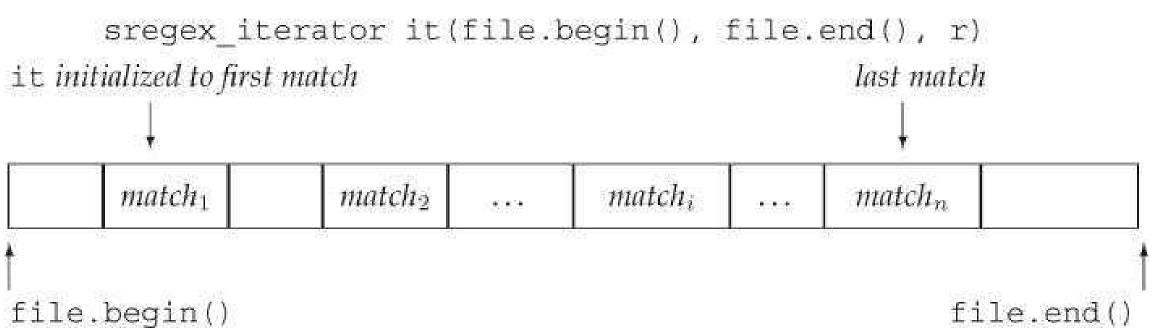


Figure 17.1. Using an sregex_iterator

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

Click here to view code image

```
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 17.10. smatch Operations

These operations also apply to the `cmatch`, `wsmatch`, `wcmatch` and the corresponding `csub_match`, `wssub_match`, and `wcsub_match` types.	
`m.ready()`	`true` if `m` has been set by a call to `regex_search` or `regex_match`; `false` otherwise. Operations on `m` are undefined if `ready` returns `false`.
`m.size()`	Zero if the match failed; otherwise, one plus the number of subexpressions in the most recently matched regular expression.
`m.empty()`	`true` if `m.size()` is zero.
`m.prefix()`	An `ssub_match` representing the sequence before the match.
`m.suffix()`	An `ssub_match` representing the part after the end of the match.
`m.format(...)`	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.	
`m.length(n)`	Size of the `n`th matched subexpression.
`m.position(n)`	Distance of the `n`th subexpression from the start of the sequence.
`m.str(n)`	The matched `string` for the `n`th subexpression.
`m[n]`	`ssub_match` object corresponding to the `n`th subexpression.
`m.begin()`, `m.end()` `m.cbegin()`, `m.cend()`	Iterators across the `sub_match` elements in `m`. As usual, `cbegin` and `cend` return `const_iterator`s.

Table 17.11. Submatch Operations

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

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:

Click here to view code image

```
// 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\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.

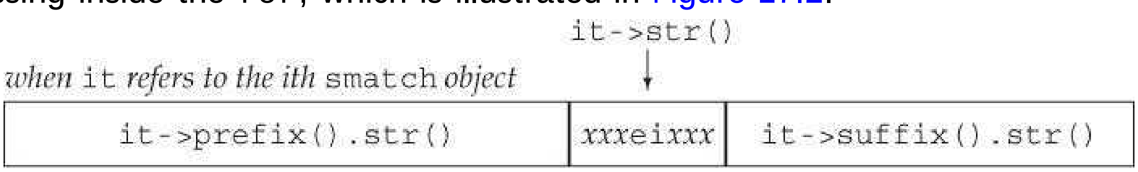


Figure 17.2. The smatch Object Representing a Particular Match

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.

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.

Exercises Section 17.3.2

Exercise 17.17: Update your program so that it finds all the words in an input sequence that violiate 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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:

- `\{d}` represents a single digit and `\{d}{`*n*`}` represents a sequence of *n* digits. (E.g., `\{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., `[-. ]` 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., `\{d}{3}[-. ]?\{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 `\(`or `\)`.

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

Click here to view code image

```
// 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
"(\\()?(\\d{3})(\\))?([-. ])?(\\d{3})([-. ]?)(\\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. `(\\()?` an optional open parenthesis for the area code
2. `(\\d{3})` the area code
3. `(\\))?` an optional close parenthesis for the area code
4. `([-. ])?` an optional separator after the area code
5. `(\\d{3})` the next three digits of the number
6. `([-. ])?` another optional separator
7. `(\\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:

Click here to view code image

```
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:

Click here to view code image

```
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();
}
```

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.

Table 17.12. Regular Expression Replace Operations

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

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:

Click here to view code image

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

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

Click here to view code image

```
regex r(phone);  // a regex to find our pattern
string number = "(908) 555-1800";
cout << regex_replace(number, r, fmt) << endl;
```

The output from this program is

```
908.555.1800
```

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:

Click here to view code image

```
morgan (201) 555-2368 862-555-0123
```

drew (973)555.0130
lee (609) 555-0132 2015550175 800.555-0000

that we want to transform to data like this:

Click here to view code image

morgan 201.555.2368 862.555.0123
drew 973.555.0130
lee 609.555.0132 201.555.0175 800.555.0000

We can generate this transformation with the following program:

Click here to view code image

```
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`.

Table 17.13. Match Flags

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

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:

Click here to view code image

```
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):

Click here to view code image

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

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`:

Click here to view code image

```
// 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

Click here to view code image

201.555.2368 862.555.0123
973.555.0130
609.555.0132 201.555.0175 800.555.0000

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 `ddddd-dddd`.

17.4. Random Numbers

C++ 11

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.

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 clases 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.

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

Best Practices

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

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:

Click here to view code image

```
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:

Click here to view code image

```
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).

Table 17.15. Random Number Engine Operations

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

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:

Click here to view code image

```
// 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.

> **Note**
>
> 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:

Click here to view code image

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

On our system this program produces the following output:

min: 1 max: 2147483646

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:

Click here to view code image

```
// 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:

Click here to view code image

```
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 `vector`s `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):

Click here to view code image

```
// 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.

> **Warning**
>
> 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:

Click here to view code image

```
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:

Click here to view code image

```
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.

Warning

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.

Table 17.16. Distribution Operations

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

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:

Click here to view code image

```
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:

Click here to view code image

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

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):

Click here to view code image

```
// 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:

Click here to view code image

```
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

Click here to view code image

```
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:

Click here to view code image

```
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.

> **Warning**
>
> 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:

Click here to view code image

```
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.

Warning

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:

Click here to view code image

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

When executed, this program generates the following:

Click here to view code image

```
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 `bool`s will print them as either `true` or `false`.

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

Click here to view code image

```
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`:

Click here to view code image

```
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.

Note

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`:

Click here to view code image

```
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:

Click here to view code image

```
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.

> **Note**
>
> 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:

Click here to view code image

```
// 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:

Click here to view code image

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

Table 17.17. Manipulators Defined in iostream

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

> **Best Practices**
>
> 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.

C++ 11

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 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:

Click here to view code image

```
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: 0x1.1ad7bcp+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:

Click here to view code image

```
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.

Note

> `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:

Click here to view code image

```
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

`setfill(ch)`	Fill whitespace with `ch`
`setprecision(n)`	Set floating-point precision to `n`
`setw(w)`	Read or write value to `w` characters
`setbase(b)`	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:

Click here to view code image

```
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

`is.get(ch)`	Put the next byte from the `istream` `is` in character `ch`. Returns `is`.
`os.put(ch)`	Put the character `ch` onto the `ostream` `os`. Returns `os`.
`is.get()`	Returns next byte from `is` as an `int`.
`is.putback(ch)`	Put the character `ch` back on `is`; returns `is`.
`is.unget()`	Move `is` back one byte; returns `is`.
`is.peek()`	Return the next byte as an `int` 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 `iostream` 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:

Click here to view code image

```
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.

Table 17.20. Multi-Byte Low-Level IO Operations

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

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`.

> **Warning**
>
> 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.

> **Note**
>
> 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 *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.

> **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 `char`s are implemented as `unsigned char`s, this loop will run forever:

Click here to view code image

```
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 `char`s are implemented as `signed char`s, 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.

> **Warning**
>
> 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 *seek*ing to a given position; the second *tell*s 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

`tellg()` `tellp()`	Return the current position of the marker in an input stream (`tellg`) or an output stream (`tellp`).
`seekg(pos)` `seekp(pos)`	Reposition the marker in an input or output stream to the given absolute address in the stream. `pos` is usually a value returned by a previous call to the corresponding `tellg` or `tellp` function.
`seekp(off, from)` `seekg(off, from)`	Reposition the marker for an input or output stream integral number `off` characters ahead or behind `from`. `from` can be one of • `beg`, seek relative to the beginning of the stream • `cur`, seek relative to the current position of the stream • `end`, seek relative to the end of the stream

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.

> **Note**
>
> 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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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. Specifying `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. Formatted 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.

Chapter 18. Tools for Large Programs

Contents

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 `try`s. 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 `try`s 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.

Note

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.

> **Warning**
>
> 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.

Warning

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 `throw`s?

(a) `range_error r("error");`

`throw r;`

(b) `exception *p = &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:

Click here to view code image

```
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.

Best Practices

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.

> **Note**
>
> 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:

Click here to view code image

```
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:

Click here to view code image

```
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.

Note

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 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.

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.

Click here to view code image

```
try {
    //  use of the C++ standard library
} catch(exception) {
    //  ...
} catch(const runtime_error &re) {
    //  ...
} catch(overflow_error eobj) { /* ... */ }
```

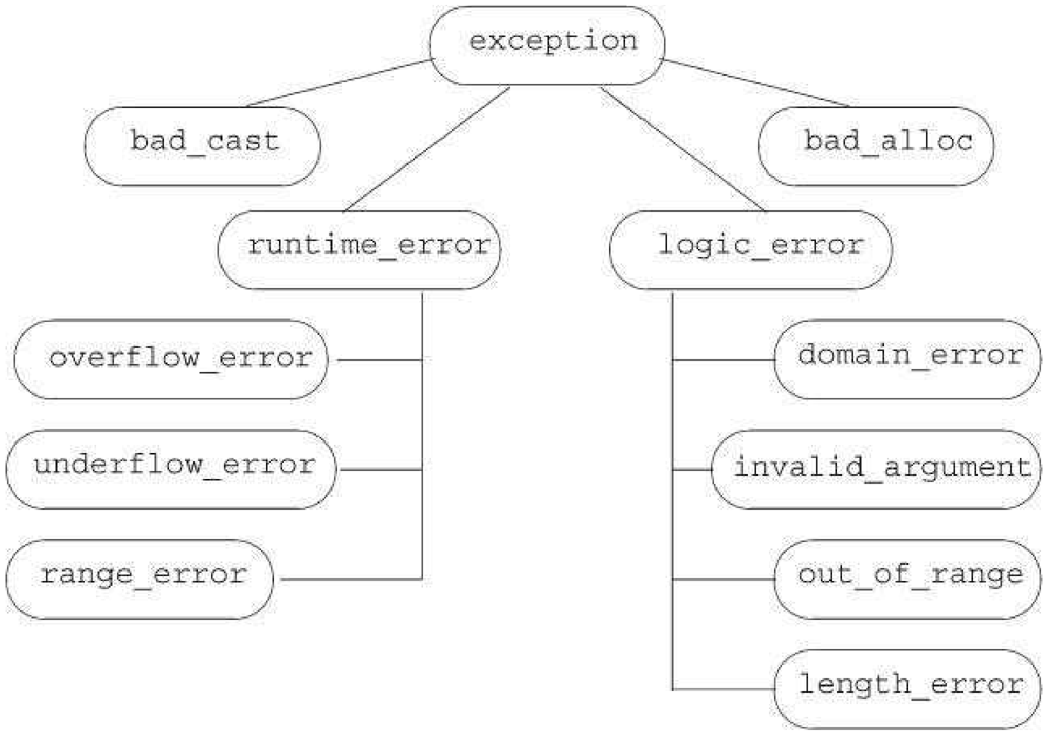


Figure 18.1. Standard exception Class Hierarchy

Exercise 18.5: Modify the following `main` function to catch any of the exception types shown in Figure 18.1 (p. 783):

Click here to view code image

```
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) { }
```

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:

Click here to view code image

```
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.

Note

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.

C++ 11

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:

Click here to view code image

```
void recoup(int) noexcept;    // won't throw
void alloc(int);              // might throw
```

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):

Click here to view code image

```
// 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.

Warning

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:

Click here to view code image

```
void recoup(int) noexcept;   // recoup doesn't throw
void recoup(int) throw();    // equivalent declaration
```

These declarations of `recoup` are equivalent. 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:

Click here to view code image

```
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`:

Click here to view code image

```
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:

Click here to view code image

```
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.

Note

`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:

Click here to view code image

```
// 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:

Click here to view code image

```
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.

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

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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 assigment 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:

Click here to view code image

```
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:

Click here to view code image

```
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.

Note

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:

Click here to view code image

```
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:

Click here to view code image

```
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.

> **Best Practices**
>
> 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`:

Click here to view code image

```
// ---- 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:

Click here to view code image

```
// ---- 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:

Click here to view code image

```
#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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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

Click here to view code image

```
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`:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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.

> **Note**
>
> 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:

Click here to view code image

```
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):

Click here to view code image

```
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.

Warning

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`:

Click here to view code image

```
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

Click here to view code image

```
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:

Click here to view code image

```
namespace Qlib = cplusplus_primer::QueryLib;
Qlib::Query q;
```

Note

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.

> **Warning**
>
> 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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.

Tip

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`.

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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`.

Tip

> 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:

Click here to view code image

```
using std::operator>>;            // needed to allow cin >> s
```

or we would have to use the function-call notation in order to include the namespace qualifer:

Click here to view code image

```
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:

Click here to view code image

```
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`:

Click here to view code image

```
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.

Click here to view code image

```
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*:

Click here to view code image

```
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):

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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?

Click here to view code image

```
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:

Click here to view code image

```
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 non`static` data members, if any, declared within the `Panda` class.

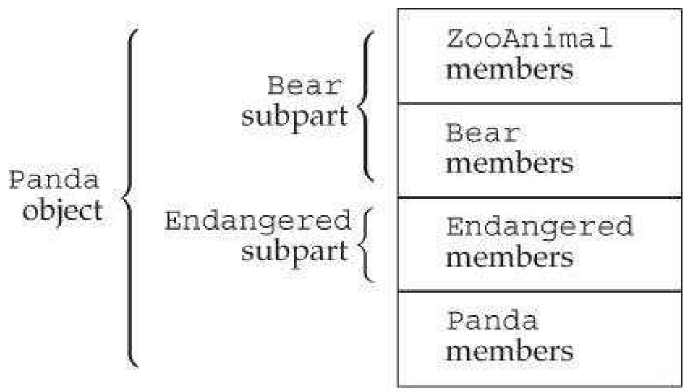


Figure 18.2. Conceptual Structure of a Panda Object

Derived Constructors Initialize All Base Classes

Constructing an object of derived type constructs and initializes all its base subobjects. 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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:

Click here to view code image

```
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`:

Click here to view code image

```
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 refer to a derived object. For example, a pointer or reference to `ZooAnimal`, `Bear`, or `Endangered` can be bound to a `Panda` object:

Click here to view code image

```
// 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
```

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:

Click here to view code image

```
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;
```

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`:

Click here to view code image

```
void print(const Bear&);
```

```
void print(const Endangered&);
```

an unqualified call to `print` with a Panda object would be a compile-time error:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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`
	`Bear::Bear`
	`Endangered::Endangered`
	`Panda::Panda`
`highlight`	`Endangered::Endangered`
	`Panda::Panda`
`toes`	`Bear::Bear`
	`Panda::Panda`
`cuddle`	`Panda::Panda`
destructor	`ZooAnimal::ZooAnimal`
	`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:

Click here to view code image

```
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.

> **Warning**
>
> 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:

Click here to view code image

```
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:

Click here to view code image

```
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

Click here to view code image

```
struct Base1 {
    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 Base1 {
    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.

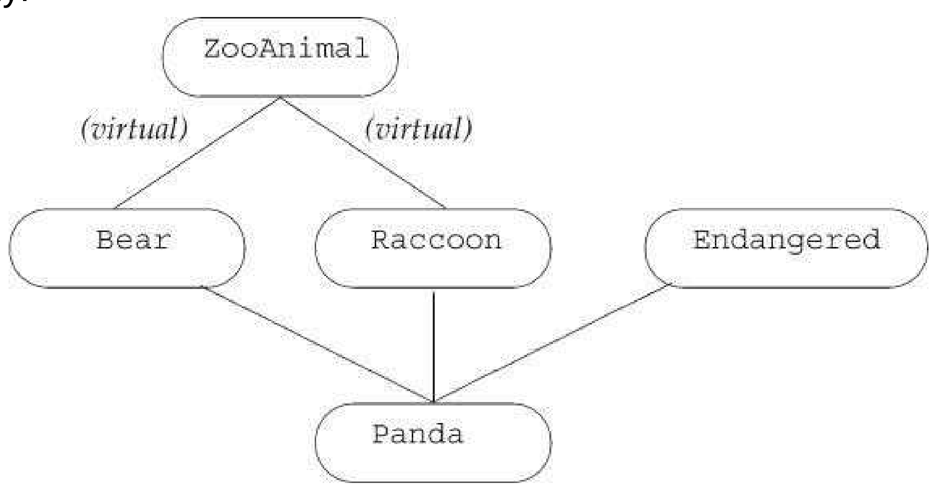


Figure 18.3. Virtual Inheritance Panda Hierarchy

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.

Note

Virtual derivation affects the classes that subsequently derive from a class with a virtual base; it doesn't affect the derived class itself.

Using a Virtual Base Class

We specify that a base class is virtual by including the keyword `virtual` in the derivation list:

Click here to view code image

```
// 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:

Click here to view code image

```
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:

Click here to view code image

```
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.

Click here to view code image

```
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:

Click here to view code image

```
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`:

Click here to view code image

```
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.

> **Note**
>
> 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`:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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 nonreference 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 initizlier 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 namepsace 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 subseuquently 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.

Chapter 19. Specialized Tools and Techniques

Contents

The first three parts of this book discussed apects 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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.

Warning

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`:

Click here to view code image

```
// 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:

Click here to view code image

```
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:

Click here to view code image

```
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

Click here to view code image

```
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.

Note

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:

Click here to view code image

```
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.

Note

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.

Warning

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:

Click here to view code image

```
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.

> **Note**
>
> 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.

Best Practices

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_casts

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:

Click here to view code image

```
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:

Click here to view code image

```
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&`:

Click here to view code image

```
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:

Click here to view code image

```
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`:

Click here to view code image

```
// 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*.

Warning

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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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.

Table 19.1. Operations on type_info

`t1 == t2`	Returns `true` if the `type_info` objects `t1` and `t2` refer to the same type, `false` otherwise.
`t1 != t2`	Returns `true` if the `type_info` objects `t1` and `t2` refer to different types, `false` otherwise.
`t.name()`	Returns a C-style character string that is a printable version of the type name. Type names are generated in a system-dependent way.
`t1.before(t2)`	Returns a `bool` that indicates whether `t1` comes before `t2`. The ordering imposed by `before` is compiler dependent.

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`.

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:

Click here to view code image

```
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:

Click here to view code image

i, A10_i, 10Sales_data, Ss, P4Base, 7Derived

Note

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?

Click here to view code image

```
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:

C++ 11

Click here to view code image

```
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`:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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

C++ 11

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:

Click here to view code image

```
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 `enum`s have `int` as the underlying type. There is no default for unscoped `enum`s; 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

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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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 non`static` 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):

Click here to view code image

```
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:

Click here to view code image

```
// pdata can point to a string member of a const (or non const) 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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
// 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`

Click here to view code image

```
// 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:

Click here to view code image

```
// 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

Click here to view code image

```
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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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.

Note

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`:

Click here to view code image

```
// 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:

Click here to view code image

```
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:

Click here to view code image

```
// 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`:

Click here to view code image

```
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
```

Note

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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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?

Click here to view code image

```
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 `string`s, the obvious call won't work:

Click here to view code image

```
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

Click here to view code image

```
// 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):

Click here to view code image

```
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

Click here to view code image

```
// 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

Click here to view code image

```
// 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:

Click here to view code image

```
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

C++ 11

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:

Click here to view code image

```
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:

Click here to view code image

```
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
```

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:

Click here to view code image

```
// 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`:

Click here to view code image

```
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`.

Click here to view code image

```
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:

Click here to view code image

```
// 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.

Warning

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:

Click here to view code image

```
// 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

Click here to view code image

```
// 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:

Click here to view code image

```
// 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`

Click here to view code image

```
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 `union`s. 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 `struct`s, 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

`union`s 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:

Click here to view code image

```
// 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 `union`s 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:

Click here to view code image

```
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:

Click here to view code image

```
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.

> **Note**
>
> An anonymous `union` cannot have `private` or `protected` members, nor can an anonymous `union` define member functions.

`union`s with Members of Class Type

C++ 11

Under earlier versions of C++, `union`s 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, `union`s with members that define their own constructors and/or copy-control members are more complicated to use than `union`s 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 `union`s 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 `union`s 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:

Click here to view code image

```
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&);
    // 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`:

Click here to view code image

```
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:

Click here to view code image

```
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`:

Click here to view code image

```
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`:

Click here to view code image

```
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 `union`s 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.

> **Note**
>
> 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:

Click here to view code image

```
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.

Click here to view code image

```
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.

Note

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:

Click here to view code image

```
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.

> **Warning**
>
> 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:

Click here to view code image

```
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):

Click here to view code image

```
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:

Click here to view code image

```
inline bool File::isRead() const { return mode & READ; }
inline void File::setWrite() { mode |= WRITE; }
```

19.8.2. volatile Qualifier

Warning

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:

Click here to view code image

```
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:

Click here to view code image

```
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 (non`volatile`) `const`, and we cannot bind a non`volatile` 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`:

Click here to view code image

```
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.

> **Note**
>
> 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:

Click here to view code image

```
// 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

Click here to view code image

```
// 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.

> **Note**
>
> 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:

Click here to view code image

```
// 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:

Click here to view code image

```
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
```

> **Warning**
>
> 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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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:

Click here to view code image

```
// 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++:

Click here to view code image

```
#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:

Click here to view code image

```
// 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:

Click here to view code image

```
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:

Click here to view code image

```
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, compacted 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 reference 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.

Appendix A. The Library

Contents