18

Visiting with the New C++ Standard

In this chapter you'll review C++11 features previously covered and then learn about the following:

- Move semantics and rvalue references
- Lambda expressions
- The `function` wrapper template
- Variadic templates

This chapter concentrates on the new C++11 changes to the C++ language. The book already has covered several C++11 features, and we'll begin by reviewing them. Then we'll look at some additional features in some detail. Next, we'll identify some of the C++11 additions that are beyond the scope of this book. (Given that the C++11 draft is over 80% longer than C++98, we won't cover everything.) Finally, we'll briefly examine the BOOST library.

C++11 Features Revisited

By now you may have lost track of the many C++11 changes we already have encountered. This section reviews them briefly.

New Types

C++11 adds the `long long` and `unsigned long long` types to support 64-bit integers (or wider) and the `char16_t` and `char32_t` types to support 16-bit and 32-bit character representations, respectively. It also adds the "raw" string. Chapter 3, "Dealing with Data," discusses these additions.

Uniform Initialization

C++11 extends the applicability of the brace-enclosed list (*list-initialization*) so that it can be used with all built-in types and with user-defined types (that is, class objects). The list can be used either with or without the = sign:

```
int x = {5};
double y {2.75};
short quar[5] {4,5,2,76,1};
```

Also the list-initialization syntax can be used in `new` expressions:

```
int * ar = new int [4] {2,4,6,7};          // C++11
```

With class objects, a braced list can be used instead of a parenthesized list to invoke a constructor:

```
class Stump
{
private:
    int roots;
    double weight;
public:
     Stump(int r, double w) : roots(r), weight(w) {}
};
Stump s1(3,15.6);       // old style
Stump s2{5, 43.4};      // C++11
Stump s3 = {4, 32.1};   // C++11
```

However, if a class has a constructor whose argument is a `std::initializer_list` template, then only that constructor can use the list-initialization form. Various aspects of list-initialization were discussed in Chapters 3, 4, 9, 10, and 16.

Narrowing

The initialization-list syntax provides protection against narrowing—that is, against assigning a numeric value to a numeric type not capable of holding that value. Ordinary initialization allows you to do things that may or may not make sense:

```
char c1 = 1.57e27;   // double-to-char, undefined behavior
char c2 = 459585821; // int-to-char, undefined behavior
```

If you use initialization-list syntax, however, the compiler disallows type conversions that attempt to store values in a type "narrower" than the value:

```
char c1 {1.57e27};    // double-to-char, compile-time error
char c2 = {459585821};// int-to-char,out of range, compile-time error
```

However, conversions to wider types are allowed. Also a conversion to a narrower type is allowed if the value is within the range allowed by the type:

```
char c1 {66};     // int-to-char, in range, allowed
double c2 = {66}; // int-to-double, allowed
```

std::initializer_list

C++11 provides an `initializer_list` template class (discussed in Chapter 16, "The `string` Class and the Standard Template Library") that can be used as a constructor argument. If a class has such a constructor, the brace syntax can be used only with that constructor. The items in the list should all be of the same type or else be convertible to the same type. The STL containers provide constructors with `initializer_list` arguments:

```
vector<int> a1(10);    // uninitialized vector with 10 elements
vector<int> a2{10};    // initializer-list, a2 has 1 element set to 10
vector<int> a3{4,6,1}; // 3 elements set to 4,6,1
```

The `initializer_list` header file provides support for this template class. The class has `begin()` and `end()` member functions specifying the range of the list. You can use an `initializer_list` argument for regular functions as well as for constructors:

```
#include <initializer_list>
double sum(std::initializer_list<double> il);
int main()
{
   double total = sum({2.5,3.1,4});  // 4 converted to 4.0
...
}
double sum(std::initializer_list<double> il)
{
    double tot = 0;
    for (auto p = il.begin(); p !=il.end(); p++)
        tot += *p;
    return tot;
}
```

Declarations

C++11 implements several features that simplify declarations, particularly for situations arising when templates are used.

auto

C++11 strips the keyword `auto` of its former meaning as a storage class specifier (Chapter 9, "Memory Models and Namespaces") and puts it to use (Chapter 3) to implement automatic type deduction, provided that an explicit initializer is given. The compiler sets the type of the variable to the type of the initialization value:

```
auto maton = 112;  // maton is type int
auto pt = &maton;  // pt is type int *
double fm(double, int);
auto pf = fm;      // pf is type double (*)(double,int)
```

The auto keyword can simplify template declarations too. For example, if il is an object of type std::initializer_list<double>, you can replace

```
for (std::initializer_list<double>::iterator p = il.begin();
                                             p !=il.end(); p++)
```

with this:

```
for (auto p = il.begin(); p !=il.end(); p++)
```

decltype

The decltype keyword creates a variable of the type indicated by an expression. The following statement means "make y the same type as x," where x is an expression:

```
decltype(x) y;
```

Here are a couple of examples:

```
double x;
int n;
decltype(x*n) q; // q same type as x*n, i.e., double
decltype(&x) pd; // pd same as &x, i.e., double *
```

This is particularly useful in template definitions, when the type may not be determined until a specific instantiation is made:

```
template<typename T, typename U)
void ef(T t, U u)
{
    decltype(T*U) tu;
    ...
}
```

Here, tu is of whatever type results from the operation T*U, assuming that operation is defined. For example, if T is char and U is short, tu would be of type int because of the automatic integer promotions that take place in integer arithmetic.

The workings of decltype are more complicated than those of auto, and the resulting types can be references and can be const-qualified, depending on the expressions used. Here are some more examples:

```
int j = 3;
int &k = j
const int &n = j;
decltype(n) i1;      // i1 type const int &
decltype(j) i2;      // i2 type int
decltype((j)) i3;    // i3 type int &
decltype(k + 1) i4;  // i4 type int
```

See Chapter 8, "Adventures in Functions," for the rules that lead to these results.

Trailing Return Type

C++11 introduces a new syntax for declaring functions, one in which the return type comes after the function name and parameter list instead of before them:

```
double f1(double, int);          // traditional syntax
auto f2(double, int) -> double;  // new syntax, return type is double
```

The new syntax may look like a step backwards in readability for the usual function declarations, but it does make it possible to use `decltype` to specify template function return types:

```
template<typename T, typename U)
auto eff(T t, U u) -> decltype(T*U)
{
    ...
}
```

The problem that's addressed here is that `T` and `U` are not in scope before the compiler reads the `eff` parameter list, so any use of `decltype` has to come after the parameter list. The new syntax makes that possible.

Template Aliases: `using =`

It's handy to be able to create aliases for long or complex type identifiers. C++ already had `typedef` for that purpose:

```
typedef std::vector<std::string>::iterator itType;
```

C++11 provides a second syntax (discussed in Chapter 14, "Reusing Code in C++") for creating aliases:

```
using itType = std::vector<std::string>::iterator;
```

The difference is that the new syntax also can be used for partial template specializations, whereas `typedef` can't:

```
template<typename T>
  using arr12 = std::array<T,12>;  // template for multiple aliases
```

This statement specializes the `array<T,int>` template by setting the `int` parameter to `12`. For instance, consider these declarations:

```
std::array<double, 12> a1;
std::array<std::string, 12> a2;
```

They can be replaced with the following:

```
arr12<double> a1;
arr12(std::string> a2;
```

nullptr

The null pointer is a pointer guaranteed not to point to valid data. Traditionally, C++ has represented this pointer in source code with `0`, although the internal representation could be different. This raises some problems because it makes `0` both a pointer constant and an integer constant. As discussed in Chapter 12, "Classes and Dynamic Memory Allocation," C++11 introduces the keyword `nullptr` to represent the null pointer; it is a pointer type and not convertible to an integer type. For backward compatibility, C++ still allows the use of `0`, and the expression `nullptr == 0` evaluates as `true`, but using `nullptr` instead of `0` provides better type safety. For example, the value `0` could be passed to a function accepting an `int` argument, but the compiler will identify an attempt to pass `nullptr` to such a function as an error. So, for clarity and added safety, use `nullptr` if your compiler accepts it.

Smart Pointers

A program that uses `new` to allocate memory from the heap (or free store) should use `delete` to free that memory when it no longer is needed. Earlier, C++ introduced the `auto_ptr` smart pointer to help automate the process. Subsequent programming experience, particularly with the STL, indicated that something more sophisticated was needed. Guided by the experience of programmers and by solutions provided by the BOOST library, C++11 deprecates `auto_ptr` and introduces three new smart pointer types: `unique_ptr`, `shared_ptr`, and `weak_ptr`. Chapter 16 discusses the first two.

All the new smart pointers have been designed to work with STL containers and with move semantics.

Exception Specification Changes

C++ provides a syntax for specifying what exceptions, if any, a function may throw (refer to Chapter 15, "Friends, Exceptions, and More"):

```
void f501(int) throw(bad_dog);    // can throw type bad_dog exception
void f733(long long) throw();     // doesn't throw an exception
```

As with `auto_ptr`, the collective experience of the C++ programming community is that, in practice, exception specifications didn't work as well as intended. Thus, the C++11 standard deprecates exception specifications. However, the standards committee felt that there is some value in documenting that a function does not throw an exception, and it added the keyword `noexcept` for this purpose:

```
void f875(short, short) noexcept; // doesn't throw an exception
```

Scoped Enumerations

Traditional C++ enumerations provide a way to create named constants. However, they come with a rather low level of type checking. Also the scope for enumeration names is the scope that encloses the enumeration definition, which means that two enumerations

defined in the same scope must not have enumeration members with the same name. Finally, enumerations may not be completely portable because different implementations may choose to use different underlying types. C++11 introduces a variant of enumerations that addresses these problems. The new form is indicated by using `class` or `struct` in the definition:

```
enum Old1 {yes, no, maybe};                        // traditional form
enum class New1 {never, sometimes, often, always}; // new form
enum struct New2 {never, lever, sever};            // new form
```

The new forms avoid name conflicts by requiring explicit scoping. Thus, you would use `New1::never` and `New2::never` to identify those particular enumerations. Chapter 10, "Objects and Classes," provides more details.

Class Changes

C++11 makes several changes to simplify and expand class design. These include ways to allow constructors to call one another and to be inherited, better ways to control method access, and move constructors and assignment operators, all of which will be covered in this chapter. Meanwhile, let's review the changes that have been discussed previously.

`explicit` Conversion Operators

One of the exciting aspects in the early days of C++ was the ease with which automatic conversions for classes could be established. One of the realizations that grew as programming experience accumulated was that automatic type conversions could lead to problems in the form of unexpected conversions. C++ then addressed one aspect of the problem by introducing the keyword `explicit` to suppress automatic conversions invoked by one-argument constructors:

```
class Plebe
{
    Plebe(int);   // automatic int-to-plebe conversion
    explicit Plebe(double);  // requires explicit use
    ...
};
...
Plebe a, b;
a = 5;           // implicit conversion, call Plebe(5)
b = 0.5;         // not allowed
b = Plebe(0.5);  // explicit conversion
```

C++11 extends the use of `explicit` (discussed in Chapter 11, "Working with Classes") so that conversion functions can be treated similarly:

```
class Plebe
{
...
// conversion functions
```

```
    operator int() const;
    explicit operator double() const;
    ...
};
...
Plebe a, b;
int n = a;       // int-to-Plebe automatic conversion
double x = b;    // not allowed
x = double(b);   // explicit conversion, allowed
```

Member In-Class Initialization

Many first-time users of C++ have wondered why they can't initialize members simply by providing values in the class definition. Now they (and you) can. The syntax looks like this:

```
class Session
{
    int mem1 = 10;         // in-class initialization
    double mem2 {1966.54}; // in-class initialization
    short mem3;
public:
    Session(){}                                                    // #1
    Session(short s) : mem3(s) {}                                  // #2
    Session(int n, double d, short s) : mem1(n), mem2(d), mem3(s) {} // #3
...
};
```

You can use the equal sign or the brace forms of initialization, but not the parenthesized version. The result is the same as if you provided the first two constructors with member initialization list entries for `mem1` and `mem2`:

```
Session() : mem1(10), mem2(1966.54) {}
Session(short s) : mem1(10), mem2(1966.54), mem3(s) {}
```

Note how using in-class initialization avoids having to duplicate code in the constructors, thus reducing work and the number of opportunities for error and boredom for the programmer.

These default values are overridden by a constructor that supplies values in the member initialization list, so the third constructor overrides the in-class member initialization.

Template and STL Changes

C++11 makes several changes extending the usability of templates in general and the Standard Template Library in particular. Some are in the library itself; others relate to ease of use. In this chapter we've already mentioned template aliases and the STL-friendly smart pointers.

Range-based `for` Loop

The range-based `for` loop (discussed in Chapter 5, "Loops and Relational Expressions," and Chapter 16) simplifies writing loops for built-in arrays and for classes, such as `std::string` and the STL containers, that have `begin()` and `end()` methods identifying a range. The loop applies the indicated action to each element in the array or container:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
    std::cout << x << std::endl;
```

Here, `x` takes on the value of each element in `prices` in turn. The type of `x` should match the type of the array element. An easier and safer way of doing this is to use `auto` to declare `x`; the compiler will deduce the type from the information in the `prices` declaration:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (auto x : prices)
    std::cout << x << std::endl;
```

If your intent is to have the loop modify elements of the array or container, use a reference type:

```
std::vector<int> vi(6);
for (auto & x: vi)         // use a reference if loop alters contents
    x = std::rand();
```

New STL Containers

C++11 adds `forward_list`, `unordered_map`, `unordered_multimap`, `unordered_set`, and `unordered_multiset` to its collection of STL containers (see Chapter 16). The `forward_list` container is a singly linked list that can be traversed in just one direction; it's simpler and more economical of space than the doubly linked `list` container. The other four containers support implementing hash tables.

C++11 also adds the `array` template (discussed in Chapter 4, "Compound Types," and Chapter 16), for which one specifies an element type and a fixed number of elements:

```
std::array<int,360> ar;  // array of 360 ints
```

This template class does not satisfy all the usual template requirements. For example, because the size is fixed, you can't use any method, such as `put_back()`, that changes the size of a container. But `array` does have the `begin()` and `end()` methods, which allow you to use many of the range-based STL algorithms with array objects.

New STL Methods

C++11 adds `cbegin()` and `cend()` to the list of STL methods. Like `begin()` and `end()`, the new methods return iterators to the first element and to one past the last element of a container, thus specifying a range encompassing all the elements. In addition, the new

methods treat the elements as if they were `const`. Similarly, `crbegin()` and `crend()` are `const` versions of `rbegin()` and `rend()`.

More significantly, STL containers now have move constructors and move assignment operators in addition to the traditional copy constructors and regular assignment operators. This chapter describes move semantics later.

`valarray` Upgrade

The `valarray` template was developed independently of the STL, and its original design precluded using range-based STL algorithms with `valarray` objects. C++11 adds two functions, `begin()` and `end()`, that each take a `valarray` argument. They return iterators to the first and one past the last element of a `valarray` object, allowing one to use range-based STL algorithms (see Chapter 16).

`export` Departs

C++98 introduced the `export` keyword in the hopes of creating a way to separate template definitions into interface files containing the prototypes and template declarations and implementation files containing the template function and methods definitions. This proved to be impractical, and C++11 ends that roll for export. However, the Standard retains `export` as a keyword for possible future use.

Angle Brackets

To avoid confusion with the `>>` operator, C++ required a space between the brackets in nested template declarations:

```
std::vector<std::list<int> > vl; // >> not ok
```

C++11 removes that requirement:

```
std::vector<std::list<int>> vl; //  >> ok in C++11
```

The rvalue Reference

The traditional C++ reference, now called an *lvalue reference*, binds an identifier to an lvalue. An lvalue is an expression, such as a variable name or a dereferenced pointer, that represents data for which the program can obtain an address. Originally, an lvalue was one that could appear on the left side of an assignment statement, but the advent of the `const` modifier allowed for constructs that cannot be assigned to but which are still addressable:

```
int n;
int * pt = new int;
const int b = 101;  // can't assign to b, but &b is valid
int & rn = n;       // n identifies datum at address &n
int & rt = *pt;     // *pt identifies datum at address pt
const int & rb = b; // b identifies const datum at address &b
```

C++11 adds the rvalue reference (discussed in Chapter 8), indicated by using `&&`, that can bind to rvalues—that is, values that can appear on the right-hand side of an

assignment expression but for which one cannot apply the address operator. Examples include literal constants (aside from C-style strings, which evaluate as addresses), expressions such as `x+y`, and function return values, providing the function does not return a reference:

```
int x = 10;
int y = 23;
int && r1 = 13;
int && r2 = x + y;
double && r3 = std::sqrt(2.0);
```

Note that what `r2` really binds to is the value to which `x + y` evaluates at that time. That is, `r2` binds to the value `23`, and `r2` is unaffected by subsequent changes to `x` or `y`.

Interestingly, binding an rvalue to an rvalue reference results in the value being stored in a location whose address can be taken. That is, although you can't apply the `&` operator to `13`, you can apply it to `r1`. This binding of the data to particular addresses is what makes it possible to access the data through the rvalue references.

Listing 18.1 provides a short example illustrating some of these points about rvalue references.

Listing 18.1 **rvref.cpp**

```
// rvref.cpp -- simple uses of rvalue references
#include <iostream>

inline double f(double tf) {return 5.0*(tf-32.0)/9.0;};
int main()
{
    using namespace std;
    double tc = 21.5;
    double && rd1 = 7.07;
    double && rd2 = 1.8 * tc + 32.0;
    double && rd3 = f(rd2);
    cout << " tc value and address: " << tc <<", " << &tc << endl;
    cout << "rd1 value and address: " << rd1 <<", " << &rd1 << endl;
    cout << "rd2 value and address: " << rd2 <<", " << &rd2 << endl;
    cout << "rd3 value and address: " << rd3 <<", " << &rd3 << endl;
    cin.get();
    return 0;
}
```

Here is a sample output:

```
 tc value and address: 21.5, 002FF744
rd1 value and address: 7.07, 002FF728
rd2 value and address: 70.7, 002FF70C
rd3 value and address: 21.5, 002FF6F0
```

One of the main reasons for introducing the rvalue reference is to implement move semantics, the next topic in this chapter.

Move Semantics and the Rvalue Reference

Now let's move on to topics we haven't yet discussed. C++11 enables a technique called *move semantics*. This raises some questions, such as what are move semantics, how does C++11 enable them, and why do we need them? We'll begin with the last question.

The Need for Move Semantics

Let's look into the copying process as it worked prior to C++11. Suppose we start with something like this:

```
vector<string> vstr;
// build up a vector of 20,000 strings, each of 1000 characters
...
vector<string> vstr_copy1(vstr);  // make vstr_copy1 a copy of vstr
```

Both the `vector` class and the `string` class use dynamic memory allocation, so they will have copy constructors defined that use some version of `new`. To initialize the `vstr_copy1` object, the `vector<string>` copy constructor will use `new` to allocate memory for 20,000 `string` objects, and each `string` object, in turn, will call the `string` copy constructor, which will use `new` to allocate memory for 1000 characters. Then all 20,000,000 characters will be copied from the memory controlled by `vstr` to the memory controlled by `vstr_copy1`. That's a lot of work, but if it's got to be done, then it's got to be done.

But does it "got to be done?" There are times when the answer is no. For instance, suppose we have a function that returns a `vector<string>` object:

```
vector<string> allcaps(const vector<string> & vs)
{
    vector<string> temp;
// code that stores an all-uppercase version of vs in temp
    return temp;
}
```

Next, suppose we use it this way:

```
vector<string> vstr;
// build up a vector of 20,000 strings, each of 1000 characters
vector<string> vstr_copy1(vstr);              // #1
vector<string> vstr_copy2(allcaps(vstr));     // #2
```

Superficially, statements #1 and #2 are similar; each uses an existing object to initialize a new `vector<string>` object. If we take this code at face value, `allcaps()` creates a `temp` object managing 20,000,000 characters, the `vector` and `string` copy constructors go through the effort of creating a 20,000,000-character duplicate, then the program

deletes the temporary object returned by `allcaps()`. (A really uninspired compiler could even copy `temp` to a temporary return object, delete `temp`, and then delete the return object.) The main point is that a lot of effort is wasted here. Because the temporary object is deleted, wouldn't it be better if the compiler could just transfer ownership of the data to `vstr_copy2`? That is, instead of copying 20,000,000 characters to a new location and then deleting the old location, just leave the characters in place and attach the `vstr_copy2` label to them. This would be similar to what happens when you move a file from one directory to another: The actual file stays where it is on the hard drive, and just the bookkeeping is altered. Such an approach is called *move semantics*. Somewhat paradoxically, move semantics actually avoids moving the primary data; it just adjusts the bookkeeping.

To implement move semantics, we need a way to let the compiler know when it needs to do a real copy and when it doesn't. Here's where the rvalue reference comes into play. We can define two constructors. One, the regular copy constructor, can use the usual `const` lvalue reference as a parameter. This reference will bind to lvalue arguments, such as `vstr` in statement #1. The other, called a *move constructor*, can use an rvalue reference, and it can bind to rvalue arguments, such as the return value of `allcaps(vstr)` in statement #2. The copy constructor can do the usual deep copy, while the move constructor can just adjust the bookkeeping. A move constructor may alter its argument in the process of transferring ownership to a new object, and this implies that an rvalue reference parameter should not be `const`.

A Move Example

Let's look at an example to see how move semantics and rvalue references work. Listing 18.2 defines and uses the `Useless` class, which incorporates dynamic memory allocation, a regular copy constructor, and a move constructor, which uses move semantics and an rvalue reference. In order to illustrate the processes involved, the constructors and destructor are unusually verbose, and the class uses a state variable to keep track of the number of objects. Also some important methods, such as the assignment operator, are omitted. (Despite these omissions, the `Useless` class should not be confused with the eco-friendly `Use_Less` class.)

Listing 18.2 **useless.cpp**

```
// useless.cpp -- an otherwise useless class with move semantics
#include <iostream>
using namespace std;

// interface
class Useless
{
private:
    int n;          // number of elements
    char * pc;      // pointer to data
    static int ct;  // number of objects
```

```
    void ShowObject() const;
public:
    Useless();
    explicit Useless(int k);
    Useless(int k, char ch);
    Useless(const Useless & f); // regular copy constructor
    Useless(Useless && f);      // move constructor
    ~Useless();
    Useless operator+(const Useless & f)const;
// need operator=() in copy and move versions
    void ShowData() const;
};

// implementation
int Useless::ct = 0;

Useless::Useless()
{
    ++ct;
    n = 0;
    pc = nullptr;
    cout << "default constructor called; number of objects: " << ct << endl;
    ShowObject();
}

Useless::Useless(int k) : n(k)
{
    ++ct;
    cout << "int constructor called; number of objects: " << ct << endl;
    pc = new char[n];
    ShowObject();
}

Useless::Useless(int k, char ch) : n(k)
{
    ++ct;
    cout << "int, char constructor called; number of objects: " << ct
         << endl;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = ch;
    ShowObject();
}

Useless::Useless(const Useless & f): n(f.n)
{
```

```
    ++ct;
    cout << "copy const called; number of objects: " << ct << endl;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    ShowObject();
}

Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    cout << "move constructor called; number of objects: " << ct << endl;
    pc = f.pc;       // steal address
    f.pc = nullptr;  // give old object nothing in return
    f.n = 0;
    ShowObject();
}

Useless::~Useless()
{
    cout << "destructor called; objects left: " << --ct << endl;
    cout << "deleted object:\n";
    ShowObject();
    delete [] pc;
}

Useless Useless::operator+(const Useless & f)const
{
    cout << "Entering operator+()\n";
    Useless temp = Useless(n + f.n);
    for (int i = 0; i < n; i++)
        temp.pc[i] = pc[i];
    for (int i = n; i < temp.n; i++)
        temp.pc[i] = f.pc[i - n];
    cout << "temp object:\n";
    cout << "Leaving operator+()\n";
    return temp;
}

void Useless::ShowObject() const
{
    cout << "Number of elements: " << n;
    cout << " Data address: " << (void *) pc << endl;
}
```

```
void Useless::ShowData() const
{
    if (n == 0)
        cout << "(object empty)";
    else
        for (int i = 0; i < n; i++)
            cout << pc[i];
    cout << endl;
}

// application
int main()
{
    {
        Useless one(10, 'x');
        Useless two = one;           // calls copy constructor
        Useless three(20, 'o');
        Useless four (one + three);  // calls operator+(), move constructor
        cout << "object one: ";
        one.ShowData();
        cout << "object two: ";
        two.ShowData();
        cout << "object three: ";
        three.ShowData();
        cout << "object four: ";
        four.ShowData();
    }
}
```

The crucial definitions are those of the two copy/move constructors. First, shorn of the output statements, here is the copy constructor:

```
Useless::Useless(const Useless & f): n(f.n)
{
    ++ct;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
}
```

It does the usual deep copy, and it is the constructor that's used by the following statement:

```
Useless two = one;           // calls copy constructor
```

The reference `f` refers to the lvalue object `one`.

Next, again shorn of the output statements, here is the move constructor:

```
Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    pc = f.pc;       // steal address
    f.pc = nullptr;  // give old object nothing in return
    f.n = 0;
}
```

It takes ownership of the existing data by setting `pc` to point to the data. At this time, both `pc` and `f.pc` point to the same data. This would be awkward when the destructors are called, because a program shouldn't call `delete []` twice for the same address. To avoid this problem, the constructor then sets the original pointer to the null pointer because it is not an error to apply `delete []` to the null pointer. This appropriation of ownership often is termed *pilfering*. The code also sets the element count in the original object to 0. This isn't necessary, but it makes the output for our example look more self-consistent. Note that the changes to the `f` object require not using `const` in the parameter declaration.

It is this constructor that is used in the following statement:

```
Useless four (one + three);  // calls move constructor
```

The expression `one + three` invokes `Useless::operator+()`, and the rvalue reference `f` binds to the rvalue temporary object returned by the method.

Here's the output when the program was compiled with Microsoft Visual C++ 2010:

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 006F4B68
copy const called; number of objects: 2
Number of elements: 10 Data address: 006F4BB0
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 006F4BF8
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 006F4C48
temp object:
Leaving operator+()
move constructor called; number of objects: 5
Number of elements: 30 Data address: 006F4C48
destructor called; objects left: 4
deleted object:
Number of elements: 0 Data address: 00000000
object one: xxxxxxxxxx
object two: xxxxxxxxxx
object three: oooooooooooooooooo
object four: xxxxxxxxxxoooooooooooooooooooo
destructor called; objects left: 3
```

```
deleted object:
Number of elements: 30 Data address: 006F4C48
destructor called; objects left: 2
deleted object:
Number of elements: 20 Data address: 006F4BF8
destructor called; objects left: 1
deleted object:
Number of elements: 10 Data address: 006F4BB0
destructor called; objects left: 0
deleted object:
Number of elements: 10 Data address: 006F4B68
```

Note that object `two` is a separate copy of object `one`: Both display the same data output, but the data addresses (`006F4B68` and `006F4BB0`) are different. On the other hand, the data address (`006F4C48`) of the object created in the `Useless::operator+()` method is the same as the data address stored in the `four` object, which was constructed by the move copy constructor. Also note how the destructor was called for the temporary object after the `four` object was constructed. You can tell that is the temporary object that was deleted because the size and the data address both show as 0.

Compiling the same program (but replacing `nullptr` with `0`) with g++ 4.5.0 with the `-std=c++11` flag leads to an interestingly different output:

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 0xa50338
copy const called; number of objects: 2
Number of elements: 10 Data address: 0xa50348
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 0xa50358
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 0xa50370
temp object:
Leaving operator+()
object one: xxxxxxxxxx
object two: xxxxxxxxxx
object three: oooooooooooooooooo
object four: xxxxxxxxxxoooooooooooooooooo
destructor called; objects left: 3
deleted object:
Number of elements: 30 Data address: 0xa50370
destructor called; objects left: 2
deleted object:
Number of elements: 20 Data address: 0xa50358
destructor called; objects left: 1
deleted object:
Number of elements: 10 Data address: 0xa50348
```

```
destructor called; objects left: 0
deleted object:
Number of elements: 10 Data address: 0xa50338
```

Note that the move constructor is not called and that only four objects were created. The compiler did not call any of our constructors to construct the `four` object; instead, it deduced that the `four` object should be the beneficiary of the work done by `operator+()` and transferred the name `four` to the object created in `operator+()`. In general, compilers are empowered to make their own optimizations if the result is the same that would have been obtained by going through all the steps. Even if you omit the move constructor from the code and compile with g++, you get the same behavior.

Move Constructor Observations

Although using an rvalue reference enables move semantics, it doesn't magically make it happen. There are two steps to enablement. The first step is that the rvalue reference allows the compiler to identify when move semantics can be used:

```
Useless two = one;          // matches Useless::Useless(const Useless &)
Useless four (one + three); // matches Useless::Useless(Useless &&)
```

The object `one` is an lvalue, so it matches the lvalue reference, and the expression `one + three` is an rvalue, so it matches the rvalue reference. Thus, the rvalue reference directs initialization for object `four` to the move constructor. The second step in enabling move semantics is coding the move constructor so that it provides the behavior we want.

In short, the presence of one constructor with an lvalue reference and a second constructor with an rvalue reference sorts possible initializations into two groups. Objects initialized with an lvalue object use the copy constructor, and objects initialized with an rvalue object use the move constructor. The code writer then can endow these constructors with different behaviors.

This raises the question of what happened before rvalue references were part of the language. If there is no move constructor and the compiler doesn't optimize away the need for the copy constructor, what should happen? Under C++98, the following statement would invoke the copy constructor:

```
Useless four (one + three);
```

But an lvalue reference doesn't bind to an rvalue. So what happens? As you may recall from Chapter 8, a `const` reference parameter will bind to a temporary variable or object if the actual argument is an rvalue:

```
int twice(const & rx) {return 2 * rx;}
...
int main()
{
    int m = 6;
    // below, rx refers to m
    int n = twice(m);
```

```
    // below, rx refers to a temporary variable initialized to 21
    int k = twice(21);
...
```

So in the `Useless` case, the formal parameter `f` will be initialized to a temporary object that's initialized to the return value of `operator+()`. Here is an excerpt from the result of using an older compiler with Listing 18.2 and omitting the move constructor:

```
...
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 01C337C4
temp object:
Leaving operator+()
copy const called; number of objects: 5
Number of elements: 30 Data address: 01C337E8
destructor called; objects left: 4
deleted object:
Number of elements: 30 Data address: 01C337C4
copy const called; number of objects: 5
Number of elements: 30 Data address: 01C337C4
destructor called; objects left: 4
deleted object:
Number of elements: 30 Data address: 01C337E8
...
```

First, within the `Useless::operator+()` method, a constructor creates `temp`, allocating storage for 30 elements at location `01C337C4`. Then the copy constructor creates a temporary copy to which `f` will refer, copying the information to location `01C337E8`. Next, `temp`, which uses location `01C337C4`, gets deleted. Then a new object, `four`, is constructed, reusing the recently freed memory at `01C337C4`. Then the temporary argument object, which used location `01C337E8`, gets deleted. So three complete objects were constructed, and two of them were destroyed. This is the sort of extra work that move semantics are meant to eliminate.

As the g++ example shows, an optimizing compiler might eliminate extra copying on its own, but using an rvalue reference lets the programmer dictate move semantics when appropriate.

Assignment

The same considerations that make move semantics appropriate for constructors make them appropriate for assignment. Here, for example, is how you could code the copy assignment and the move assignment operators for the `Useless` class:

```
Useless & Useless::operator=(const Useless & f)  // copy assignment
{
    if (this == &f)
```

```
        return *this;
    delete [] pc;
    n = f.n;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    return *this;
}

Useless & Useless::operator=(Useless && f)       // move assignment
{
    if (this == &f)
        return *this;
    delete [] pc;
    n = f.n;
    pc = f.pc;
    f.n = 0;
    f.pc = nullptr;
    return *this;
}
```

The copy assignment operator follows the usual pattern given in Chapter 12. The move assignment operator deletes the original data in the target and pilfers the source object. It's important that only one pointer points to the data, so the method resets the pointer in the source object to the null pointer.

As with the move constructor, the move assignment operator parameter is not a `const` reference because the method alters the source object.

Forcing a Move

Move constructors and move assignment operators work with rvalues. What if you want to use them with lvalues? For instance, a program could analyze an array of some sort of candidate objects, select one object for further use, and discard the array. It would be convenient if you could use a move constructor or a move assignment operator to preserve the selected object. However, suppose you tried the following:

```
Useless choices[10];
Useless best;
int pick;
... // select one object,  set pick to index
best = choices[pick];
```

The `choices[pick]` object is an lvalue, so the assignment statement will use the copy assignment operator, not the move assignment operator. But if you could make `choices[pick]` look like an rvalue, then the move assignment operator would be used. This can be done by using the `static_cast<>` operator to cast the object to type `Useless &&`. C++11 provides a simpler way to do this—use the `std::move()` function,

which is declared in the `utility` header file. Listing 18.3 illustrates this technique. It adds verbose versions of the assignment operators to the `Useless` class while silencing the previously verbose constructors and destructor.

Listing 18.3 **stdmove.cpp**

```
// stdmove.cpp -- using std::move()
#include <iostream>
#include <utility>

// interface
class Useless
{
private:
    int n;          // number of elements
    char * pc;      // pointer to data
    static int ct;  // number of objects
    void ShowObject() const;
public:
    Useless();
    explicit Useless(int k);
    Useless(int k, char ch);
    Useless(const Useless & f); // regular copy constructor
    Useless(Useless && f);      // move constructor
    ~Useless();
    Useless operator+(const Useless & f)const;
    Useless & operator=(const Useless & f); // copy assignment
    Useless & operator=(Useless && f);      // move assignment
    void ShowData() const;
};

// implementation
int Useless::ct = 0;

Useless::Useless()
{
    ++ct;
    n = 0;
    pc = nullptr;
 }

Useless::Useless(int k) : n(k)
{
    ++ct;
    pc = new char[n];
}
```

```
Useless::Useless(int k, char ch) : n(k)
{
    ++ct;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = ch;
}

Useless::Useless(const Useless & f): n(f.n)
{
    ++ct;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
}

Useless::Useless(Useless && f): n(f.n)
{
    ++ct;
    pc = f.pc;       // steal address
    f.pc = nullptr;  // give old object nothing in return
    f.n = 0;
}

Useless::~Useless()
{
    delete [] pc;
}

Useless & Useless::operator=(const Useless & f)  // copy assignment
{
    std::cout << "copy assignment operator called:\n";
    if (this == &f)
        return *this;
    delete [] pc;
    n = f.n;
    pc = new char[n];
    for (int i = 0; i < n; i++)
        pc[i] = f.pc[i];
    return *this;
}

Useless & Useless::operator=(Useless && f)       // move assignment
{
    std::cout << "move assignment operator called:\n";
    if (this == &f)
```

```
        return *this;
    delete [] pc;
    n = f.n;
    pc = f.pc;
    f.n = 0;
    f.pc = nullptr;
    return *this;
}

Useless Useless::operator+(const Useless & f)const
{
    Useless temp = Useless(n + f.n);
    for (int i = 0; i < n; i++)
        temp.pc[i] = pc[i];
    for (int i = n; i < temp.n; i++)
        temp.pc[i] = f.pc[i - n];
    return temp;
}

void Useless::ShowObject() const
{
    std::cout << "Number of elements: " << n;
    std::cout << " Data address: " << (void *) pc << std::endl;
}

void Useless::ShowData() const
{
    if (n == 0)
        std::cout << "(object empty)";
    else
        for (int i = 0; i < n; i++)
            std::cout << pc[i];
    std::cout << std::endl;
}

// application
int main()
{
    using std::cout;
    {
        Useless one(10, 'x');
        Useless two = one +one;   // calls move constructor
        cout << "object one: ";
        one.ShowData();
        cout << "object two: ";
        two.ShowData();
```

```
        Useless three, four;
        cout << "three = one\n";
        three = one;              // automatic copy assignment
        cout << "now object three = ";
        three.ShowData();
        cout << "and object one = ";
        one.ShowData();
        cout << "four = one + two\n";
        four = one + two;         // automatic move assignment
        cout << "now object four = ";
        four.ShowData();
        cout << "four = move(one)\n";
        four = std::move(one);    // forced move assignment
        cout << "now object four = ";
        four.ShowData();
        cout << "and object one = ";
        one.ShowData();
    }
}
```

Here is a sample run:

```
object one: xxxxxxxxxx
object two: xxxxxxxxxxxxxxxxxxxx
three = one
copy assignment operator called:
now object three = xxxxxxxxxx
and object one = xxxxxxxxxx
four = one + two
move assignment operator called:
now object four = xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
four = move(one)
move assignment operator called:
now object four = xxxxxxxxxx
and object one = (object empty)
```

As you can see, assigning `one` to `three` invokes copy assignment, but assigning `move(one)` to `four` invokes move assignment.

You should realize that the `std::move()` function doesn't necessarily produce a move operation. Suppose, for instance, that `Chunk` is a class with private data and that we have the following code:

```
Chunk one;
...
Chunk two;
two = std::move(one);  // move semantics?
```

The expression `std::move(one)` is an rvalue, so the assignment statement will invoke the move assignment operator for `Chunk`, providing that one has been defined. But if the `Chunk` class doesn't define a move assignment operator, the compiler will use the copy assignment operator. And if that also isn't defined, then assignment isn't allowed at all.

The main benefit rvalue references bring to most programmers is not the opportunity to write code using them. Rather, it is the opportunity to use library code that utilizes rvalue references to implement move semantics. For example, the STL classes now have copy constructors, move constructors, copy assignment operators, and move assignment operators.

New Class Features

C++11 adds several features to classes in addition to those already mentioned in this chapter—that is, explicit conversion operators and in-class member initialization.

Special Member Functions

C++11 adds two more *special member functions* (the move constructor and the move assignment operator) to four previous ones (the default constructor, the copy constructor, the copy assignment operator, and the destructor). These are member functions that the compiler provides automatically, subject to a variety of conditions.

The default constructor, recall, is a constructor that can be called with no arguments. The compiler provides one if you fail to define any constructors for the class. This default version of a default constructor is termed the *defaulted* default constructor. The defaulted default constructor leaves members of the built-in types uninitialized, and it invokes the default constructors for members that are class objects.

Also the compiler provides a defaulted copy constructor if you don't provide one and if your code requires its use, and it now provides a defaulted move constructor if you don't provide one and if your code requires its use. If the class name is `Someclass`, these two defaulted constructors have the following prototypes:

```
Someclass::Someclass(const Someclass &);  // defaulted copy constructor
Someclass::Someclass(Someclass &&);       // defaulted move constructor
```

In similar circumstances, the compiler provides a defaulted copy assignment operator and a defaulted move assignment operator with the following prototypes:

```
Someclass & Someclass::operator(const Someclass &);  // defaulted copy assignment
Someclass & Someclass::operator(Someclass &&);       // defaulted move assignment
```

Finally, the compiler provides a destructor if you don't.

There are various exceptions to this description. If you do provide a destructor or a copy constructor or a copy assignment operator, the compiler does not automatically provide a move constructor or a move assignment operator. If you do provide a move constructor or a move assignment operator, the compiler does not automatically provide a copy constructor or a copy assignment operator.

Also the defaulted move constructor and defaulted move assignment operator work similarly to their copy counterparts, doing memberwise initialization and copying for built-in types. For members that are class objects, constructors and assignment operators for those classes are used as if the parameters were rvalues. This, in turn, invokes move constructors and assignment operators, if defined, and copy constructors and assignment operators otherwise, if defined.

Defaulted and Deleted Methods

C++11 provides more control over which methods are used. Suppose that you wish to use a defaulted function that, due to circumstances, isn't created automatically. For example, if you provide a move constructor, then the default constructor, the copy constructor, and the copy assignment operator are not provided. In that case, you can use the keyword `default` to explicitly declare the defaulted versions of these methods:

```
class Someclass
{
public:
   Someclass(Someclass &&);
   Someclass() = default;      // use compiler-generated default constructor
   Someclass(const Someclass &) = default;
   Someclass & operator=(const Someclass &) = default;
...
};
```

The compiler provides the same constructor that it would have provided automatically had you not provided the move constructor.

The `delete` keyword, on the other hand, can be used to prevent the compiler from using a particular method. For example, to prevent an object from being copied, you can disable the copy constructor and copy assignment operator:

```
class Someclass
{
public:
   Someclass() = default;      // use compiler-generated default constructor
// disable copy constructor and copy assignment operator:
   Someclass(const Someclass &) = delete;
   Someclass & operator=(const Someclass &) = delete;
// use compiler-generated move constructor and move assignment operator:
   Someclass(Someclass &&) = default;
   Someclass & operator=(Someclass &&) = default;
   Someclass & operator+(const Someclass &) const;
...
};
```

You may recall (from Chapter 12) that you can disable copying by placing the copy constructor and assignment operator in the `private` section of a class. But using `delete` is a less devious and more easily understood way to accomplish that end.

What is the effect of disabling copy methods while enabling move methods? Recall that an rvalue reference, such as used by move operations, binds only to rvalue expressions. This implies the following:

```
Someclass one;
Someclass two;
Someclass three(one);       // not allowed, one an lvalue
Someclass four(one + two);  // allowed, expression is an rvalue
```

Only the six special member functions can be defaulted, but you can use `delete` with any member function. One possible use is to disable certain conversions. Suppose, for example, that the `Someclass` class has a method with a type `double` parameter:

```
class Someclass
{
public:
...
    void redo(double);
...
};
```

Then suppose we have the following code:

```
Someclass sc;
sc.redo(5);
```

The `int` value `5` will be promoted to `5.0`, and the `redo()` method will execute.

Now suppose the `Someclass` definition is modified thusly:

```
class Someclass
{
public:
...
    void redo(double);
    void redo(int) = delete;
...
};
```

In this case, the method call `sc.redo(5)` matches the `redo(int)` prototype. The compiler will detect that fact and also detect that `redo(int)` is deleted, and it will then flag the call as a compile-time error. This illustrates an important fact about deleted functions. They do exist as far as function look-up is concerned, but to use them is an error.

Delegating Constructors

If you provide a class with several constructors, you may find yourself writing the same code over and over. That is, some of the constructors may require you to duplicate code

already present in other constructors. To make coding simpler and more reliable, C++11 allows to you use a constructor as part of the definition of another constructor. This process is termed *delegation* because one constructor temporarily delegates responsibility to another constructor to work on the object it is constructing. Delegation uses a variant of the member initialization list syntax:

```
class Notes {
    int k;
    double x;
    std::string st;
public:
    Notes();
    Notes(int);
    Notes(int, double);
    Notes(int, double, std::string);
};
Notes::Notes(int kk, double xx, std::string stt) : k(kk),
             x(xx), st(stt) {/*do stuff*/}
Notes::Notes() : Notes(0, 0.01, "Oh") {/* do other stuff*/}
Notes::Notes(int kk) : Notes(kk, 0.01, "Ah") {/* do yet other stuff*/ }
Notes::Notes( int kk, double xx ) : Notes(kk, xx, "Uh") {/* ditto*/ }
```

The default constructor, for example, uses the first constructor in the list to initialize the data members and to also do whatever the body of that constructor requests. Then it finishes up doing whatever its own body requests.

Inheriting Constructors

In another move to simplify coding, C++11 provides a mechanism for derived classes to inherit constructors from the base class. C++98 already had a syntax for making functions from a namespace available:

```
namespace Box
{
    int fn(int) { ... }
    int fn(double) { ... }
    int fn(const char *) { ... }
}
using Box::fn;
```

This makes all the overloaded `fn` functions available. The same technique works for making nonspecial member functions of a base class available to a derived class. For example, consider the following code:

```
class C1
{
...
public:
...
```

```
    int fn(int j) { ... }
    double fn(double w) { ... }
    void fn(const char * s) { ... }
};
class C2 : public C1
{
...
public:
...
    using C1::fn;
    double fn(double) { ... };
};
...
C2 c2;
int k = c2.fn(3);       // uses C1::fn(int)
double z = c2.fn(2.4);  // uses C2::fn(double)
```

The `using` declaration in `C2` makes the three `fn()` methods in `C1` available to a `C2` object. However, the `fn(double)` method defined in `C2` is chosen over the one from `C1`.

C++11 brings the same technique to constructors. All the constructors of the base class, other than the default, copy, and move constructors, are brought in as possible constructors for the derived class, but the ones that have function signatures matching derived class constructors aren't used:

```
class BS
{
    int q;
    double w;
public:
    BS() : q(0), w(0) {}
    BS(int k) : q(k), w(100) {}
    BS(double x) : q(-1), w(x) {}
    B0(int k, double x) : q(k), w(x) {}
    void Show() const {std::cout << q <<", " << w << '\n';}
};

class DR : public BS
{
    short j;
public:
    using BS::BS;
    DR() : j(-100) {}      // DR needs its own default constructor
    DR(double x) : BS(2*x), j(int(x)) {}
    DR(int i) : j(-2), BS(i, 0.5* i) {}
    void Show() const {std::cout << j << ", "; BS::Show();}
};
int main()
```

```
{
    DR o1;          // use DR()
    DR o2(18.81);   // use DR(double) instead of BS(double)
    DR o3(10, 1.8); // use BS(int, double)
    ...
}
```

Because there is no `DR(int, double)` constructor, the inherited `BS(int, double)` is used for `03`. Note that an inherited base-class constructor only initializes base-class members. If you need to initialize derived class members too, you can use the member list initialization syntax instead of inheritance:

```
DR(int i, int k, double x) : j(i), BS(k,x) {}
```

Managing Virtual Methods: `override` and `final`

Virtual methods are an important component of implementing polymorphic class hierarchies, in which a base class reference or pointer can invoke the particular method appropriate to the type of object referred to. Virtual methods do pose some programming pitfalls. For instance, suppose the base class declares a particular virtual method, and you decide to provide a different version for a derived class. This is called *overriding* the old version. But, as discussed in Chapter 13, "Class Inheritance," if you mismatch the function signature, you hide rather than override the old version:

```
class Action
{
    int a;
public:
    Action(int i = 0) : a(i) {}
    int val() const {return a;};
    virtual void f(char ch) const { std::cout << val() << ch << "\n";}
};
class Bingo : public Action
{
public:
    Bingo(int i = 0) : Action(i) {}
    virtual void f(char * ch) const { std::cout << val() << ch  << "!\n"; }
};
```

Because class `Bingo` uses `f(char * ch)` instead of `f(char ch)`, `f(char ch)` is hidden to a `Bingo` object. This prevents a program from using code like the following:

```
Bingo b(10);
b.f('@');  // works for Action object, fails for Bingo object
```

With C++11, you can use the virtual specifier `override` to indicate that you intend to override a virtual function. Place it after the parameter list. If your declaration does not match a base method, the compiler objects. Thus, the following version of `Bingo::f()` would generate a compile-time error message:

```
virtual void f(char * ch) const override { std::cout << val()
                                           << ch  << "!\n"; }
```

For example, Microsoft Visual C++ 2010 has this to say:

```
method with override specifier 'override' did not override any
base class methods
```

The specifier `final` addresses a different issue. You may find that you want to prohibit derived classes from overriding a particular virtual method. To do so, place `final` after the parameter list. For example, the following code would prevent classes based on `Action` to redefine the `f()` function:

```
virtual void f(char ch) const final { std::cout << val() << ch << "\n";}
```

The specifiers `override` and `final` do not quite have the status of keywords. Instead, they are labeled "identifiers with special meaning." This means that the compiler uses the context in which they appear to decide if they have a special meaning. In other contexts, they can be used as ordinary identifiers (for example, as variable names or enumerations).

Lambda Functions

When you see the term *lambda functions* (a.k.a. *lambda expressions* or, simply, *lambdas*), you may suspect that this is not one of the C++11 additions intended to help the novice programmer. You will have your suspicions seemingly confirmed when you see how lambda functions actually look—here's an example:

```
[&count](int x){count += (x % 13 == 0);}
```

But they aren't as arcane as they may look, and they do provide a useful service, particularly with STL algorithms using function predicates.

The How of Function Pointers, Functors, and Lambdas

Let's look at an example using three approaches for passing information to an STL algorithm: function pointers, functors, and lambdas. (For convenience, we'll refer to these three forms as *function objects* so that we won't have to keep repeating "function pointer or functor or lambda.") Suppose you wish to generate a list of random integers and determine how many of them are divisible by 3 and how many are divisible by 13. If necessary, imagine that this is a quest you find absolutely fascinating.

Generating the list is pretty straightforward. One option is to use a `vector<int>` array to hold the numbers and use the STL `generate()` algorithm to stock the array with random numbers:

```
#include <vector>
#include <algorithm>
#include <cmath>
...
std::vector<int> numbers(1000);
std::generate(vector.begin(), vector.end(), std::rand);
```

The `generate()` function takes a range, specified by the first two arguments, and sets each element to the value returned by the third argument, which is a function object that takes no arguments. In this case, the function object is a pointer to the standard `rand()` function.

With the help of the `count_if()` algorithm, it's easy to count the number of elements divisible by 3. The first two arguments should specify the range, just as for `generate()`. The third argument should be a function object that returns `true` or `false`. The `count_if()` function then counts all the elements for which the function object returns `true`. To find elements divisible by 3, you can use this function definition:

```
bool f3(int x) {return x % 3 == 0;}
```

Similarly, you can use the following function definition for finding elements divisible by 13:

```
bool f13(int x) {return x % 13 == 0;}
```

With these definitions in place, you can count elements as follows:

```
int count3 = std::count_if(numbers.begin(), numbers.end(), f3);
cout << "Count of numbers divisible by 3: " << count3 << '\n';
int count13 = std::count_if(numbers.begin(), numbers.end(), f13);
cout << "Count of numbers divisible by 13: " << count13 << "\n\n";
```

Next, let's review how to accomplish the same task using a functor. A functor, as you'll recall from Chapter 16, is a class object than can be used as if it were a function name, thanks to the class defining `operator()()` as a class method. One advantage of the functor in our example is that you can use the same functor for both counting tasks. Here's one possible definition:

```
class f_mod
{
private:
    int dv;
public:
    f_mod(int d = 1) : dv(d) {}
    bool operator()(int x) {return x % dv == 0;}
};
```

Recall how this works. You can use the constructor to create an `f_mod` object storing a particular integer value:

```
f_mod obj(3);  // f_mod.dv set to 3
```

This object can use the `operator()` method to return a `bool` value:

```
bool is_div_by_3 = obj(7); // same as obj.operator()(7)
```

The constructor itself can be used as an argument to functions such as `count_if()`:

```
count3 = std::count_if(numbers.begin(), numbers.end(), f_mod(3));
```

The argument `f_mod(3)` creates an object storing the value `3`, and `count_if()` uses the created object to call the `operator()()` method, setting the parameter `x` equal to an element of `numbers`. To count how many numbers are divisible by `13` instead of `3`, use `f_mod(13)` as the third argument.

Finally, let's examine the lambda approach. The name comes from *lambda calculus*, a mathematical system for defining and applying functions. The system enables one to use anonymous functions—that is, it allows one to dispense with function names. In the C++11 context, you can use an anonymous function definition (a lambda) as an argument to functions expecting a function pointer or functor. The lambda corresponding to the `f3()` function is this:

```
[](int x) {return x % 3 == 0;}
```

It looks much like the definition of `f3()`:

```
bool f3(int x) {return x % 3 == 0;}
```

The two differences are that the function name is replaced with `[]` (how anonymous is that!) and that there is no declared return type. Instead, the return type is the type that `decltype` would deduce from the return value, which would be `bool` in this case. If the lambda doesn't have a return statement, the type is deduced to be `void`. In our example, you would use this lambda as follows:

```
count3 = std::count_if(numbers.begin(), numbers.end(),
         [](int x){return x % 3 == 0;});
```

That is, you use the entire lambda expression as you would use a pointer or a functor constructor.

The automatic type deduction for lambdas works only if the body consists of a single return statement. Otherwise, you need to use the new trailing-return-value syntax:

```
[](double x)->double{int y = x; return x - y;}  // return type is double
```

Listing 18.4 illustrates the points just discussed.

Listing 18.4 **lambda0.cpp**

```
// lambda0.cpp -- using lambda expressions
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <ctime>
const long Size1 = 39L;
const long Size2 = 100*Size1;
const long Size3 = 100*Size2;
```

```
bool f3(int x) {return x % 3 == 0;}
bool f13(int x) {return x % 13 == 0;}

int main()
{
    using std::cout;
    std::vector<int> numbers(Size1);

    std::srand(std::time(0));
    std::generate(numbers.begin(), numbers.end(), std::rand);

// using function pointers
    cout << "Sample size = " << Size1 << '\n';

    int count3 = std::count_if(numbers.begin(), numbers.end(), f3);
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    int count13 = std::count_if(numbers.begin(), numbers.end(), f13);
    cout << "Count of numbers divisible by 13: " << count13 << "\n\n";

// increase number of numbers
    numbers.resize(Size2);
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size2 << '\n';
// using a functor
    class f_mod
    {
    private:
        int dv;
    public:
        f_mod(int d = 1) : dv(d) {}
        bool operator()(int x) {return x % dv == 0;}
    };

    count3 = std::count_if(numbers.begin(), numbers.end(), f_mod(3));
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    count13 = std::count_if(numbers.begin(), numbers.end(), f_mod(13));
    cout << "Count of numbers divisible by 13: " << count13 << "\n\n";

// increase number of numbers again
    numbers.resize(Size3);
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size3 << '\n';
// using lambdas
    count3 = std::count_if(numbers.begin(), numbers.end(),
             [](int x){return x % 3 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
```

```
    count13 = std::count_if(numbers.begin(), numbers.end(),
              [](int x){return x % 13 == 0;});
    cout << "Count of numbers divisible by 13: " << count13 << '\n';

    return 0;
}
```

Here is a sample output:

```
Sample size = 39
Count of numbers divisible by 3: 15
Count of numbers divisible by 13: 6

Sample size = 3900
Count of numbers divisible by 3: 1305
Count of numbers divisible by 13: 302

Sample size = 390000
Count of numbers divisible by 3: 130241
   Count of numbers divisible by 13: 29860
```

The output illustrates that one should not rely on statistics based on small samples.

The Why of Lambdas

You may be wondering what need, other than the flowering of geekly expressionism, the lambda serves. Let's examine this question in terms of four qualities: proximity, brevity, efficiency, and capability.

Many programmers feel that it is useful to locate definitions close to where they are used. That way, you don't have to scan through pages of source code to find, say, what the third argument to a `count_if()` function call accomplishes. Also if you need to modify the code, all the components are close at hand. And if you cut and paste the code for use elsewhere, again all the components are at hand. From this standpoint, lambdas are ideal because the definition is at the point of usage. Functions are worst because functions cannot be defined inside other functions, so the definition will be located possibly quite far from the point of usage. Functors can be pretty good because a class, including a functor class, can be defined inside a function, so the definition can be located close to the point of use.

In terms of brevity, the functor code is more verbose than the equivalent function or lambda code. Functions and lambdas are approximately equally brief. One apparent exception would be if you had to use a lambda twice:

```
count1 = std::count_if(n1.begin(), n1.end(),
         [](int x){return x % 3 == 0;});
count2 = std::count_if(n2.begin(), n2.end(),
         [](int x){return x % 3 == 0;});
```

But you don't actually have to write out the lambda twice. Essentially, you can create a name for the anonymous lambda and then use the name twice:

```
auto mod3 = [](int x){return x % 3 == 0;}  // mod3 a name for the lambda
count1 = std::count_if(n1.begin(), n1.end(), mod3);
count2 = std::count_if(n2.begin(), n2.end(), mod3);
```

You even can use this no-longer-anonymous lambda as an ordinary function:

```
bool result = mod3(z); // result is true if z % 3 == 0
```

Unlike an ordinary function, however, a named lambda can be defined inside a function. The actual type for `mod3` will be some implementation-dependent type that the compiler uses to keep track of lambdas.

The relative efficiencies of the three approaches boils down to what the compiler chooses to inline. Here, the function pointer approach is handicapped by the fact that compilers traditionally don't inline a function that has its address taken because the concept of a function address implies a non-inline function. With functors and lambdas, there is no apparent contradiction with inlining.

Finally, lambdas have some additional capabilities. In particular, a lambda can access by name any automatic variable in scope. Variables to be used are *captured* by having their names listed within the brackets. If just the name is used, as in `[z]`, the variable is accessed by value. If the name is preceded by an `&`, as in `[&count]`, the variable is accessed by reference. Using `[&]` provides access to all the automatic variables by reference, and `[=]` provides access to all the automatic variables by value. You also can mix and match. For instance, `[ted, &ed]` would provide access to `ted` by value and `ed` by reference, `[&, ted]` would provide access to `ted` by value and to all other automatic variables by reference, and `[=, &ed]` would provide access by reference to `ed` and by value to the remaining automatic variables. In Listing 18.4, you can replace

```
int count13;
...
count13 = std::count_if(numbers.begin(), numbers.end(),
          [](int x){return x % 13 == 0;});
```

with this:

```
int count13 = 0;
std::for_each(numbers.begin(), numbers.end(),
     [&count13](int x){count13 += x % 13 == 0;});
```

The `[&count13]` allows the lambda to use `count13` in its code. Because `count13` is captured by reference, any changes to `count13` in the lambda are changes to the original `count13`. The expression `x % 13 == 0` evaluates to `true` if `x` is divisible by `13`, and `true` converts to `1` when added to `count13`. Similarly, `false` converts to `0`. Thus, after `for_each()` applies the lambda expression to each element of `numbers`, `count13` counts the number of elements divisible by 13.

You can use this technique to count elements divisible by 3 and elements divisible by 13 using a single lambda expression:

```
int count3 = 0;
int count13 = 0;
std::for_each(numbers.begin(), numbers.end(),
     [&](int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
```

This time, `[&]` makes all the automatic variables, including `count3` and `count13`, available to the lambda expression.

Listing 18.5 puts these techniques to use.

Listing 18.5 **lambda1.cpp**

```
// lambda1.cpp -- use captured variables
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <ctime>
const long Size = 390000L;

int main()
{
    using std::cout;
    std::vector<int> numbers(Size);

    std::srand(std::time(0));
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size << '\n';
// using lambdas
    int count3 = std::count_if(numbers.begin(), numbers.end(),
          [](int x){return x % 3 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    int count13 = 0;
    std::for_each(numbers.begin(), numbers.end(),
         [&count13](int x){count13 += x % 13 == 0;});
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
// using a single lambda
    count3 = count13 = 0;
    std::for_each(numbers.begin(), numbers.end(),
         [&](int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
    return 0;
}
```

Here is a sample output:

```
Sample size = 390000
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
```

It's reassuring that both approaches (two separate lambdas and a single lambda) in this program led to the same answers.

The main motivation for adding lambdas to C++ was to enable using a function-like expression as an argument to a function expecting a function pointer or functor as an argument. So the typical lambda would be a test expression or comparison expression that could be written as a single return statement. That keeps the lambda short and easy to understand and enables automatic deduction of the return value. However, it is likely that a subset of the ingenious C++ programming community will develop other uses.

Wrappers

C++ provides several *wrappers* or *adapters*. These are objects used to provide a more uniform or more appropriate interface for other programming elements. For example, Chapter 16 described `bind1st` and `bind2nd`, which adapt functions with two parameters to match up with STL algorithms that expect functions with one parameter to be supplied as an argument. C++11 provides additional wrappers. They include the `bind` template, which provides a more flexible alternative to `bind1st` and `bind2nd`, the `mem_fn` template, which allows a member function to pass as a regular function, the `reference_wrapper` template allows you to create an object that acts like reference but which can be copied, and the `function` wrapper, which provides a way to handle several function-like forms uniformly.

Let's look more closely at one example of wrapper, the `function` wrapper, and at the problem it addresses.

The `function` Wrapper and Template Inefficiencies

Consider the following line of code:

```
answer = ef(q);
```

What is `ef`? It could be the name of a function. It could be a pointer to a function. It could be a function object. It could be a name assigned to a lambda expression. These all are examples of *callable types*. The abundance of callable types can lead to template inefficiencies. To see this, let's examine a simple case.

First, let's define some templates in a header file, as shown in Listing 18.6.

Listing 18.6 **somedefs.h**

```
// somedefs.h
#include <iostream>

template <typename T, typename F>
T use_f(T v, F f)
{
    static int count = 0;
    count++;
    std::cout << "  use_f count = " << count
              << ", &count = " << &count << std::endl;
    return f(v);
}

class Fp
{
private:
    double z_;
public:
    Fp(double z = 1.0) : z_(z) {}
    double operator()(double p) { return z_*p; }
};

class Fq
{
private:
    double z_;
public:
    Fq(double z = 1.0) : z_(z) {}
    double operator()(double q) { return z_+ q; }
};
```

The `use_f()` template uses the parameter f to represent a callable type:

```
return f(v);
```

Next the program in Listing 18.7 calls the `use_f()` template function six times.

Listing 18.7 **callable.cpp**

```
// callable.cpp -- callable types and templates
#include "somedefs.h"
#include <iostream>

double dub(double x) {return 2.0*x;}
double square(double x) {return x*x;}
```

```
int main()
{
    using std::cout;
    using std::endl;

    double y = 1.21;
    cout << "Function pointer dub:\n";
    cout << "  " << use_f(y, dub) << endl;
    cout << "Function pointer square:\n";
    cout << "  " << use_f(y, square) << endl;
    cout << "Function object Fp:\n";
    cout << "  " << use_f(y, Fp(5.0)) << endl;
    cout << "Function object Fq:\n";
    cout << "  " << use_f(y, Fq(5.0)) << endl;
    cout << "Lambda expression 1:\n";
    cout << "  " << use_f(y, [](double u) {return u*u;}) << endl;
    cout << "Lambda expression 2:\n";
    cout << "  " << use_f(y, [](double u) {return u+u/2.0;}) << endl;
    return 0;
}
```

The template parameter `T` is set to type `double` for each call. What about template parameter `F`? Each time the actual argument is something that takes a type `double` argument and returns a type `double` value, so it might seem that `F` would be the same type for all six calls to `use_f()` and that the template would be instantiated just once. But as the following sample output shows, that belief is naïve:

```
Function pointer dub:
  use_f count = 1, &count = 0x402028
  2.42
Function pointer square:
  use_f count = 2, &count = 0x402028
  1.1
Function object Fp:
  use_f count = 1, &count = 0x402020
  6.05
Function object Fq:
  use_f count = 1, &count = 0x402024
  6.21
Lambda expression 1:
  use_f count = 1, &count = 0x405020
  1.4641
Lambda expression 2:
  use_f count = 1, &count = 0x40501c
  1.815
```

The template function `use_f()` has a static member `count`, and we can use its address to see how many instantiations are made. There are five distinct addresses, so there must have been five distinct instantiations of the `use_f()` template.

To see what is happening, consider how the compiler determines the type for the `F` template parameter. First, look at this call:

```
use_f(y, dub);
```

Here `dub` is the name of a function that takes a `double` argument and returns a `double` value. The name of a function is a pointer, hence the parameter `F` becomes type `double (*)(double)`, a pointer to a function with a `double` argument and a `double` return value.

The next call is this:

```
use_f(y, square);
```

Again, the second argument is type `double (*)(double)`, so this call uses the same instantiation of `use_f()` as the first call.

The next two calls to `use_f()` have objects as second arguments, so `F` becomes type `Fp` and `Fq` respectively, so we get two new instantiations for these values of `F`. Finally, the last two calls set `F` to whatever types the compiler uses for lambda expressions.

Fixing the Problem

The function wrapper lets you rewrite the program so that it uses just one instantiation of `use_f()` instead of five. Note that the function pointers, function objects, and lambda expressions in Listing 18.7 share a common behavior—each takes one type `double` argument and each returns a type `double` value. We can say that each has the same *call signature*, which is described by the return type followed by a comma-separated list of parameter types enclosed in a pair of parentheses. Thus, these six examples all have `double(double)` as the call signature.

The `function` template, declared in the `functional` header file, specifies an object in terms of a call signature, and it can be used to wrap a function pointer, function object, or lambda expression having the same call signature. For example, the following declaration creates a `function` object `fdci` that takes a `char` and an `int` argument and returns type `double`:

```
std::function<double(char, int)> fdci;
```

You can then assign to `fdci` any function pointer, function object, or lambda expression that takes type `char` and `int` arguments and returns type `double`.

The various callable arguments in Listing 18.7 all have the same call signature – `double(double)`. So to fix Listing 18.7 and reduce the number of instantiations, we can use `function<double(double)>` to create six wrappers for the six functions, functors, and lambdas. Then all six calls to `use_f()` can be made with the same type (`function<double(double)>`) for `F`, resulting in just one instantiation. Listing 18.8 shows the result.

Listing 18.8 **wrapped.cpp**

```
//wrapped.cpp -- using a function wrapper as an argument
#include "somedefs.h"
#include <iostream>
#include <functional>

double dub(double x) {return 2.0*x;}
double square(double x) {return x*x;}

int main()
{
    using std::cout;
    using std::endl;
    using std::function;

    double y = 1.21;
    function<double(double)> ef1 = dub;
    function<double(double)> ef2 = square;
    function<double(double)> ef3 = Fq(10.0);
    function<double(double)> ef4 = Fp(10.0);
    function<double(double)> ef5 =  [](double u) {return u*u;};
    function<double(double)> ef6 =  [](double u) {return u+u/2.0;};
    cout << "Function pointer dub:\n";
    cout << "  " << use_f(y, ef1) << endl;
    cout << "Function pointer square:\n";
    cout << "  " << use_f(y, ef2) << endl;
    cout << "Function object Fp:\n";
    cout << "  " << use_f(y, ef3) << endl;
    cout << "Function object Fq:\n";
    cout << "  " << use_f(y, ef4) << endl;
    cout << "Lambda expression 1:\n";
    cout << "  " << use_f(y, ef5) << endl;
    cout << "Lambda expression 2:\n";
    cout << "  " << use_f(y,ef6) << endl;
    return 0;
}
```

Here is a sample output:

```
Function pointer dub:
  use_f count = 1, &count = 0x404020
  2.42
Function pointer sqrt:
  use_f count = 2, &count = 0x404020
  1.1
```

```
Function object Fp:
  use_f count = 3, &count = 0x404020
  11.21
Function object Fq:
  use_f count = 4, &count = 0x404020
  12.1
Lambda expression 1:
  use_f count = 5, &count = 0x404020
  1.4641
Lambda expression 2:
  use_f count = 6, &count = 0x404020
  1.815
```

As you can see from the output, there is just one address for `count`, and the value of `count` shows `use_f()` has been called six times. So we now have just one instantiation invoked six times, reducing the size of the executable code.

Further Options

Let's look at a couple more things you can do using `function`. First, we don't actually have to declare six `function<double(double)>` objects in Listing 18.8. Instead, we can use a temporary `function<double(double)>` object as an argument to the `use_f()` function:

```
typedef function<double(double)> fdd; // simplify the type declaration
cout << use_f(y, fdd(dub)) << endl;   // create and initialize object to dub
cout << use_f(y, fdd(square)) << endl;
...
```

Second, Listing 18.8 adapts the second arguments in `use_f()` to match the formal parameter `f`. Another approach is to adapt the type of the formal parameter `f` to match the original arguments. This can be done by using a function wrapper object as the second parameter for the `use_f()` template definition. We can define `use_f()` this way:

```
#include <functional>
template <typename T>
T use_f(T v,  std::function<T(T)> f)   // f call signature is T(T)
{
    static int count = 0;
    count++;
    std::cout << "  use_f count = " << count
              << ", &count = " << &count << std::endl;
    return f(v);
}
```

Then the function calls can look like this:

```
cout << "  " << use_f<double>(y, dub) << endl;
...
```

```
cout << "  " << use_f<double>(y, Fp(5.0)) << endl;
...
cout << "  " << use_f<double>(y, [](double u) {return u*u;}) << endl;
```

The arguments `dub`, `Fp(5.0)`, etc., are not themselves type `function<double(double)>`, and so the calls use `<double>` after `use_f` to indicate the desired specialization. Thus `T` is set to `double`, and `std::function<T(T)>` becomes `std::function<double(double)>`.

Variadic Templates

Variadic templates provide a means to create template functions and template classes that accept a variable number of arguments. We'll look at variadic template functions here. For example, suppose we want a function that will accept any number of parameters of any type, providing the type can be displayed with `cout`, and display the arguments as a comma-separated list. For instance, consider this code:

```
int n = 14;
double x = 2.71828;
std::string mr = "Mr. String objects!";
show_list(n, x);
show_list(x*x, '!', 7, mr);
```

The goal is to be able to define `show_list()` in such a way that this code would compile and lead to this output:

```
14, 2.71828
7.38905, !, 7, Mr. String objects!
```

There are a few key points to understand in order to create variadic templates:

- Template parameter packs
- Function parameter packs
- Unpacking a pack
- Recursion

Template and Function Parameter Packs

As a starting point to see how parameter packs work, let's consider a simple template function, one that displays a list consisting of just one item:

```
template<typename T>
void show_list0(T value)
{
    std::cout << value << ", ";
}
```

This definition has two parameter lists. The template parameter list is just `T`. The function parameter list is just `value`. A function call such as the following sets `T` in the template parameter list to `double` and `value` in the function parameter list to `2.15`:

```
show_list0(2.15);
```

C++11 provides an ellipsis meta-operator that enables you to declare an identifier for a template parameter pack, essentially a list of types. Similarly, it lets you declare an identifier for a function parameter pack, essentially a list of values. The syntax looks like this:

```
template<typename... Args>    // Args is a template parameter pack
void show_list1(Args... args) // args is a function parameter pack
{
...
}
```

`Args` is a template parameter pack, and `args` is a function parameter pack. (As with other parameter names, any name satisfying C++ identifier rules can be used for these packs.) The difference between `Args` and `T` is that `T` matches a single type, whereas `Args` matches any number of types, including none. Consider the following function call:

```
show_list1('S', 80, "sweet", 4.5);
```

In this case the parameter pack `Args` contains the types matching the parameters in the function call: `char`, `int`, `const char *`, and `double`.

Next, much as

```
void show_list0(T value)
```

states that `value` is of type `T`, the line

```
void show_list1(Args... args)  // args is a function parameter pack
```

states that `args` is of type `Args`. More precisely, this means that the function pack `args` contains a list of values that matches the list of types in the template pack `Args`, both in type and in number. In this case, `args` contains the values `'S'`, `80`, `"sweet"`, and `4.5`.

In this manner, the `show_list1()` variadic template can match any of the following function calls:

```
show_list1();
show_list1(99);
show_list1(88.5, "cat");
show_list1(2,4,6,8, "who do we", std::string("appreciate"));
```

In the last case, the `Args` template parameter pack would contain the types `int`, `int`, `int`, `int`, `const char *`, and `std::string`, and the `args` function parameter pack would contain the matching values `2`, `4`, `6`, `8`, `"who do we"`, and `std::string("appreciate")`.

Unpacking the Packs

But how can the function access the contents of these packs? There is no indexing feature. That is, you can't use something like `Args[2]` to access the third type in a pack.

Instead, you can unpack the pack by placing the ellipsis to the right of the function parameter pack name. For example, consider the following flawed code:

```
template<typename... Args>    // Args is a template parameter pack
void show_list1(Args... args) // args is a function parameter pack
{
    show_list1(args...); // passes unpacked args to show_list1()
}
```

What does this mean, and why is it flawed? Suppose we have this function call:

```
show_list1(5,'L',0.5);
```

The call packs the values `5`, `'L'`, and `0.5` into `args`. Within the function, the call

```
show_list1(args...);
```

expands to the following:

```
show_list1(5,'L',0.5);
```

That is, the single entity `args` is replaced by the three values stored within the pack. So the notation `args...` expands to a list of discrete function arguments. Unfortunately, the new call is the same as the original function call, so it will call itself again with the same arguments, initiating an infinite and futile recursion. (That would be the flaw.)

Using Recursion in Variadic Template Functions

Although recursion dooms `show_list1()` aspirations to be a useful function, properly used recursion provides a solution to accessing pack items. The central idea is to unpack the function parameter pack, process the first item in the list, then pass the rest of the list on to a recursive call, and so on, until the list is empty. As usual with recursion, it's important to make sure that there is a call that terminates the recursion. Part of the trick involves changing the template heading to this:

```
template<typename T, typename... Args>
void show_list3( T value, Args... args)
```

With this definition, the first argument to `show_list3()` gets picked up as type `T` and is assigned to `value`. The remaining arguments are picked up by `Args` and `args`. This allows the function to do something with `value`, such as display it. Then the remaining arguments, in the form `args...`, can be passed to a recursive call of `show_list3()`. Each recursive call then prints a value and passes on a shortened list until the list is exhausted.

Listing 18.9 presents an implementation that, although not perfect, illustrates the technique.

Listing 18.9 **variadic1.cpp**

```
//variadic1.cpp -- using recursion to unpack a parameter pack
#include <iostream>
#include <string>
```

```
// definition for 0 parameters -- terminating call
void show_list3() {}

// definition for 1 or more parameters
template<typename T, typename... Args>
void show_list3( T value, Args... args)
{
    std::cout << value << ", ";
    show_list3(args...);
}

int main()
{
    int n = 14;
    double x = 2.71828;
    std::string mr = "Mr. String objects!";
    show_list3(n, x);
    show_list3(x*x, '!', 7, mr);
    return 0;
}
```

Program Notes

Consider this function call:

```
show_list3(x*x, '!', 7, mr);
```

The first argument matches `T` to `double` and `value` to `x*x`. The remaining three types (`char`, `int`, and `std::string`) are placed in the `Args` pack, and the remaining three values (`'!'`, `7`, and `mr`) are placed in the `args` pack.

Next, the `show_list3()` function uses `cout` to display `value` (approximately `7.38905`) and the string `", "`. That takes care of displaying the first item in the list.

Next comes this call:

```
show_list3(args...);
```

This, given the expansion of `args...`, is the same as the following:

```
show_list3('!', 7, mr);
```

As promised, the list is shortened by one item. This time `T` and `value` become `char` and `'!'`, and the remaining two types and values are packed into `Args` and `args`, respectively. The next recursive call processes these reduced packs. Finally, when `args` is empty, the version of `show_list3()` with no arguments is called, and the process terminates.

Here is the output for the two function calls in Listing 18.5:

```
14, 2.71828, 7.38905, !, 7, Mr. String objects!,
```

Improvements

We can improve `show_list3()` with a couple of changes. As it stands, the function displays a comma after every item in the list, but it would be better to omit the comma after the last item. This can be accomplished by adding a template for just one item and having it behave slightly differently from the general template:

```
// definition for 1 parameter
template<typename T>
void show_list3(T value)
{
    std::cout << value << '\n';
}
```

Thus, when the `args` pack is reduced to one item, this version is called, and it prints a newline instead of a comma. Because it lacks a recursive call to `show_list3()`, it also terminates the recursion.

The second area for improvement is that the current version passes everything by value. This is okay for the simple types we've used, but it's inefficient for classes of large size that might be printable by `cout`. It would be better to use `const` references. With variadic templates, you can impose a *pattern* on the unpacking. Instead of using

```
show_list3(Args... args);
```

you can use this:

```
show_list3(const Args&... args);
```

That will cause each function parameter to have the `const&` pattern applied. Thus, instead of `std::string mr`, the final paring of parameters becomes `const std::string& mr`.

Listing 18.10 incorporates these two changes.

Listing 18.10 **variadic2.cpp**

```
// variadic2.cpp
#include <iostream>
#include <string>

// definition for 0 parameters
void show_list() {}

// definition for 1 parameter
template<typename T>
void show_list(const T& value)
{
    std::cout << value << '\n';
}
```

```
// definition for 2 or more parameters
template<typename T, typename... Args>
void show_list(const T& value, const Args&... args)
{
    std::cout << value << ", ";
    show_list(args...);
}

int main()
{
    int n = 14;
    double x = 2.71828;
    std::string mr = "Mr. String objects!";
    show_list(n, x);
    show_list(x*x, '!', 7, mr);
    return 0;
}
```

Here is the output:

```
14, 2.71828
7.38905, !, 7, Mr. String objects!
```

More C++11 Features

C++11 adds many more features than this book can cover, even ignoring the fact that many of them are not widely implemented at the time this book was written. Still, it's worth taking a quick look at the nature of some of these features.

Concurrent Programming

These days it's easier to improve computer performance by adding processors than it is to increase processor speed. So computers with dual-core or quad-core processors or even with multiple multicore processors are the norm. This enables computers to run multiple threads of execution simultaneously. One processor might handle a video download while another processor handles your spreadsheet.

Some activities can benefit from multiple threads, whereas others don't. Consider searching for something in a singly linked list. A program has to start at the beginning and follow the links, in order, to the end of the list; there's nothing much a second thread could do to help. Now consider an unsorted array. Using the random access feature of arrays, you could start one thread from the beginning of the array and one from the middle, thus halving the search time.

Multiple threads do raise many problems. What happens if one thread hangs up or if two threads try to access the same data simultaneously? C++11 addresses concurrency by

defining a memory model that supports threaded execution, by adding the keyword `thread_local`, and by providing library support. The keyword `thread_local` is used to declare variables having static storage duration relative to a particular thread; that is, they expire when the thread in which they are defined expires.

The library support consists of the atomic operations library, which specifies the `atomic` header file, and the thread support library, which specifies the `thread`, `mutex`, `condition_variable`, and `future` header files.

Library Additions

C++11 adds several specialized libraries. An extensible random number library, supported by the `random` header file, provides much more extensive and sophisticated random number facilities than `rand()`. For example, it offers a choice of random-number generators and a choice of distributions, including uniform (like `rand()`), binomial, normal, and several others.

The `chrono` header file supports ways to deal with time duration.

The `tuple` header file supports the `tuple` template. A `tuple` object is a generalization of a `pair` object. Whereas a `pair` object can hold two values whose types need not be the same, a `tuple` can hold an arbitrary number of items of different types.

The compile-time rational arithmetic library, supported by the `ratio` header file, allows the exact representation of any rational number whose numerator and denominator can be represented by the widest integer type. It also provides arithmetic operations for these numbers.

One of the most interesting additions is a regular expression library, supported by the `regex` header file. A regular expression specifies a pattern that can be used to match contents in a text string. For example, a bracket expression matches any single character in the brackets. Thus, `[cCkK]` matches a single `c`, `C`, `k`, or `K`, and `[cCkK]at` matches the words `cat`, `Cat`, `kat`, and `Kat`. Other patterns include `\d` for a digit, `\w` for a word, `\t` for a tab, and many, many others. The fact that a backslash has a special meaning in C++ as the first character in an escape sequence requires a pattern like `\d\t\w\d` (that is, digit-tab-word-digit) to be written as the string literal `"\\d\\t\\w\\d"`, using `\\` to represent `\`. This is one reason the raw string was introduced (see Chapter 4); it enables you to write the same pattern as `R"\d\t\w\t"`.

Unix utilities such as `ed`, `grep`, and `awk` used regular expressions, and the interpreted language Perl extended their capabilities. The C++ regular expressions library allows you to choose from several flavors of regular expressions.

Low-Level Programming

The "low level" in low-level programming refers to the level of abstraction, not to the quality of the programming. Low level means closer to the bits and bytes of computer hardware and machine language. Low-level programming is important for embedded programming and for increasing the efficiency of some operations. C++11 offers some aids to those who do low-level programming.

One change is relaxing the constraints on what qualifies as "Plain Old Data," or POD. In C++98, a POD is a scalar type (a one-value type, such as `int` or `double`) or an old-fashioned structure with no constructors, base classes, private data, virtual functions, and so on. The idea was that a POD is something for which it's safe to make a byte-by-byte copy. That is still the idea, but C++11 recognizes one can remove some the C++98 restrictions and still have a viable POD. This helps low-level programming because some low-level operations, such as using the C functions for byte-wise copying or binary I/O, required PODs.

Another change is making unions more flexible by allowing them to have members that have constructors and destructors, but with some restrictions on other properties, for example, not allowing virtual functions. Unions are often utilized when minimizing the amount of memory used is important, and the new rules allow programmers to bring more flexibility and capability to these situations.

C++11 addresses memory alignment issues. Computer systems can restrict how data is stored in memory. For example, one system might require that a `double` value be stored at an even-numbered memory location, whereas another might require the storage to begin at a location that is a multiple of eight. The `alignof()` operator (see Appendix E, "Other Operators") provides information on the alignment requirements for a type or object. The `alignas` specifier provides some control over the alignment used.

The `constexpr` mechanism expands the ability of the compiler to evaluate during compile time expressions that evaluate to a constant value. The low-level aspect of this is to allow `const` variables to be stored in read-only memory, which can be particularly useful in embedded programming. (Variables, `const` or otherwise, that are initialized during runtime, are stored in random-access memory.)

Miscellaneous

C++11 follows the lead of C99 in allowing for implementation-dependent extended integer types. Such types, for example, could be used on a system with 128-bit integers. Extended types are supported in the C header file `stdint.h` and in the C++ version, `cstdint`.

C++11 provides a mechanism, the *literal operator*, for creating user-defined literals. Using this mechanism, for instance, one can define binary literals, such as `1001001b`, which the corresponding literal operator will convert to an integer value.

C++ has a debugging tool called `assert`. It is a macro that checks during runtime if an assertion is true and which displays a message and calls `abort()` if the assertion is false. The assertion would typically be about something the programmer thinks should be true at that point in the program. C++11 adds the keyword `static_assert`, which can be used to test assertions during compile time. The primary motivation is to make it easier to debug templates for which instantiation takes place during compile time, not runtime.

C++11 provides more support for metaprogramming, which is creating programs that create or modify other programs or even themselves. In C++ this can be done during compile time using templates.

Language Change

How does a computer language grow and evolve? after usage of C++ became sufficiently widespread, the need for an international standard became clear, and control of the language essentially passed to a standards committee—first the ANSI committee, then the joint ISO/ANSI committees, and currently to ISO/IEC JTC1/SC22/WG21 (the C++ Standards Committee). ISO is the International Organization for Standardization, IEC is the International Electrotechnical Commission, JTC1 is the Joint Technical Committee 1 of the preceding two organizations, SC22 is the JTC1 subcommittee for programming languages, and WG21 is the SC22 working group for C++.

The committee considers defect reports and proposals for changes in and extensions to the language, and it attempts to reach a consensus. This is neither a speedy nor a simple process. *The Design and Evolution of C++* (Stroustrup, Addison-Wesley, 1994) provides some insight on this subject. Anyway, the possibly contentious and ponderous dynamics of a consensus-seeking committee are not the best way to encourage the blossoming of a multitude of innovations. Nor is that the proper role of a standards committee.

But with C++ there is a second avenue for change: direct action by the creative C++ programming community. Programmers can't independently change the language, but they can create useful libraries. Well-designed libraries can extend the usefulness and versatility of the language, increase reliability, and make programming easier and more enjoyable. Libraries build on the existing features of a language, so they don't require any additional compiler support. And if they are implemented through templates, they can be distributed as text files in the form of header files.

One example of this sort of change is the STL, primarily created by Alexander Stepanov, and made freely available by Hewlett-Packard. Its success with the programming community made it a candidate for the first ANSI/ISO standard. Indeed, its design influenced other aspects of the emerging standard.

The Boost Project

More recently, the Boost library has become an important part of C++ programming and has had a significant influence on C++11. The Boost project began in 1998 when Beman Dawes, the then-chairman of the C++ library working group, brought together a few other members of the group and developed a plan to generate new libraries outside the confines of the standards committee. The basic idea was to provide a website that acts as an open forum for people to post free C++ libraries. The project provides guidelines for licensing and programming practices, and it requires peer review of proposed libraries. The result is a group of highly praised and highly used libraries. The project provides an environment in which the programming community can test and evaluate programming ideas and provide feedback.

Boost has over 100 libraries at the time of this writing, and they can be downloaded as a set from www.boost.org, as can documentation. Most of the libraries can be used by including the appropriate header files.

The TR1

The Technical Report 1, or TR1, was a project of a subset of the C++ Standards committee. TR1 was a compilation of library extensions that were compatible with the C++98 standard, but which were not required by the standard. They were candidates for the next iteration of the standard. The TR1 library enabled the C++ community to test the worthiness of the library components. Thus, when the standards committee incorporated most of TR1 into C++11, it was dealing with known and proven libraries.

The Boost libraries contributed much to TR1. Examples include the `tuple` template class, the `array` template class, the `bind` and `function` templates, smart pointers (with some name and implementation changes), `static_assert`, the `regex` library, and the `random` library. Furthermore, the experiences of the Boost community and of users of the TR1 led to actual language changes, such as the deprecation of exception specifications, and the addition of variadic templates, which allows for a better implementation of the `tuple` template class and of the `function` template.

Using Boost

Although you now can access many Boost-developed libraries as part of the C++11 standard, there are many additional Boost libraries to explore. For example, `lexical_cast` from the Conversion library provides simple conversions between numeric and string types. The syntax is modeled after `dynamic_cast`, in which you provide the target type as a template parameter. Listing 18.11 shows a simple example.

Listing 18.11 **lexcast.cpp**

```
// lexcast.cpp -- simple cast from float to string
#include <iostream>
#include <string>
#include "boost/lexical_cast.hpp"
int main()
{
    using namespace std;
    cout << "Enter your weight: ";
    float weight;
    cin >> weight;
    string gain = "A 10% increase raises ";
    string wt = boost::lexical_cast<string>(weight);
    gain = gain + wt + " to ";  // string operator+()
    weight = 1.1 * weight;
    gain = gain + boost::lexical_cast<string>(weight) + ".";
    cout << gain << endl;
    return 0;
}
```

Here are a couple of sample runs:

```
Enter your weight: 150
A 10% increase raises 150 to 165.

Enter your weight: 156
A 10% increase raises 156 to 171.600006.
```

The second sample run shows one of the limitations of `lexical_cast`; it doesn't provide fine control over floating-point formatting. To get that, you need to use the more elaborate incore formatting facilities discussed in Chapter 17, "Input, Output, and Files."

You also can use `lexical_cast` to convert strings to numeric values.

Obviously, there is much, much more to Boost. For example, the Any library allows you to store (and recover) a heterogeneous collection of values and objects of different types in an STL container, using the `Any` template as a wrapper for the various values. The Boost Math library extends the list of math functions beyond that of the standard math library. The Boost Filesystem library facilitates writing code that is portable across platforms using different file systems. You can refer to the Boost website (www.boost.org) to get more information about the contents of the library and how to add it to various platforms. Also some C++ distributions, such as the one from Cygwin, come with the Boost library included.

What Now?

If you have worked your way through this book, you should have a good grasp of the rules of C++. However, that's just the beginning in learning this language. The second stage is learning to use the language effectively, and that is the longer journey. The best situation is to be in a work or learning environment that brings you into contact with good C++ code and programmers. Also now that you know C++, you can read books that concentrate on more advanced topics and on object-oriented programming. Appendix H, "Selected Readings and Internet Resources," lists some of these resources.

One promise of OOP is to facilitate the development and enhance the reliability of large projects. One of the essential activities of the OOP approach is to invent the classes that represent the situation (called the *problem domain*) that you are modeling. Because real problems are often complex, finding a suitable set of classes can be challenging. Creating a complex system from scratch usually doesn't work; instead, it's best to take an iterative, evolutionary approach. Toward this end, practitioners in the field have developed several techniques and strategies. In particular, it's important to do as much of the iteration and evolution in the analysis and design stages as possible instead of writing and rewriting actual code.

Two common techniques are *use-case analysis* and *CRC cards*. In use-case analysis, the development team lists the common ways, or scenarios, in which it expects the final system to be used, identifying elements, actions, and responsibilities that suggest possible

classes and class features. Using CRC (short for Class/Responsibilities/Collaborators) cards is a simple way to analyze such scenarios. The development team creates an index card for each class. On the card are the class name; class responsibilities, such as data represented and actions performed; and class collaborators, such as other classes with which the class must interact. Then the team can walk through a scenario, using the interface provided by the CRC cards. This can lead to suggestions for new classes, shifts of responsibility, and so on.

On a larger scale are the systematic methods for working on entire projects. The most recent of these is the Unified Modeling Language (UML). UML is not a programming language; rather, it is a language for representing the analysis and design of a programming project. It was developed by Grady Booch, Jim Rumbaugh, and Ivar Jacobson, who had been the primary developers of three earlier modeling languages: the Booch Method, OMT (Object Modeling Technique), and OOSE (Object-Oriented Software Engineering), respectively. UML is the evolutionary successor of these three, and ISO/IEC published a standard for it in 2005.

In addition to increasing your understanding of C++ in general, you might want to learn about specific class libraries. Microsoft and Embarcadero, for example, offer extensive class libraries to facilitate programming for the Windows environment, and Apple Xcode offers similar facilities for Apple products, including the iPhone.

Summary

The new C++ standard adds many features to the language. Some are intended to make the language easier to learn and easier to use. Examples include uniform braced list initialization, automatic type deduction with `auto`, in-class member initialization, and the range-based `for` loop. Other changes expand and clarify class design. These changes include defaulted and deleted methods, delegated constructors, inherited constructors, and the `override` and `final` specifiers for clarifying virtual function design.

Several additions aim to make programs and the act programming more efficient. Lambda expressions provide advantages over function pointers and functors. The `function` template can be used to reduce the number of template instantiations. The rvalue reference enables move semantics and allows for move constructors and move assignment operators.

Other changes deliver better ways of doing things. Scoped enumerations provide better control over the scope and underlying types for enumerations. The `unique_ptr` and `shared_ptr` templates provide better ways of handling memory allocated with `new`.

Template design has been enhanced with the addition of `decltype`, trailing return types, template aliases, and variadic templates.

Modified rules for unions, PODs, the `alignof()` operator, the `alignas` specifier, and the `constexpr` mechanism support low-level programming.

Several library additions, including the new STL classes, the `tuple` template, and the `regex` library, provide solutions to many common programming needs.

The new standard addresses concurrent programming with the `thread_local` keyword and the `atomic` library.

All in all, the new standard improves the usability and reliability of C++, both for novices and for experts.

Chapter Review

1. Rewrite the following code using braced initialization list syntax; the rewrite should dispense with using the array ar:

```
class Z200
{
private:
    int j;
    char ch;
    double z;
public:
    Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
...
};

double x = 8.8;
std::string s = "What a bracing effect!";
int k(99);
Z200 zip(200,'Z',0.675);
std::vector<int> ai(5);
int ar[5] = {3, 9, 4, 7, 1};
for (auto pt = ai.begin(), int i = 0; pt != ai.end(); ++pt, ++i)
    *pt = ai[i];
```

2. For the following short program, which function calls are errors and why? For the valid calls, what does the reference argument refer to?

```
#include <iostream>
using namespace std;

double up(double x) { return 2.0* x;}
void r1(const double &rx) {cout << rx << endl;}
void r2(double &rx) {cout << rx << endl;}
void r3(double &&rx) {cout << rx << endl;}

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
```

```
    r2(w);
    r2(w+1);
    r2(up(w));
    r3(w);
    r3(w+1);
    r3(up(w));
    return 0;
}
```

3. a. What does the following short program display and why?

```
#include <iostream>
using namespace std;

double up(double x) { return 2.0* x;}
void r1(const double &rx) {cout << "const double & rx\n";}
void r1(double &rx) {cout << "double & rx\n";}

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    return 0;
}
```

   b. What does the following short program display and why?

```
#include <iostream>
using namespace std;

double up(double x) { return 2.0* x;}
void r1(double &rx) {cout << "double & rx\n";}
void r1(double &&rx) {cout << "double && rx\n";}

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    return 0;
}
```

   c. What does the following short program display and why?

```
#include <iostream>
using namespace std;
```

```
double up(double x) {return 2.0* x;}
void r1(const double &rx) {cout << "const double & rx\n";}
void r1(double &&rx) {cout << "double && rx\n";}

int main()
{
    double w = 10.0;
    r1(w);
    r1(w+1);
    r1(up(w));
    return 0;
}
```

4. Which member functions are special member functions, and what makes them special?

5. Suppose the `Fizzle` class has only the data members shown:

```
class Fizzle
{
private:
    double bubbles[4000];
...
};
```

   Why would this class not be a good candidate for a user-defined move constructor? What change in approach to storing the 4000 double values would make the class a good candidate for a move function?

6. Revise the following short program so that it uses a lambda expression instead of `f1()`. Don't change `show2()`.

```
#include <iostream>
template<typename T>
  void show2(double x, T& fp) {std::cout << x << " -> " << fp(x) << '\n';}
double f1(double x) { return 1.8*x + 32;}
int main()
{
    show2(18.0, f1);
    return 0;
}
```

7. Revise the following short and ugly program so that it uses a lambda expression instead of the `Adder` functor. Don't change `sum()`.

```
#include <iostream>
#include <array>
const int Size = 5;
template<typename T>
```

```
  void sum(std::array<double,Size> a, T& fp);
class Adder
{
    double tot;
public:
    Adder(double q = 0) : tot(q) {}
    void operator()(double w) { tot +=w;}
    double tot_v () const {return tot;};
};
int main()
{
    double total = 0.0;
    Adder ad(total);
    std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
    sum(temp_c,ad);
    total = ad.tot_v();
    std::cout << "total: " << ad.tot_v() << '\n';
    return 0;
}
template<typename T>
  void sum(std::array<double,Size> a, T& fp)
{
    for(auto pt = a.begin(); pt != a.end(); ++pt)
    {
        fp(*pt);
    }
}
```

Programming Exercises

1. Here is part of a short program:

```
int main()
{
    using namespace std;
 // list of double deduced from list contents
    auto q = average_list({15.4, 10.7, 9.0});
    cout << q << endl;
// list of int deduced from list contents
    cout << average_list({20, 30, 19, 17, 45, 38} ) << endl;
// forced list of double
    auto ad = average_list<double>({'A', 70, 65.33});
    cout << ad << endl;
    return 0;
}
```

Complete the program by supplying the `average_list()` function. It should be a template function, with the type parameter being used to specify the kind of `initialized_list` template to be used as the function parameter and also to give the function return type.

2. Here is declaration for the `Cpmv` class:

```
class Cpmv
{
public:
    struct Info
    {
        std::string qcode;
        std::string zcode;
    };
private:
    Info *pi;
public:
    Cpmv();
    Cpmv(std::string q, std::string z);
    Cpmv(const Cpmv & cp);
    Cpmv(Cpmv && mv);
    ~Cpmv();
    Cpmv & operator=(const Cpmv & cp);
    Cpmv & operator=(Cpmv && mv);
    Cpmv operator+(const Cpmv & obj) const;
    void Display() const;
};
```

   The `operator+()` function should create an object whose `qcode` and `zcode` members concatenate the corresponding members of the operands. Provide code that implements move semantics for the move constructor and the move assignment operator. Write a program that uses all the methods. For testing purposes, make the various methods verbose so that you can see when they are used.

3. Write and test a variadic template function `sum_values()` that accepts an arbitrarily long list of arguments with numeric values (they can be a mixture of types) and returns the sum as a `long double` value.

4. Redo Listing 16.5 using lambdas. In particular, replace the `outint()` function with a named lambda and replace the two uses of a functor with two anonymous lambda expressions.