24

C++11: Additional Features

Objectives

In this chapter you'll:

- Avoid memory leaks by using smart pointers to manage dynamic memory.
- Use multithreading to execute compute intensive tasks in parallel, especially on multicore systems.
- Use rvalue references and move semantics to eliminate unnecessary object copies, improving program performance.
- Learn about C++11 language features, including raw string literals `static_assert`, `noexcept`, `decltype`, `constexpr`, variadic templates and template aliases.
- Use = `default` to generate default versions of the special member functions (that is, the constructors, assignment operators and destructor that the compiler can generate for a class).
- Create functions that can receive list initalizers as arguments.
- Use regular expressions to search for strings, validate data and replace substrings.

Outline

Note: **This chapter is online so that we can update it as key C++ compilers gradually provide more support for C++11.**

24.1 Introduction

11

In this chapter, we consider additional C++11 features, including more features of the core language and various C++ Standard Library enhancements. This chapter is not exhaustive—many of C++11's new features are meant for class library implementers and are beyond this book's scope. The topics in this chapter do not need to be read sequentially. Throughout the chapter, we provide links to many online resources for further study.

The chapter's examples were tested on Microsoft Visual C++ 2012, GNU C++ 4.7 and Apple Xcode LLVM. We noted any issues we encountered with these compilers—support for many C++11 features varies across compilers.

24.2 Smart Pointers

11

Many common bugs in C and C++ code are related to pointers. **Smart pointers** help you avoid errors by providing additional functionality to standard pointers. This functionality typically strengthens the process of memory allocation and deallocation. Smart pointers also help you write exception safe code. If a program throws an exception before `delete` has been called on a pointer, it creates a memory leak. After an exception is thrown, a smart pointer's destructor will still be called, which calls `delete` on the pointer for you.

Section 17.9 showed one of the smart pointer classes—`unique_ptr`—which is responsible for managing dynamically allocated memory. A `unique_ptr` automatically calls `delete` to free its associated dynamic memory when the `unique_ptr` is destroyed or

goes out of scope. A `unique_ptr` is a basic smart pointer. C++11 provides other smart pointer options with additional functionality. The examples in this section were tested on the Microsoft Visual C++ 2012, Apple Xcode LLVM and GNU C++ 4.7 compilers.

24.2.1 Reference Counted `shared_ptr`

`shared_ptr`s (from header `<memory>` hold an internal pointer to a resource (e.g., a dynamically allocated object) that may be shared with other objects in the program. You can have any number of `shared_ptr`s to the same resource. `shared_ptr`s really do share the resource—if you change the resource with one `shared_ptr`, the changes also will be "seen" by the other `shared_ptr`s. The internal pointer is deleted once the last `shared_ptr` to the resource is destroyed. `shared_ptr`s use **reference counting** to determine how many `shared_ptr`s point to the resource. Each time a new `shared_ptr` to the resource is created, the **reference count** increases, and each time one is destroyed, the reference count decreases. When the reference count reaches zero, the internal pointer is deleted and the memory is released.

`shared_ptr`s are useful in situations where multiple pointers to the same resource are needed, such as in STL containers. `shared_ptr`s can safely be copied and used in STL containers.

`shared_ptr`s also allow you to determine how the resource will be destroyed. For most dynamically allocated objects, `delete` is used. However, some resources require more complex cleanup. In that case, you can supply a custom **deleter** function, or function object, to the `shared_ptr` constructor. The deleter determines how to destroy the resource. When the reference count reaches zero and the resource is ready to be destroyed, the `shared_ptr` calls the custom deleter function. This functionality enables a `shared_ptr` to manage almost any kind of resource.

Example Using ***`shared_ptr`***

Figure 24.1–Fig. 24.2 define a simple class to represent a `Book` with a `string` to represent the title of the `Book`. The destructor for class `Book` (Fig. 24.2, lines 12–15) displays a message on the screen indicating that an instance is being destroyed. We use this class to demonstrate the basic functionality of `shared_ptr`.

```
1  // Fig. 24.1: Book.h
2  // Declaration of class Book.
3  #ifndef BOOK_H
4  #define BOOK_H
5  #include <string>
6  using namespace std;
7
8  class Book
9  {
10 public:
11    Book( const string &bookTitle ); // constructor
12    ~Book(); // destructor
13    string title; // title of the Book
14 };
15 #endif // BOOK_H
```

Fig. 24.1 | `Book` header.

```
1  // Fig. 24.2: Book.cpp
2  // Member-function definitions for class Book.
3  #include <iostream>
4  #include <string>
5  #include "Book.h"
6  using namespace std;
7
8  Book::Book( const string &bookTitle ) : title( bookTitle )
9  {
10 }
11
12 Book::~Book()
13 {
14    cout << "Destroying Book: " << title << endl;
15 } // end of destructor
```

Fig. 24.2 | `Book` member-function definitions.

Creating `shared_ptr`*s*

The program in Fig. 24.3 uses `shared_ptr`s (from the header `<memory>`) to manage several instances of class `Book`. We also create a `typedef`, `BookPtr`, as an alias for the type `shared_ptr<Book>` (line 10). Line 28 creates a `shared_ptr` to a `Book` titled `"C++ How to Program"` (using the `BookPtr` `typedef`). The `shared_ptr` constructor takes as its argument a pointer to an object. We pass it the pointer returned from the `new` operator. This creates a `shared_ptr` that manages the `Book` object and sets the reference count to one. The constructor can also take another `shared_ptr`, in which case it shares ownership of the resource with the other `shared_ptr` and the reference count is increased by one. The first `shared_ptr` to a resource should always be created using the `new` operator. A `shared_ptr` created with a regular pointer assumes it's the first `shared_ptr` assigned to that resource and starts the reference count at one. If you make multiple `shared_ptr`s with the same pointer, the `shared_ptr`s won't acknowledge each other and the reference count will be wrong. When the `shared_ptr`s are destroyed, they both call `delete` on the resource.

```
1  // Fig. 24.3: fig24_03.cpp
2  // Demonstrate shared_ptrs.
3  #include <algorithm>
4  #include <iostream>
5  #include <memory>
6  #include <vector>
7  #include "Book.h"
8  using namespace std;
9
10 typedef shared_ptr< Book > BookPtr; // shared_ptr to a Book
11
12 // a custom delete function for a pointer to a Book
13 void deleteBook( Book* book )
14 {
15    cout << "Custom deleter for a Book, ";
```

Fig. 24.3 | `shared_ptr` example program. (Part 1 of 3.)

```
16    delete book; // delete the Book pointer
17 } // end of deleteBook
18
19 // compare the titles of two Books for sorting
20 bool compareTitles( BookPtr bookPtr1, BookPtr bookPtr2 )
21 {
22    return ( bookPtr1->title < bookPtr2->title );
23 } // end of compareTitles
24
25 int main()
26 {
27    // create a shared_ptr to a Book and display the reference count
28    BookPtr bookPtr( new Book( "C++ How to Program" ) );
29    cout << "Reference count for Book " << bookPtr->title << " is: "
30       << bookPtr.use_count() << endl;
31
32    // create another shared_ptr to the Book and display reference count
33    BookPtr bookPtr2( bookPtr );
34    cout << "Reference count for Book " << bookPtr->title << " is: "
35       << bookPtr.use_count() << endl;
36
37    // change the Book's title and access it from both pointers
38    bookPtr2->title = "Java How to Program";
39    cout << "The Book's title changed for both pointers: "
40       << "\nbookPtr: " << bookPtr->title
41       << "\nbookPtr2: " << bookPtr2->title << endl;
42
43    // create a std::vector of shared_ptrs to Books (BookPtrs)
44    vector< BookPtr > books;
45    books.push_back( BookPtr( new Book( "C How to Program" ) ) );
46    books.push_back( BookPtr( new Book( "VB How to Program" ) ) );
47    books.push_back( BookPtr( new Book( "C# How to Program" ) ) );
48    books.push_back( BookPtr( new Book( "C++ How to Program" ) ) );
49
50    // print the Books in the vector
51    cout << "\nBooks before sorting: " << endl;
52    for ( int i = 0; i < books.size(); ++i )
53       cout << ( books[ i ] )->title << "\n";
54
55    // sort the vector by Book title and print the sorted vector
56    sort( books.begin(), books.end(), compareTitles );
57    cout << "\nBooks after sorting: " << endl;
58    for ( int i = 0; i < books.size(); ++i )
59       cout << ( books[ i ] )->title << "\n";
60
61    // create a shared_ptr with a custom deleter
62    cout << "\nshared_ptr with a custom deleter." << endl;
63    BookPtr bookPtr3( new Book( "Small C++ How to Program" ), deleteBook);
64    bookPtr3.reset(); // release the Book this shared_ptr manages
65
66    // shared_ptrs are going out of scope
67    cout << "\nAll shared_ptr objects are going out of scope." << endl;
68 } // end of main
```

Fig. 24.3 | `shared_ptr` example program. (Part 2 of 3.)

```
Reference count for Book C++ How to Program is: 1
Reference count for Book C++ How to Program is: 2

The Book's title changed for both pointers:
bookPtr: Java How to Program
bookPtr2: Java How to Program

Books before sorting:
C How to Program
VB How to Program
C# How to Program
C++ How to Program

Books after sorting:
C How to Program
C# How to Program
C++ How to Program
VB How to Program

shared_ptr with a custom deleter.
Custom deleter for a Book, Destroying Book: Small C++ How to Program

All shared_ptr objects are going out of scope.
Destroying Book: C How to Program
Destroying Book: C# How to Program
Destroying Book: C++ How to Program
Destroying Book: VB How to Program
Destroying Book: Java How to Program
```

Fig. 24.3 | `shared_ptr` example program. (Part 3 of 3.)

Manipulating ***shared_ptr*****s**

Lines 29–30 display the `Book`'s `title` and the number of `shared_ptr`s referencing that instance. Notice that we use the `->` operator to access the `Book`'s data member `title`, as we would with a regular pointer. `shared_ptr`s provide the pointer operators `*` and `->`. We get the reference count using the `shared_ptr` member function **`use_count`**, which returns the number of `shared_ptr`s to the resource. Then we create another `shared_ptr` to the instance of class `Book` (line 33). Here we use the `shared_ptr` constructor with the original `shared_ptr` as its argument. You can also use the assignment operator (`=`) to create a `shared_ptr` to the same resource. Lines 34–35 print the reference count of the original `shared_ptr` to show that the count increased by one when we created the second `shared_ptr`. As mentioned earlier, changes made to the resource of a `shared_ptr` are "seen" by all `shared_ptr`s to that resource. When we change the title of the `Book` using `bookPtr2` (line 38), we can see the change when using `bookPtr` (lines 39–41).

Manipulating ***shared_ptr*****s** *in an STL Container*

Next we demonstrate using `shared_ptr`s in an STL container. We create a `vector` of `BookPtr`s (line 44) and add four elements (recall that `BookPtr` is a `typedef` for a `shared_ptr<Book>`, line 10). Lines 51–53 print the contents of the `vector`. Then we sort the `Book`s in the `vector` by title (line 56). We use the function `compareTitles` (lines 20–23) in the `sort` algorithm to compare the `title` data members of each `Book` alphabetically.

***shared_ptr* Custom Deleter**

Line 63 creates a `shared_ptr` with a custom deleter. We define the custom deleter function `deleteBook` (lines 13–17) and pass it to the `shared_ptr` constructor along with a pointer to a new instance of class `Book`. When the `shared_ptr` destroys the instance of class `Book`, it calls `deleteBook` with the internal `Book *` as the argument. Notice that `deleteBook` takes a `Book *`, not a `shared_ptr`. A custom deleter function must take one argument of the `shared_ptr`'s internal pointer type. `deleteBook` displays a message to show that the custom deleter was called, then deletes the pointer. A primary use for custom deleters is when using third-party C libraries. Rather than providing a class with a constructor and destructor as a C++ library would, C libraries frequently provide one function that returns a pointer to a `struct` representing a resource and another that does the necessary cleanup when the resource is no longer needed. Using a custom deleter allows you to use a `shared_ptr` to keep track of the resource and still ensure it is freed correctly.

Resetting a shared_ptr

We call the `shared_ptr` member function `reset` (line 64) to show the custom deleter at work. The **`reset`** function releases the current resource and sets the `shared_ptr` to `NULL`. If there are no other `shared_ptr`s to the resource, it's destroyed. You can also pass a pointer or `shared_ptr` representing a new resource to the `reset` function, in which case the `shared_ptr` will manage the new resource. But, as with the constructor, you should only use a regular pointer returned by the `new` operator.

***shared_ptr*s Are Destroyed When They Go Out of Scope**

All the `shared_ptr`s and the `vector` go out of scope at the end of the `main` function and are destroyed. When the `vector` is destroyed, so are the `shared_ptr`s in it. The program output shows that each instance of class `Book` is destroyed automatically by the `shared_ptr`s. There is no need to `delete` each pointer placed in the `vector`.

24.2.2 weak_ptr: shared_ptr Observer

A `weak_ptr` points to the resource managed by a `shared_ptr` without assuming any responsibility for it. The reference count for a `shared_ptr` doesn't increase when a `weak_ptr` references it. That means that the resource of a `shared_ptr` can be deleted while there are still `weak_ptr`s pointing to it. When the last `shared_ptr` is destroyed, the resource is deleted and any remaining `weak_ptr`s are set to `NULL`. One use for `weak_ptr`s, as we'll demonstrate later in this section, is to avoid *memory leaks* caused by *circular references*.

A `weak_ptr` can't directly access the resource it points to—you must create a `shared_ptr` from the `weak_ptr` to access the resource. There are two ways to do this. You can pass the `weak_ptr` to the `shared_ptr` constructor. That creates a `shared_ptr` to the resource being pointed to by the `weak_ptr` and properly increases the reference count. If the resource has already been deleted, the `shared_ptr` constructor will throw a **`bad_weak_ptr`** exception. You can also call the `weak_ptr` member function **`lock`**, which returns a `shared_ptr` to the `weak_ptr`'s resource. If the `weak_ptr` points to a deleted resource (i.e., `NULL`), `lock` will return an empty `shared_ptr` (i.e., a `shared_ptr` to `NULL`). `lock` should be used when an empty `shared_ptr` isn't considered an error. You can access the resource once you have a `shared_ptr` to it. `weak_ptr`s should be used in any situation where you need to *observe* the resource but don't want to assume any management respon-

sibilities for it. The following example demonstrates the use of `weak_ptr`s in **circularly referential data**, a situation in which two objects refer to each other internally.

Example Using ***`weak_ptr`***

Figure 24.4–24.7 define classes `Author` and `Book`. Each class has a pointer to an instance of the other class. This creates a circular reference between the two classes. Note that we use both `weak_ptr`s and `shared_ptr`s to hold the cross reference to each class (Fig. 24.4 and 24.5, lines 20–21 in each figure). If we set the `shared_ptr`s, it creates a *memory leak*—we'll explain why soon and show how we can use the `weak_ptr`s to fix this problem.

```
1  // Fig. 24.4: Author.h
2  // Definition of class Author.
3  #ifndef AUTHOR_H
4  #define AUTHOR_H
5  #include <string>
6  #include <memory>
7
8  using namespace std;
9
10 class Book; // forward declaration of class Book
11
12 // Author class definition
13 class Author
14 {
15 public:
16    Author( const string &authorName ); // constructor
17    ~Author(); // destructor
18    void printBookTitle(); // print the title of the Book
19    string name; // name of the Author
20    weak_ptr< Book > weakBookPtr; // Book the Author wrote
21    shared_ptr< Book > sharedBookPtr; // Book the Author wrote
22 };
23 #endif // AUTHOR_H
```

Fig. 24.4 | `Author` class definition.

```
1  // Fig. 24.5: Book.h
2  // Definition of class Book.
3  #ifndef BOOK_H
4  #define BOOK_H
5  #include <string>
6  #include <memory>
7
8  using namespace std;
9
10 class Author; // forward declaration of class Author
11
12 // Book class definition
13 class Book
14 {
```

Fig. 24.5 | `Book` class definition. (Part 1 of 2.)

```
15  public:
16     Book( const string &bookTitle ); // constructor
17     ~Book(); // destructor
18     void printAuthorName(); // print the name of the Author
19     string title; // title of the Book
20     weak_ptr< Author > weakAuthorPtr; // Author of the Book
21     shared_ptr< Author > sharedAuthorPtr; // Author of the Book
22  };
23  #endif // BOOK_H
```

Fig. 24.5 | `Book` class definition. (Part 2 of 2.)

Classes `Author` and `Book` define destructors that each display a message to indicate when an instance of either class is destroyed (Fig. 24.6 and 24.7, lines 15–18). Each class also defines a member function to print the `title` of the `Book` and `Author`'s `name` (lines 21–34 in each figure). Recall that you can't access the resource directly through a `weak_ptr`, so first we create a `shared_ptr` from the `weak_ptr` data member (line 24 in each figure). If the resource the `weak_ptr` is referencing doesn't exist, the call to the `lock` function returns a `shared_ptr` which points to `NULL` and the condition fails. Otherwise, the new `shared_ptr` contains a valid pointer to the `weak_ptr`'s resource, and we can access the resource. If the condition in line 24 is true (i.e., `bookPtr` and `authorPtr` aren't `NULL`), we print the reference count to show that it increased with the creation of the new `shared_ptr`, then we print the `title` of the `Book` and `Author`'s `name`. The `shared_ptr` is destroyed when the function exits so the reference count decreases by one.

```
1  // Fig. 24.6: Author.cpp
2  // Member-function definitions for class Author.
3  #include <iostream>
4  #include <string>
5  #include <memory>
6  #include "Author.h"
7  #include "Book.h"
8
9  using namespace std;
10
11 Author::Author( const string &authorName ) : name( authorName )
12 {
13 }
14
15 Author::~Author()
16 {
17    cout << "Destroying Author: " << name << endl;
18 } // end of destructor
19
20 // print the title of the Book this Author wrote
21 void Author::printBookTitle()
22 {
23    // if weakBookPtr.lock() returns a non-empty shared_ptr
24    if ( shared_ptr< Book > bookPtr = weakBookPtr.lock() )
25    {
```

Fig. 24.6 | `Author` member-function definitions. (Part 1 of 2.)

```
26         // show the reference count increase and print the Book's title
27         cout << "Reference count for Book " << bookPtr->title
28            << " is " << bookPtr.use_count() << "." << endl;
29         cout << "Author " << name << " wrote the book " << bookPtr->title
30            << "\n" << endl;
31      } // end if
32      else // weakBookPtr points to NULL
33         cout << "This Author has no Book." << endl;
34   } // end of printBookTitle
```

Fig. 24.6 | `Author` member-function definitions. (Part 2 of 2.)

```
1  // Fig. 24.7: Book.cpp
2  // Member-function definitions for class Book.
3  #include <iostream>
4  #include <string>
5  #include <memory>
6  #include "Author.h"
7  #include "Book.h"
8
9  using namespace std;
10
11 Book::Book( const string &bookTitle ) : title( bookTitle )
12 {
13 }
14
15 Book::~Book()
16 {
17    cout << "Destroying Book: " << title << endl;
18 } // end of destructor
19
20 // print the name of this Book's Author
21 void Book::printAuthorName()
22 {
23    // if weakAuthorPtr.lock() returns a non-empty shared_ptr
24    if ( shared_ptr< Author > authorPtr = weakAuthorPtr.lock() )
25    {
26       // show the reference count increase and print the Author's name
27       cout << "Reference count for Author " << authorPtr->name
28          << " is " << authorPtr.use_count() << "." << endl;
29       cout << "The book " << title << " was written by "
30          << authorPtr->name << "\n" << endl;
31    } // end if
32    else // weakAuthorPtr points to NULL
33       cout << "This Book has no Author." << endl;
34 } // end of printAuthorName
```

Fig. 24.7 | `Book` member-function definitions.

Figure 24.8 defines a `main` function that demonstrates the *memory leak* caused by the *circular reference* between classes `Author` and `Book`. Lines 12–13 create `shared_ptr`s to an instance of each class. The `weak_ptr` data members are set in lines 16–17. Lines 20–21 set the `shared_ptr` data members for each class. The instances of classes `Author` and `Book` now

reference each other. We then print the reference count for the `shared_ptr`s to show that each instance is referenced by two `shared_ptr`s (lines 24–27), the ones we create in the `main` function and the data member of each instance. Remember that *`weak_ptr`s don't affect the reference count*. Then we call each class's member function to print the information stored in the `weak_ptr` data member (lines 32–33). The functions also display the fact that another `shared_ptr` was created during the function call. Finally, we print the reference counts again to show that the additional `shared_ptr`s created in the `printAuthorName` and `printBookTitle` member functions are destroyed when the functions finish.

```
1  // Fig. 24.8: fig24_08.cpp
2  // Demonstrate use of weak_ptr.
3  #include <iostream>
4  #include <memory>
5  #include "Author.h"
6  #include "Book.h"
7  using namespace std;
8
9  int main()
10 {
11    // create a Book and an Author
12    shared_ptr< Book > bookPtr( new Book( "C++ How to Program" ) );
13    shared_ptr< Author > authorPtr( new Author( "Deitel & Deitel" ) );
14
15    // reference the Book and Author to each other
16    bookPtr->weakAuthorPtr = authorPtr;
17    authorPtr->weakBookPtr = bookPtr;
18
19    // set the shared_ptr data members to create the memory leak
20    bookPtr->sharedAuthorPtr = authorPtr;
21    authorPtr->sharedBookPtr = bookPtr;
22
23    // reference count for bookPtr and authorPtr is one
24    cout << "Reference count for Book " << bookPtr->title << " is "
25       << bookPtr.use_count() << endl;
26    cout << "Reference count for Author " << authorPtr->name << " is "
27       << authorPtr.use_count() << "\n" << endl;
28
29    // access the cross references to print the data they point to
30    cout << "\nAccess the Author's name and the Book's title through "
31       << "weak_ptrs." << endl;
32    bookPtr->printAuthorName();
33    authorPtr->printBookTitle();
34
35    // reference count for each shared_ptr is back to one
36    cout << "Reference count for Book " << bookPtr->title << " is "
37       << bookPtr.use_count() << endl;
38    cout << "Reference count for Author " << authorPtr->name << " is "
39       << authorPtr.use_count() << "\n" << endl;
40
41    // the shared_ptrs go out of scope, the Book and Author are destroyed
42    cout << "The shared_ptrs are going out of scope." << endl;
43 } // end of main
```

Fig. 24.8 | `shared_ptr`s cause a memory leak in circularly referential data. (Part 1 of 2.)

```
Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

Access the Author's name and the Book's title through weak_ptrs.
Reference count for Author Deitel & Deitel is 3.
The book C++ How to Program was written by Deitel & Deitel

Reference count for Book C++ How to Program is 3.
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 2
Reference count for Author Deitel & Deitel is 2

The shared_ptrs are going out of scope.
```

Fig. 24.8 | `shared_ptr`s cause a memory leak in circularly referential data. (Part 2 of 2.)

Memory Leak

At the end of `main`, the `shared_ptr`s to the instances of `Author` and `Book` we created go out of scope and are destroyed. Notice that the output doesn't show the destructors for classes `Author` and `Book`. The program has a *memory leak*—the instances of `Author` and `Book` aren't destroyed because of the `shared_ptr` data members. When `bookPtr` is destroyed at the end of the `main` function, the reference count for the instance of class `Book` becomes one—the instance of `Author` still has a `shared_ptr` to the instance of `Book`, so it isn't deleted. When `authorPtr` goes out of scope and is destroyed, the reference count for the instance of class `Author` also becomes one—the instance of `Book` still has a `shared_ptr` to the instance of `Author`. Neither instance is deleted because the reference count for each is still one.

Fixing the Memory Leak

Now, comment out lines 20–21 by placing `//` at the beginning of each line. This prevents the code from setting the `shared_ptr` data members for classes `Author` and `Book`. Recompile the code and run the program again. Figure 24.9 shows the output. Notice that the initial reference count for each instance is now one instead of two because we don't set the `shared_ptr` data members. The last two lines of the output show that the instances of classes `Author` and `Book` were destroyed at the end of the `main` function. We eliminated the *memory leak* by using the `weak_ptr` data members rather than the `shared_ptr` data members. The `weak_ptr`s don't affect the reference count but still allow us to access the resource when we need it by creating a temporary `shared_ptr` to the resource. When the `shared_ptr`s we created in `main` are destroyed, the *reference counts become zero* and the instances of classes `Author` and `Book` are deleted properly.

```
Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

Access the Author's name and the Book's title through weak_ptrs.
Reference count for Author Deitel & Deitel is 2.
The book C++ How to Program was written by Deitel & Deitel
```

Fig. 24.9 | `weak_ptr`s used to prevent a memory leak in circularly referential data. (Part 1 of 2.)

```
Reference count for Book C++ How to Program is 2.
Author Deitel & Deitel wrote the book C++ How to Program

Reference count for Book C++ How to Program is 1
Reference count for Author Deitel & Deitel is 1

The shared_ptrs are going out of scope.
Destroying Author: Deitel & Deitel
Destroying Book: C++ How to Program
```

Fig. 24.9 | `weak_ptr`s used to prevent a memory leak in circularly referential data. (Part 2 of 2.)

24.3 Multithreading

Multithreading is one of the most significant updates in the C++11 standard. Though multithreading has been around for decades, interest in it is rising quickly due to the proliferation of *multicore* systems—even today's smartphones and tablets are typically multicore. The most common level of multicore processor today is *dual core*, though *quad core* processors are becoming popular. The number of cores will continue to grow. In multicore systems, the hardware can put multiple processors to work *simultaneously* on different parts of your task, thereby enabling the program to complete faster. To take full advantage of multicore architecture you need to write multithreaded applications. When a program splits tasks into separate threads, a multicore system can run those threads *in parallel*. This section introduces basic multithreading features that enable you to execute functions in separate threads. At the end of the section we provide links to online references where you can learn more about C++11 multithreading.

24.3.1 Multithreading Headers in C++11

Previously, C++ multithreading libraries were non-standard, platform-specific extensions. You'll often want your code to be portable across platforms. This is a key benefit of standardized multithreading. C++11 provides several headers that declare the new multithreading capabilities for writing more portable multithreaded C++ code:

- **`<thread>` header**—Contains class `thread` for manually creating and starting threads, and functions `yield`, `get_id`, `sleep_for` and `sleep_until`.
- **`<mutex>` header**—Contains classes and class templates for ensuring *mutually exclusive* access to resources shared among threads in an application—also known as *thread synchronization*.
- **`<condition_variable>` header**—Contains classes, a function and an `enum` that are used together with facilities in header `<mutex>` to implement thread synchronization. In particular, *condition variables* can be used to make threads *wait* for a specific condition in a program, then to *notify* the waiting threads when that condition is satisfied.
- **`<future>` header**—Contains class templates, a function template and `enum`s that enable you specify functions to execute in separate threads and to receive the results of those functions when the threads complete.

For the examples in this section, we used the Microsoft's Visual C++ and Apple's Xcode LLVM compilers. GNU's g++ compiler provides partial C++11 multithreading support.

24.3.2 Running Multithreaded Programs

When you run any program on a modern computer system, your program's tasks compete for the attention of the processor(s) with the operating system, other programs and other activities that the operating system is running on your behalf. All kinds of tasks are typically running in the background on your system. When you execute the examples in this section, the time to perform each calculation will vary based on your computer's processor speed, number of processor cores and what's running on your computer. It's not unlike a drive to the supermarket. The time it takes you to drive there can vary based on traffic conditions, weather and other factors. Some days the drive might take 10 minutes, but during rush hour or bad whether it could take longer. The same is true for executing applications on computer systems.

There's also overhead inherent in multithreading itself. Simply dividing a task into two threads and running it on a dual core system does not run it twice as fast, though it will typically run faster than performing the thread's tasks in sequence on one core. As you'll see, executing a multithreaded application on a single-core processor can actually take longer than simply performing the thread's tasks in sequence.

24.3.3 Overview of this Section's Examples

To provide a convincing demonstration of multithreading on a multicore system, this section presents two programs:

- The first performs two compute-intensive calculations sequentially (Fig. 24.10).
- The second executes the same compute-intensive calculations in parallel threads (Fig. 24.11).

We executed each program on single-core *and* dual-core Windows 7 computers to demonstrate the performance of each program in each scenario. We used features of the `<ctime>` header to time each calculation and the total calculation time in both programs. The program outputs show the time improvements when the multithreaded program executes on a *multicore* system.

24.3.4 Example: Sequential Execution of Two Compute-Intensive Tasks

Figure 24.10 uses the recursive `fibonacci` function (lines 42–53) that we introduced in Section 6.20. Recall that, for larger Fibonacci values, the recursive implementation can require *significant* computation time. The example sequentially performs the calculations `fibonacci(45)` (line 19) and `fibonacci(44)` (line 29). Before and after each `fibonacci` call, we capture the time so that we can calculate the total time required for the calculation. We also use this to calculate the total time required for both calculations. Lines 24, 34 and 38 use function `difftime` (from header `<ctime>`) to calculate the number of seconds between two times.

```
1  // Fig. 24.10: fibonacci.cpp
2  // Fibonacci calculations performed sequentially
3  #include <iostream>
4  #include <iomanip>
```

Fig. 24.10 | Fibonacci calculations performed sequentially. (Part 1 of 3.)

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

unsigned long long int fibonacci( unsigned int n ); // function prototype

// function main begins program execution
int main( void )
{
   cout << fixed << setprecision( 6 );
   cout << "Sequential calls to fibonacci(45) and fibonacci(44)" << endl;

   // calculate fibonacci value for number input by user
   cout << "Calculating fibonacci( 45 )" << endl;
   time_t startTime1 = time( nullptr );
   unsigned long long int result1 = fibonacci( 45 );
   time_t endTime1 = time( nullptr );

   cout << "fibonacci( 45 ) = " << result1 << endl;
   cout << "Calculation time = "
      << difftime( endTime1, startTime1 ) / 60.0
      << " minutes\n" << endl;

   cout << "Calculating fibonacci( 44 )" << endl;
   time_t startTime2 = time( nullptr );
   unsigned long long int result2 = fibonacci( 44 );
   time_t endTime2 = time( nullptr );

   cout << "fibonacci( 44 ) = " << result2 << endl;
   cout << "Calculation time = "
      << difftime( endTime2, startTime2 ) / 60.0
      << " minutes\n" << endl;

   cout << "Total calculation time = "
      << difftime( endTime2, startTime1 ) / 60.0 << " minutes" << endl;
} // end main

// Recursively calculates fibonacci numbers
unsigned long long int fibonacci( unsigned int n )
{
   // base case
   if ( 0 == n || 1 == n )
   {
      return n;
   } // end if
   else // recursive step
   {
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } // end else
} // end function fibonacci
```

Fig. 24.10 | Fibonacci calculations performed sequentially. (Part 2 of 3.)

a) Output on a Dual Core Windows 7 Computer

```
Sequential calls to fibonacci(45) and fibonacci(44)
Calculating fibonacci( 45 )
fibonacci( 45 ) = 1134903170
Calculation time = 1.416667 minutes

Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 0.866667 minutes

Total calculation time = 2.283333 minutes
```

b) Output on a Single Core Windows 7 Computer

```
Sequential calls to fibonacci(45) and fibonacci(44)
Calculating fibonacci( 45 )
fibonacci( 45 ) = 1134903170
Calculation time = 1.500000 minutes

Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 0.916667 minutes

Total calculation time = 2.416667 minutes
```

c) Output on a Single Core Windows 7 Computer

```
Sequential calls to fibonacci(45) and fibonacci(44)
Calculating fibonacci( 45 )
fibonacci( 45 ) = 1134903170
Calculation time = 1.466667 minutes

Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 0.900000 minutes

Total calculation time = 2.366667 minutes
```

Fig. 24.10 | Fibonacci calculations performed sequentially. (Part 3 of 3.)

The first output shows the results of executing the program on a dual-core Windows 7 computer. The second and third outputs show the results of executing the program on a single-core Windows 7 computer on which the program always took longer to execute (in our testing), because the processor was being shared between this program and all the others that happened to be executing on the computer at the same time.

24.3.5 Example: Multithreaded Execution of Two Compute-Intensive Tasks

Figure 24.11 also uses the recursive `fibonacci` function, but executes each call to `fibonacci` in a separate thread. The first two outputs show the multithreaded Fibonacci example executing on a *dual-core* computer. Though execution times varied, the total time to perform both Fibonacci calculations (in our tests) was always *less* than sequential execution in

Fig. 24.10. The last two outputs show the example executing on a single-core computer. Again, times varied for each execution, but the total time was *more* than the sequential execution due to the overhead of sharing *one* processor among all the program's threads and the other programs executing on the computer at the same time.

```
1  // Fig. 24.11: ThreadedFibonacci.cpp
2  // Fibonacci calculations performed in separate threads
3  #include <iostream>
4  #include <iomanip>
5  #include <future>
6  #include <ctime>
7  using namespace std;
8
9  // class to represent the results
10 class ThreadData
11 {
12 public:
13    time_t startTime; // time thread starts processing
14    time_t endTime; // time thread finishes processing
15 }; // end class ThreadData
16
17 unsigned long long int fibonacci( unsigned int n ); // function prototype
18 ThreadData startFibonacci( unsigned int n ); // function prototype
19
20 int main()
21 {
22    cout << fixed << setprecision( 6 );
23    cout << "Calculating fibonacci( 45 ) and fibonacci( 44 ) "
24       << "in separate threads" << endl;
25
26    cout << "Starting thread to calculate fibonacci( 45 )" << endl;
27    auto futureResult1 = async( launch::async, startFibonacci, 45 );
28    cout << "Starting thread to calculate fibonacci( 44 )" << endl;
29    auto futureResult2 = async( launch::async, startFibonacci, 44 );
30
31    // wait for results from each thread
32    ThreadData result1 = futureResult1.get();
33    ThreadData result2 = futureResult2.get();
34
35    // determine time that first thread started
36    time_t startTime = ( result1.startTime < result2.startTime ) ?
37       result1.startTime : result2.startTime;
38
39    // determine time that last thread terminated
40    time_t endTime = ( result1.endTime > result2.endTime ) ?
41       result1.endTime : result2.endTime;
42
43    // display total time for calculations
44    cout << "Total calculation time = "
45       << difftime( endTime, startTime ) / 60.0 << " minutes" << endl;
46 } // end main
```

Fig. 24.11 | Fibonacci calculations performed in separate threads. (Part 1 of 3.)

```
47
48  // executes function fibonacci asynchronously
49  ThreadData startFibonacci( unsigned int n )
50  {
51     // cast ptr to ThreadData * so we can access arguments
52     ThreadData result = { 0, 0 };
53
54     cout << "Calculating fibonacci( " << n << " )" << endl;
55     result.startTime = time( nullptr ); // time before calculation
56     auto fibonacciValue = fibonacci( n );
57     result.endTime = time( nullptr ); // time after calculation
58
59     // display fibonacci calculation result and total calculation time
60     cout << "fibonacci( " << n << " ) = " << fibonacciValue << endl;
61     cout << "Calculation time = "
62        << difftime( result.endTime, result.startTime ) / 60.0
63        << " minutes\n" << endl;
64     return result;
65  } // end function startFibonacci
66
67  // Recursively calculates fibonacci numbers
68  unsigned long long int fibonacci( unsigned int n )
69  {
70     // base case
71     if ( 0 == n || 1 == n )
72     {
73        return n;
74     } // end if
75     else // recursive step
76     {
77        return fibonacci( n - 1 ) + fibonacci( n - 2 );
78     } // end else
79  } // end function fibonacci
```

a) Output on a Dual Core Windows 7 Computer

```
Calculating fibonacci( 45 ) and fibonacci( 44 ) in separate threads
Starting thread to calculate fibonacci( 45 )
Starting thread to calculate fibonacci( 44 )
Calculating fibonacci( 45 )
Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 1.050000 minutes

fibonacci( 45 ) = 1134903170
Calculation time = 1.616667 minutes

Total calculation time = 1.616667 minutes
```

Fig. 24.11 | Fibonacci calculations performed in separate threads. (Part 2 of 3.)

b) Output on a Dual Core Windows 7 Computer

```
Calculating fibonacci( 45 ) and fibonacci( 44 ) in separate threads
Starting thread to calculate fibonacci( 45 )
Starting thread to calculate fibonacci( 44 )
Calculating fibonacci( 45 )
Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 1.016667 minutes

fibonacci( 45 ) = 1134903170
Calculation time = 1.583333 minutes

Total calculation time = 1.583333 minutes
```

c) Output on a Single Core Windows 7 Computer

```
Calculating fibonacci( 45 ) and fibonacci( 44 ) in separate threads
Starting thread to calculate fibonacci( 45 )
Starting thread to calculate fibonacci( 44 )
Calculating fibonacci( 45 )
Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 2.333333 minutes

fibonacci( 45 ) = 1134903170
Calculation time = 2.966667 minutes

Total calculation time = 2.966667 minutes
```

d) Output on a Single Core Windows 7 Computer

```
Calculating fibonacci( 45 ) and fibonacci( 44 ) in separate threads
Starting thread to calculate fibonacci( 45 )
Starting thread to calculate fibonacci( 44 )
Calculating fibonacci( 45 )
Calculating fibonacci( 44 )
fibonacci( 44 ) = 701408733
Calculation time = 2.233333 minutes

fibonacci( 45 ) = 1134903170
Calculation time = 2.850000 minutes

Total calculation time = 2.850000 minutes
```

Fig. 24.11 | Fibonacci calculations performed in separate threads. (Part 3 of 3.)

Class `ThreadData`

The function that each thread executes in this example returns an object of class `ThreadData` (lines 10–15) containing two `time_t` members. We use these to store the system time before and after each thread's call to the recursive function `fibonacci`.

Creating and Executing a Thread: Function template `async`

Lines 31–41 create two threads by calling function template `async` (lines 27 and 29), which has two overloaded versions. The version used here takes three arguments:

- Thread *launch policy*: This is a value from the `launch` `enum`—either `launch::async`, `launch::deferred` or both separated by a bitwise OR (`|`) operator. The value `launch::async` indicates that the function specified in the second argument should execute in a separate thread. The value `launch::deferred` indicates that the function specified in the second argument should execute in the same thread when the program uses the `future` object returned by function template `async` to get the result.
- *Function pointer* or *function object* to execute: This specifies the task to perform in the thread.
- Arguments: The third argument can actually be any number of additional arguments that are passed to the function or function object specified by the second argument. In this example, we pass one additional argument—the `unsigned int` that should in turn be passed from function `startFibonacci` to function `fibonacci` to perform the calculation.

The other version of `async` does not take the launch policy argument. Instead, it determines for you whether to execute synchronously or asynchronously. If `async` cannot create the thread, a `system_error` exception occurs. If the thread is created successfully, the function specified as the second argument begins executing in the new thread.

Joining the Threads: Class Template `future`

Function `async` returns an object of class template `future` that you can use when the thread completes execution to obtain data returned by the function that `async` executes—in our case, a `ThreadData` object. The type `future<ThreadData>` is inferred by the compiler from the return type of function `startFibonacci`—the second argument to function template `async`.

To ensure that the program does not terminate until the threads terminate, lines 32–33 call each `future`'s `get` member function. This causes the program to *wait* until the corresponding threads complete execution—known as *joining the threads*—before executing the remaining code in `main`. Function `get` implicitly calls the underlying thread's `join` member function. When the thread completes, `get` returns whatever the function executed by `async` returns—again, a `ThreadData` object in this example.

Function `startFibonacci`

Function `startFibonacci` (lines 49–65) specifies the task to perform—in this case, to call `fibonacci` (line 56) to recursively perform a calculation, to time the calculation (lines 55 and 57, to display the calculation's result (line 60) and to display the time the calculation took (lines 61–63). Each thread in this example executes until `startFibonacci` returns—at which point the thread terminates. When threads terminate function `main` can complete its execution.

Web Resources for Multithreading

For more information on C++11 multithreading see Section 30 of the C++ standard document. In addition, see the following online articles and blog posts:

http://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/
Tutorial: "C++11 Multithreading."
http://solarianprogrammer.com/2012/02/27/cpp-11-thread-tutorial-part-2/
Tutorial: "C++11 Multithreading, Part 2."
http://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/
Tutorial: "C++11 Multithreading, Part 3."
http://www.informit.com/articles/article.aspx?p=1750198
Article: "What You Need to Know About C++11 Multicore Multithreading," by Stephen B. Morris.
http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-8-futures-and-promises.html
Blog: "Multithreading in C++0x Part 8, Futures, Promises and Asynchronous Function Calls."
http://marknelson.us/2012/05/23/c11-threading-made-easy/
Blog: "C++11-Threading Made Easy."
http://dclong.github.com/en/2012/06/cpp11-concurrency-tips/
Blog: "Tips for Multithreading/Concurrency Programming in C++11."
http://bartoszmilewski.com/category/multithreading/
Blog: "Multithreading: The Future of C++ Concurrency and Parallelism."

24.4 noexcept Exception Specifications and the noexcept Operator

Prior to C++11, you could place an *exception specification* after function definition's parameter list to specify the types of exceptions potentially thrown by that function. This specification consisted of the `throw` keyword followed by a list of types in parentheses. This feature is now *deprecated* and was not supported by many C++ compilers previously.

noexcept Specification

In C++11, you can now declare simply whether or not a function throws *any* exceptions. If the compiler knows that a function does not throw *any* exceptions, it can potentially perform additional *optimizations*. In the following function

```
int square( int value ) noexcept
{
   return value * value;
}
```

the **`noexcept`** keyword to the right of the function's parameter list indicates that this function *does not throw exceptions*. The `noexcept` specification can optionally be followed by parentheses containing a constant expression that evaluates to `true` or `false`. In the preceding function, `noexcept` is equivalent to `noexcept(true)`; if the expression's value is `false`, the function might throw exceptions.

noexcept Operator

In function templates, the expression in a `noexcept` specification typically uses the operator **`noexcept`**, which has the form

```
noexcept( expression )
```

and returns `false` if the *expression* can throw an exception. This enables the compiler to generate function template specializations with the specification `noexcept(true)` for

some types and `noexcept(false)` for others. Typically, the *expression* is a function call that uses values of the type(s) for which the function template was instantiated.

Web Resources for **`noexcept`**

`http://en.cppreference.com/w/cpp/language/noexcept`
`http://en.cppreference.com/w/cpp/language/noexcept_spec`
The C++11 `noexcept` operator and `noexcept` specification pages on `cppreference.com`.

`http://akrzemi1.wordpress.com/2011/06/10/using-noexcept/`
The blog, "Using `noexcept`," by Andrzej Krzemienski. Includes sample code.

`http://stackoverflow.com/questions/12833241/difference-between-c03-throw-specifier-c11-noexcept`
The Stackoverflow discussion, "Difference between C++03 `throw()` specifier C++11 `noexcept`."

`http://stackoverflow.com/questions/2762078/why-is-c0xs-noexcept-checked-dynamically`
The Stackoverflow discussion, "Why is C++0x's `noexcept` checked dynamically?"

`http://stackoverflow.com/questions/10787766/when-should-i-really-use-noexcept`
The Stackoverflow discussion, "When Should I Really Use `noexcept`?"

`http://cpptruths.blogspot.com/2011/09/tale-of-noexcept-swap-for-user-defined.html`
The bog, "A tale of noexcept swap for user-defined classes in C++11," by Sumant Tambe. Discusses how C++ is phasing out `throw` in favor of the new `noexcept`.

24.5 Move Semantics

11

There are many cases in which C++ makes *copies* of objects. For example, each time you pass an object to a function *by value* or return an object from a function *by value* a copy of the object is made. There are many cases in which the object being copied *is about to be destroyed*, such as a temporary object that was returned from a function *by value* or a local object that's going out of scope. In such cases, it's more efficient to *move* the contents of the object that's about to be destroyed into the destination object, thus avoiding any copying overhead.

For example, consider a function that creates a local `string` object, then returns a copy of that object:

```
string createString( string &name )
{
   return string( "Hello " ) + name;
} // end function create string
```

The `string` created in the `return` statement is a *temporary* `string` that will be *copied* and passed back to the caller. In the following statement:

```
string result = createString( "Sam" );
```

the *temporary* `string` containing `"Hello Sam"` is copied into the `string` variable `result`, then the *temporary* `string` containing `"Hello Sam"` is *destroyed*. Since the temporary object is being destroyed, it would be more efficient to *move* the temporary object's resources into the `string` `result`.

11

C++11 introduces *rvalue references* and *move semantics* to help eliminate unnecessary copying of objects in many cases. In this section, we use these new capabilities in the context of our `Array` case study from Chapter 10.

The Rule of Three is Now the Rule of Five

Recall from Chapter 10 that a *copy constructor*, a *destructor* and an *overloaded assignment operator* are usually provided as a group for any class that uses dynamically allocated memory. This is sometimes referred to as the *Rule of Three*. With the addition of *move semantics* in C++11, you should also provide a *move constructor* and a *move assignment operator*. For this reason, the *Rule of Three* is now known as the *Rule of Five*.

24.5.1 *rvalue* references

An rvalue typically represents a *temporary object*, such as the result of a calculation, an object that's implicitly created or an object that's returned from a function by value. C++11's new ***rvalue* reference** type allows a reference to refer to an *rvalue* rather than an *lvalue*. An *rvalue* reference is declared as `T&&` (where `T` is the type of the object being referenced) to distinguish it from a normal reference `T&` (called an *lvalue* reference). An *rvalue* reference is used to implement *move semantics*—instead of being *copied*, the object's state (i.e., its content) is *moved*, leaving the original in a state that can be properly destructed. For example, prior to C++11 the following code created a *temporary* `string` object and passed it to `push_back`, which then copied it into the `vector`:

```
vector< string > myVector;
myVector.push_back( "message" );
```

As of C++11, member function `push_back` is now overloaded with a version that takes an *rvalue* reference. This allows the preceding call to `push_back` to take the storage allocated for the temporary `string` and *reuse* it directly for the new element in the `vector`. The temporary `string` will be destroyed when the function returns, so there's no need for it to keep its content.

Web Resources for rvalue References

`http://thbecker.net/articles/rvalue_references/section_01.html`
Article: "C++ rvalue References Explained," by Thomas Becker. Discusses move semantics, rvalue references, forcing move semantics, are rvalue references an rvalue, compiler optimization, perfect forwarding the problem and solution, rvalue references and exceptions and the implicit move.

`http://www.cprogramming.com/c++11/rvalue-references-and-move-semantics-in-c++11.html`
Article: "Move Semantics and rvalue References in C++11," by Alex Allain. Discusses `rvalues` and `lvalues`, detecting temporary objects with rvalue references, the move constructor and move assignment operator, the `std::move`, returning an explicit `rvalue`-reference from a function, move semantics and the standard library and move semantics and `rvalue` reference compiler support.

`http://www.codesynthesis.com/~boris/blog/2012/03/06/rvalue-reference-pitfalls/`
Blog: "`Rvalue` Reference Pitfalls."

`http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Scott-Meyers-Universal-References-in-Cpp11`
Video: "C++ and Beyond 2012: Scott Myers-Universal References in C++11." A 1 and ½ hour presentation.

`http://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c/`
Blog: "Understanding `lvalues` and `rvalues` in C and C++," by Eli Bendersky. Discusses a simple definition, basic examples, modifiable `lvalues`, conversions between `lvalues` and `rvalues`, qualified `rvalues`, and `rvalue` references.

http://en.cppreference.com/w/cpp/utility/move
Reference: "std::move."

http://blogs.msdn.com/b/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx
Blog: "Rvalue References: C++0x Features in VC10, Part 2," from the Visual C++ Team. Discusses lvalues and rvalues, the copying problem, rvalue reference initialization, rvalue references overload resolution, move semantics, moving from lvalues, movable members, the forwarding problem, perfect forwarding, template argument deduction and reference collapsing, the past and the future.

http://www.slideshare.net/goldshtn/c11-12147261
Slide Presentation: "C++11 Standard," from BrightSource. A brief presentation on the C++11 standard, including lambda functions, rvalue references, automatic variables, and more.

http://www.codesynthesis.com/~boris/blog/2012/03/14/rvalue-reference-pitfalls-update/
Blog: "Rvalue Pitfalls, An Update."

http://www.codesynthesis.com/~boris/blog/2008/11/24/rvalue-reference-basics/
Blog: "Rvalue References: The Basics."

http://www.lapthorn.net/archives/800
Blog: "C++11 Part 3: Rvalue References."

24.5.2 Adding Move Capabilities to Class Array

We'll now update Section 10.10's Array class with a *move constructor* and a *move assignment* operator and demonstrate when they are used. Figures 24.12–24.13 show the new implementation of our Array class. In Fig. 24.12, we added prototypes for the *move constructor* (line 16) and *move assignment operator* (line 21). Figure 24.13 contains the implementations of these new member functions (in lines 36–45 and 84–100, respectively), which we discuss momentarily. We added output statements to the constructors, assignment operators and destructor to show in the program output (Fig. 24.14) when each is called. We also modified the overloaded << operator (Fig. 24.13, lines 149–157) to display an Array's elements separated by one space each. The other features of class Array are identical to Section 10.10's Array class, so we do not discuss them here.

```
1  // Fig. 24.12: Array.h
2  // Array class definition with overloaded operators.
3  #ifndef ARRAY_H
4  #define ARRAY_H
5
6  #include <iostream>
7
8  class Array
9  {
10    friend std::ostream &operator<<( std::ostream &, const Array & );
11    friend std::istream &operator>>( std::istream &, Array & );
12
13 public:
14    explicit Array( size_t = 10 ); // default constructor
15    Array( const Array & ); // copy constructor
16    Array( Array && ) noexcept; // move constructor
```

Fig. 24.12 | Array class header. (Part 1 of 2.)

```
17     ~Array(); // destructor
18     size_t getSize() const; // return size
19
20     Array &operator=( const Array & ); // copy assignment operator
21     Array &operator=( Array && ) noexcept; // move assignment operator
22     bool operator==( const Array & ) const; // equality operator
23
24     // inequality operator; returns opposite of == operator
25     bool operator!=( const Array &right ) const
26     {
27        return ! ( *this == right ); // invokes Array::operator==
28     } // end function operator!=
29
30     // subscript operator for non-const objects returns modifiable lvalue
31     int &operator[]( size_t );
32
33     // subscript operator for const objects returns rvalue
34     int operator[]( size_t ) const;
35  private:
36     size_t size; // pointer-based array size
37     int *ptr; // pointer to first element of pointer-based array
38  }; // end class Array
39
40  #endif
```

Fig. 24.12 | `Array` class header. (Part 2 of 2.)

```
1  // Fig. 24.13: Array.cpp
2  // Array class member- and friend-function definitions.
3  #include <iostream>
4  #include <iomanip>
5  #include <stdexcept>
6  #include <utility> // contains std::move
7  #include "Array.h" // Array class definition
8
9  using namespace std;
10
11 // default constructor for class Array (default size 10)
12 Array::Array( size_t arraySize )
13    : size( arraySize ),
14      ptr( new int[ size ] )
15 {
16    cout << "Array(int) constructor called" << endl;
17
18    for ( size_t i = 0; i < size; ++i )
19       ptr[ i ] = 0; // set pointer-based array element
20 } // end Array default constructor
21
```

Fig. 24.13 | `Array` class implementation. (Part 1 of 4.)

```
// copy constructor for class Array;
// must receive a reference to an Array
Array::Array( const Array &arrayToCopy )
   : size( arrayToCopy.size ),
     ptr( new int[ size ] )
{
   cout << "Array copy constructor called" << endl;

   for ( size_t i = 0; i < size; ++i )
      ptr[ i ] = arrayToCopy.ptr[ i ]; // copy into object
} // end Array copy constructor

// move constructor for class Array;
// must receive an rvalue reference to an Array
Array::Array( Array &&arrayToMove ) noexcept
   : size( arrayToMove.size ), // move arrayToMove's size to new Array
     ptr( arrayToMove.ptr ) // move arrayToMove's ptr to new Array
{
   cout << "Array move constructor called" << endl;

   // indicate that arrayToMove is now empty
   arrayToMove.size = 0;
   arrayToMove.ptr = nullptr;
} // end Array copy constructor

// destructor for class Array
Array::~Array()
{
   cout << "Array destructor called" << endl;
   delete [] ptr; // release pointer-based array space
} // end destructor

// return number of elements of Array
size_t Array::getSize() const
{
   return size; // number of elements in Array
} // end function getSize

// copy assignment operator
Array &Array::operator=( const Array &right )
{
   cout << "Array copy assignment operator called" << endl;

   if ( &right != this ) // avoid self-assignment
   {
      // for Arrays of different sizes, deallocate original
      // left-side Array, then allocate new left-side Array
      if ( size != right.size )
      {
         delete [] ptr; // release space
         size = right.size; // resize this object
         ptr = new int[ size ]; // create space for Array copy
      } // end inner if
```

Fig. 24.13 | `Array` class implementation. (Part 2 of 4.)

```
75
76          for ( size_t i = 0; i < size; ++i )
77             ptr[ i ] = right.ptr[ i ]; // copy array into object
78       } // end outer if
79
80       return *this; // enables x = y = z, for example
81    } // end copy assignment operator=
82
83    // move assignment operator;
84    Array &Array::operator=( Array &&arrayToMove ) noexcept
85    {
86       cout << "Array move assignment operator called" << endl;
87
88       if ( &arrayToMove != this ) // avoid self-assignment
89       {
90          delete [] ptr; // release space
91          size = arrayToMove.size; // move arrayToMove's size to new Array
92          ptr = arrayToMove.ptr; // move arrayToMove's ptr to new Array
93
94          // indicate that arrayToMove is now empty
95          arrayToMove.size = 0;
96          arrayToMove.ptr = nullptr;
97       } // end outer if
98
99       return *this; // enables x = y = z, for example
100   } // end move assignment operator=
101
102   // determine if two Arrays are equal and
103   // return true, otherwise return false
104   bool Array::operator==( const Array &right ) const
105   {
106      if ( size != right.size )
107         return false; // arrays of different number of elements
108
109      for ( size_t i = 0; i < size; ++i )
110         if ( ptr[ i ] != right.ptr[ i ] )
111            return false; // Array contents are not equal
112
113      return true; // Arrays are equal
114   } // end function operator==
115
116   // overloaded subscript operator for non-const Arrays;
117   // reference return creates a modifiable lvalue
118   int &Array::operator[]( size_t subscript )
119   {
120      // check for subscript out-of-range error
121      if ( subscript >= size )
122         throw out_of_range( "Subscript out of range" );
123
124      return ptr[ subscript ]; // reference return
125   } // end function operator[]
126
```

Fig. 24.13 | `Array` class implementation. (Part 3 of 4.)

```
127 // overloaded subscript operator for const Arrays
128 // const reference return creates an rvalue
129 int Array::operator[]( size_t subscript ) const
130 {
131    // check for subscript out-of-range error
132    if ( subscript >= size )
133       throw out_of_range( "Subscript out of range" );
134
135    return ptr[ subscript ]; // returns copy of this element
136 } // end function operator[]
137
138 // overloaded input operator for class Array;
139 // inputs values for entire Array
140 istream &operator>>( istream &input, Array &a )
141 {
142    for ( size_t i = 0; i < a.size; ++i )
143       input >> a.ptr[ i ];
144
145    return input; // enables cin >> x >> y;
146 } // end function
147
148 // overloaded output operator for class Array
149 ostream &operator<<( ostream &output, const Array &a )
150 {
151    // output private ptr-based array
152    for ( size_t i = 0; i < a.size; ++i )
153       output << a.ptr[ i ] << " ";
154
155    output << endl;
156    return output; // enables cout << x << y;
157 } // end function operator<<
```

Fig. 24.13 | `Array` class implementation. (Part 4 of 4.)

Class ***Array****'s Move Constructor*

Lines 36–45 implement class `Array`'s *move constructor*. The parameter is declared to be an *rvalue* reference (`&&`) to indicate that the resources of the argument `Array` should be *moved* into the object being constructed. The member initializer list sets the `size` and `ptr` members in the object being constructed to the values of the corresponding members in the parameter, then the constructor body sets the parameter's `size` member to `0` and `ptr` member to `nullptr` to complete the move. The argument object should be left in a state that allows it to be properly destructed and should no longer refer to the contents that were moved to the new object.

Class ***Array****'s Move Assignment Operator*

Lines 84–100 implement class `Array`'s *move assignment operator*. As in the move constructor, the move assignment operator's parameter is an *rvalue* reference (`&&`) to indicate that the resources of the argument `Array` should be moved. If the operation is not a *self assignment* (line 88), then line 92 deletes the dynamic memory that was originally allocated to the object on the left side of the assignment. Next, lines 93–94 move the resources of the

right operand into the left operand. Then lines 169–170 set the right operand's `size` member to `0` and `ptr` member to `nullptr` to complete the move.

Move Operations Should Be `noexcept`

In general, *move constructors* and *move assignment operators* should *not* throw exceptions because they're simply *moving* resources, not allocating new ones. For this reason, both the *move constructor* and *move assignment operator* are declared `noexcept` in their prototypes and definitions. At the time of this writing, Microsoft Visual C++ did not yet support `noexcept`. To execute this example in Visual C++, simply remove the `noexcept` keyword.

Testing Class `Array` *with Move Semantics*

The program of Fig. 24.14 demonstrates class `Array`'s *constructors* and *assignment operators*. In the program's output, we highlighted key lines with bold text. The program begins by creating `Array integers1` with seven elements that are initialized to 0 by default (line 23). Lines 24–26 display the size and contents of `integers1`. Next, lines 29–31 read values into `integers1` and display its new values.

```
1  // Fig. 24.14: fig24_14.cpp
2  // Array class test program.
3  #include <iostream>
4  #include <stdexcept>
5  #include <utility> // for std::move
6  #include "Array.h"
7  using namespace std;
8
9  // function to return an Array by value
10 Array getArrayByValue()
11 {
12    Array localIntegers( 3 );
13    localIntegers[ 0 ] = 10;
14    localIntegers[ 1 ] = 20;
15    localIntegers[ 2 ] = 30;
16    return localIntegers; // return by value creates an rvalue
17 } // end function getArrayByValue
18
19 int main()
20 {
21    // create 7-element Array integers1 then print its size and contents
22    cout << "Create 7 element Array integers1" << endl;
23    Array integers1( 7 ); // seven-element Array
24    cout << "Size of Array integers1 is "
25       << integers1.getSize()
26       << "\nintegers1 contains: " << integers1;
27
28    // input and print integers1
29    cout << "\nEnter 7 integers:" << endl;
30    cin >> integers1;
31    cout << "\nAfter input integers1 contains: " << integers1;
32
```

Fig. 24.14 | Testing class `Array` with move semantics.

```
33     // create Array integers2 using integers1 as an
34     // initializer; print size and contents
35     cout << "\nCreate Array integers2 as a copy of integers1" << endl;
36     Array integers2( integers1 ); // invokes copy constructor
37
38     cout << "Size of Array integers2 is "
39        << integers2.getSize()
40        << "\nintegers2 contains: " << integers2;
41
42     // create Array integers3 using the contents of the Array
43     // returned by getArrayByValue; print size and contents
44     cout << "\nCreate Array integers3 and initialize it with the "
45        << "\nrvalue Array returned by getArrayByValue" << endl;
46     Array integers3( getArrayByValue() ); // invokes move constructor
47
48     cout << "Size of Array integers3 is "
49        << integers3.getSize()
50        << "\nintegers3 contains: " << integers3;
51
52     // convert integers3 to an rvalue reference with std::move and
53     // use the result to initialize Array integers4
54     cout << "\nCreate Array integers4 and initialize it with the "
55        << "\nrvalue reference returned by std::move" << endl;
56     Array integers4( std::move( integers3 ) ); // invokes move constructor
57
58     cout << "Size of Array integers4 is "
59        << integers4.getSize()
60        << "\nintegers4 contains: " << integers4;
61
62     cout << "After moving integers3 to integers4, size of integers3 is "
63        << integers3.getSize()
64        << "\nintegers3 contains: " << integers3;
65
66     // copy contents of integers4 into integers3
67     cout << "\nUse copy assignment to copy contents "
68        << "of integers4 into integers3" << endl;
69     integers3 = integers4; // invokes copy constructor
70
71     cout << "After assigning integers4 to integers3, "
72        << "\nsize of Array integers3 is "
73        << integers3.getSize()
74        << "\nintegers3 contains: " << integers3;
75
76     // move contents of integers4 into integers1
77     cout << "\nUse move assignment to move contents "
78        << "of integers4 into integers3" << endl;
79     integers1 = std::move( integers4 ); // invokes move constructor
80
81     cout << "Size of Array integers1 is "
82        << integers1.getSize()
83        << "\nintegers1 contains: " << integers1;
84
```

Fig. 24.14 | Testing class `Array` with move semantics.

```
85      cout << "After moving integers4 to integers1, size of integers4 is "
86         << integers4.getSize()
87         << "\nintegers4 contains: " << integers4;
88   } // end main
```

```
Create 7 element Array integers1
Array(int) constructor called
Size of Array integers1 is 7
integers1 contains: 0 0 0 0 0 0 0

Enter 7 integers:
1 2 3 4 5 6 7

After input integers1 contains: 1 2 3 4 5 6 7

Create Array integers2 as a copy of integers1
Array copy constructor called
Size of Array integers2 is 7
integers3 contains: 1 2 3 4 5 6 7

Create Array integers3 and initialize it with the
rvalue Array returned by getArrayByValue
Array(int) constructor called
Array move constructor called
Array destructor called
Size of Array integers3 is 3
integers4 contains: 10 20 30

Create Array integers4 and initialize it with the
rvalue reference returned by std::move
Array move constructor called
Size of Array integers4 is 3
integers5 contains: 10 20 30
After moving integers3 to integers4, size of integers3 is 0
integers3 contains:

Use copy assignment to copy contents of integers4 into integers3
Array copy assignment operator called
After assigning integers4 to integers3,
size of Array integers3 is 3
integers3 contains: 10 20 30

Use move assignment to move contents of integers4 into integers3
Array move assignment operator called
Size of Array integers1 is 3
integers1 contains: 10 20 30
After moving integers4 to integers1, size of integers4 is 0
integers4 contains:
Array destructor called
Array destructor called
Array destructor called
Array destructor called
```

Fig. 24.14 | Testing class `Array` with move semantics.

Creating `Array integers2` *as a Copy of* `integers1`

Lines 35–40 use class `Array`'s *copy constructor* to initialize `Array` `integers2` then display the size and contents of the new `Array`. When a class contains both a *copy constructor* and a *move constructor*, the compiler decides which one to use based on the context. Line 36

```
Array integers2( integers1 ); // invokes copy constructor
```

uses class `Array`'s *copy constructor* because `integers1` is an *lvalue*, which cannot be passed to a constructor or function that receives an *rvalue* reference without first explicitly converting the *lvalue* to an *rvalue* reference first.

Creating `Array integers3` *and Initializing It With the* `Array` *Returned By Function* `getArrayByValue`

Lines 44–50 use class `Array`'s *move constructor* to initialize `Array` `integers3` then display the size and contents of the new `Array`. Line 46

```
Array integers3( getArrayByValue() ); // invokes move constructor
```

calls function `getArrayByValue` (lines 10–17), which creates an `Array` then returns it *by value*. When you return an object from a function by value, a *copy* of the object is made. That copy is an *rvalue* that exists only until the statement that called the function completes execution. In this context, the *rvalue* is more precisely known as an ***xvalue*** (*expiring value*), because the compiler knows that the object being returned from `getArrayByValue` is about to be *destroyed*. In this case, the compiler invokes class `Array`'s *move constructor* to *move* the contents of the `Array` returned by `getArrayByValue` into the `Array` being initialized, thus eliminating the overhead of *copying* the returned object. Similarly, if you were assigning the result of `getArrayByValue` to an existing `Array` object, the *move assignment operator* would be called.

Creating `Array integers4` *and Initializing It With the* rvalue *Returned By Function* `std::move`

Lines 54–64 use class `Array`'s *move constructor* to initialize `Array` `integers4` then display the size and contents of the new `Array`. Line 56

```
Array integers4( std::move(integers3) ); // invokes move constructor
```

uses the Standard Library function **`std::move`** (from header `<utility>`) to *explicitly convert* `integers3` (an *lvalue*) to an *rvalue* reference. This tells the compiler that `integers3`'s contents should be *moved* into `integers4` and invokes class `Array`'s *move constructor*. It's recommended that you use `std::move` in this manner *only* if you know the object will never be used again. For demonstration purposes, we output the size and contents of `integers3` again to show that the move constructor indeed moved `integers3`'s resources.

Assigning `Array integers4` *to* `integers3` *with the Copy Assignment Operator*

Lines 67–74 use class `Array`'s *copy assignment operator* to copy the contents of `Array` `integers4` into `integers3` then display the size and contents `integers3`. When a class contains both a *copy assignment operator* and a *move assignment operator*, the compiler decides which one to use based on the context. Line 69

```
integers3 = integers4; // invokes copy constructor
```

uses class `Array`'s *copy assignment operator* because `integers4` is an *lvalue*, which, as you just learned, cannot be passed to a constructor or function that receives an *rvalue* reference without explicitly converting the *lvalue* to an *rvalue* reference first.

Assigning `Array integers4` to `integers1` with the Move Assignment Operator
Lines 77–87 use class `Array`'s *move assignment operator* to move `integers4`'s contents into `integers1` then display the size and contents `integers1`. Line 79

```
integers1 = std::move( integers4 ); // invokes move constructor
```

uses the Standard Library function `std::move` to *explicitly convert* `integers4` (an *lvalue*) to an *rvalue* reference. This tells the compiler that `integers4`'s resources should be *moved* into `integers1` and invokes class `Array`'s *move assignment operator*. For demonstration purposes, we output the size and contents of `integers4` again to show that the *move assignment operator* indeed moved `integers4`'s resources.

Copy-and-Swap Assignment Operator Implementation
It's possible to define a *single* assignment operator that handles *both* copy and move assignment by using the so-called *copy-and-swap* technique. See the following article for an overview of copy-and-swap and a typical implementation:

```
http://bit.ly/CopyAndSwapCPP
```

24.5.3 `move` and `move_backward` Algorithms

You can use *move semantics* with ranges of elements stored in containers. The C++ Standard Library algorithms **`move`** and **`move_backward`** (from header `<algorithm>`) work like the `copy` and `copy_backward` algorithms (introduced in Section 15.5 and Section 16.3.8, respectively), but `move` and `move_backward` *move* the elements in the specified ranges rather than *copying* them.

Web Resources for the `move` and `move_backward` Algorithms
`http://en.cppreference.com/w/cpp/algorithm/move_backward`
Documentation: "Move Backward Algorithm."
`http://www.boost.org/doc/libs/1_51_0/doc/html/move/move_algorithms.html`
"C++ Move Algorithms," from Boost.
`http://en.cppreference.com/w/cpp/algorithm/move`
Documentation: `std::move`.

24.5.4 `emplace` Container Member Functions

When working with many of the C++ Standard Library containers that we introduced in Chapter 15, you can use the new member functions `emplace`, `emplace_front`, `emplace_back`, `emplace_after` and `emplace_hint` to insert objects into containers *without* invoking *any copy* or *move* operations. The `emplace` member functions contruct new objects *in place* in the new container elements.

Web Resources for the `emplace` Container Member Functions
`http://en.cppreference.com/w/cpp/container/vector/emplace`
Documentation: `std::vector::emplace`.

http://blog.haohaolee.com/blog/2012/03/11/til-what-emplace-is-in-c-plus-plus-11/
Blog: "What the Emplace Operation is in C++11."
http://www.boost.org/doc/libs/1_48_0/doc/html/container/move_emplace.html
Boost: "Efficient Insertion." Discusses move-aware containers and, emplace placement insertion.
http://www.nosid.org/cxx11-emplace-vs-insert.html
Blog: "C++11: Emplace vs. Insert."

24.6 static_assert

The **static_assert declaration** allows you to test constant integral expressions at *compile time* rather than runtime (with the assert macro; Section E.9). A static_assert declaration has the form

```
static_assert( integralConstantExpression, stringLiteral );
```

If the *integralConstantExpression* evaluates to *false*, the compiler reports an error message that includes the *stringLiteral*. The static_assert declaration can be used at namespace, class or block scope. static_assert is typically used by *library developers* to report incorrect usage of a library at compile time. For more details on static_assert see

```
http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=343
```

24.7 decltype

Operator **decltype** enables the compiler to determine an expression's type at compile time. If the expression is a function call, decltype determines the function's return type, which for a function template often changes based on the type(s) used to specialize the template.

The decltype operator is particularly useful when working with complex template types for which it's often difficult to provide, or even determine, the proper type declaration. Rather than trying to write a complex type declaration, for example, that represents the return type of a function, you can place in the parentheses of decltype an expression that returns the complex type and let the compiler "figure it out." This is used frequently in the C++11 Standard Library class templates.

The format of a decltype expression is

```
decltype( expression )
```

The expression is *not* evaluated. This is commonly used with trailing return types in function definitions (Section 6.18) that return complex types.

Web Resources for decltype

http://www.cprogramming.com/c++11/c++11-auto-decltype-return-value-after-function.html
The article, "Improved Type Inference in C++11: auto, decltype, and the new function declaration syntax," by Alex Allain. Includes code snippets.
http://en.wikipedia.org/wiki/Decltype
The Wikipedia entry for decltype. Discusses the motivation and semantics for decltype.
http://stackoverflow.com/questions/7623496/enlightening-usage-of-c11-decltype
The Stackoverflow discussion, "Enlightening Usage of C++11 decltype."
http://en.cppreference.com/w/cpp/language/decltype
The decltype specifier page on cppreference.com. Includes a small code example.

http://blogs.oracle.com/pcarlini/entry/c_11_tidbits_decltype_part
The article, "C++11 Tidbits: Decltype (Part 1)," by Paolo Carlini.
http://www.codesynthesis.com/~boris/blog/2012/08/14/using-cxx11-auto-decltype/
The article, "Using C++11 auto and decltype."
http://oopscenities.net/2011/05/04/c0x-decltype/
The blog, "C++11: decltype," by Ernesto Bascón Pantoja. Includes code snippets.

24.8 constexpr

As you know, you can declare a variable const to indicate that its value never changes. Such a variable is intialized at *compile time* when the initializer is itself a constant or literal. If the initializer for a const variable is a function call, then the initialization occurs at *runtime*. C++11 now includes keyword **constexpr** which can be used to declare variables, functions and constructors that are evaluated at compile time and result in a *constant*. A constexpr is implicitly const.

11

Consider class template numeric_limits (from header <limits>), which defines characteristics of the numeric types (char, int, double, etc.), such as their minimum and maximum values. For example, the member function

```
static constexpr T max() noexcept;
```

which returns a *literal value* representing the C++ implementation's maximum value for a numeric type. To store that value for an int in a variable, you could write

```
int maximum = numeric_limits<int>::max();
```

which calls the numeric_limits<int> specialization's max function. Prior to C++11, the max function call caused the preceding statement to be evaluated at runtime. In C++11, because this function simply returns a literal value, the preceding statement can be evaluated at compile time. This allows the compiler to perform additional optimizations and improves application performance because there's no runtime function-call overhead. See Section 5.19 of the C++ standard document (http://bit.ly/CPlusPlus11Standard) for the complete rules of defining constexpr variables, functions and constructors. Several of the web resources provided below also discuss these rules.

11

Web Resources for constexpr

http://blog.smartbear.com/software-quality/bid/248591/Using-constexpr-to-Improve-Security-Performance-and-Encapsulation-in-C
The blog, "Using constexpr to Improve Security, Performance and Encapsulation in C++," by Danny Kalev.
http://www.cprogramming.com/c++11/c++11-compile-time-processing-with-constexpr.html
The article, "Constexpr - Generalized Constant Expressions in C++11."
http://stackoverflow.com/questions/4748083/when-should-you-use-constexpr-capability-in-c11
The Stackoverflow discussion, "When should you use constexpr capability in C++11?"
http://cpptruths.blogspot.com/2011/07/want-speed-use-constexpr-meta.html
The blog, "Want speed? Use constexpr meta-programming!" by Sumant Tambe.
http://en.cppreference.com/w/cpp/language/constexpr
The constexpr specifier (since C++11) page on cppreference.com.

http://allanmcrae.com/2012/08/c11-part-7-constant-expressions/
The blog, "C++11—Part 7: Constant Expressions," by Allan McRae.
http://gist.github.com/1457531
The code example, constexpr_demo.cpp.
http://enki-tech.blogspot.com/2012/09/c11-compile-time-calculator-with.html
The blog, "C++11: Simple Compile-time Calculator With constexpr," by Thomas Badie.

24.9 Defaulted Special Member Functions

Recall that once you define *any* constructor in a class, the compiler does *not* generate a default constructor for that class. Prior to C++11, in any class that also needed a default constructor, you had to *explicitly* define a constructor with an empty parameter list or a constructor that could be called with no arguments—that is, all of its parameters had default arguments.

In C++11, you can tell the compiler to explicitly generate the default versions of the six special member functions—*default constructor*, *copy constructor*, *move constructor*, *copy assignment operator*, *move assignment operator* or *destructor*. To do so, you simply follow the special member function's prototype with `= default`. For example, in a class called `Employee` that has explicitly defined constructors, you can specify that the default constructor should be generated with the declaration

```
Employee() = default;
```

Web Resources for Defaulted Special Member Functions

http://blog.smartbear.com/software-quality/bid/167271/The-Biggest-Changes-in-C-11-and-Why-You-Should-Care
The article, "The Biggest Changes in C++11 (and Why You Should Care)," by Danny Kalev. Discusses deleted and defaulted functions.
http://stackoverflow.com/questions/6502828/c-default-keyword-classes-not-switch
The Stackoverflow discussion, "C++: Default keyword (classes, not switch)."
http://www.nullptr.me/2012/01/05/c11-defaulted-and-deleted-functions/
The blog, "C++11: defaulted and deleted functions," by Sarang Baheti.
http://stackoverflow.com/questions/7469468/defaulted-default-constructor-in-n3290-draft
The Stackoverflow discussion, "Defaulted default constructor."
http://stackoverflow.com/questions/13412468/c11-default-constructor-of-primitive-types-in-assignment-to-anonymous-instanc
The Stackoverflow discussion, "C++11: Default constructor of primitive types in assignment to anonymous instance."

24.10 Variadic Templates

Prior to C++11, each class or function template had a *fixed* number of template parameters. If you needed a class or function template with *different* numbers of template parameters, you were requrired to define a template for each case. A **variadic template** accepts *any* number of arguments, which can greatly simplify template programming. For example, you can provide one variadic function template rather than many overloaded ones with different numbers of parameters. Many template libraries prior to C++11 included

large amounts of duplicate code or made use of complex preprocessor macros to generate all the necessary template definitions. Variadic templates make it easier to implement such libraries. C++11's **`sizeof...`** operator can be used to determine the number of items in a variadic template's **parameter pack**, which is created by the compiler to store zero or more arguments to a variadic template. For an example of a variadic template, see `tuples` in Section 24.11.

Web Resources for Variadic Templates

`http://en.cppreference.com/w/cpp/language/parameter_pack`
Definition of a parameter pack and an example of a variadic function template, which must be implemented using recursion.

`http://en.wikipedia.org/wiki/Variadic_template`
Definition: From Wikipedia.

`http://en.cppreference.com/w/cpp/language/sizeof...`
`sizeof...` operator reference page discusses how the `sizeof...` operator is used with variadic templates.

`http://en.wikipedia.org/wiki/Sizeof#sizeof..._and_variadic_template_packs`
The Wikipedia description of the C++ `sizeof...` operator.

`http://oopscenities.net/2011/07/19/c0x-variadic-templates-functions/`
Blog: "C++11: Variadic Templates (Functions)," by Ernesto Bascon Pantoja. Discusses the variadic template functions.

`http://www.cplusplus.com/articles/EhvU7k9E/`
Article: "C++11-New Features-Variadic Templates," by Henri Korpela. Discusses what a variadic template is, the ellipsis operator, the `sizeof` operator, two ellipsis operators together, and uses of variadic templates.

`http://thenewcpp.wordpress.com/2011/11/23/variadic-templates-part-1-2/`
Blog: "Variadic Templates, Part 1," by Jarryd Beck. Discusses what variadic functions are and using them in classes.

`http://thenewcpp.wordpress.com/2011/11/29/variadic-templates-part-2/`
Blog: "Variadic Templates, Part 2," by Jarryd Beck. Discusses using variadic templates for function parameters.

`http://thenewcpp.wordpress.com/2012/02/15/variadic-templates-part-3-or-how-i-wrote-a-variant-class/`
Blog: "Variadic Templates, Part3 (Or How I Wrote a Variant Class," by Jarryd Beck. Discusses a stack based replacement for `boost::variant`.

`http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Variadic-Templates-are-Funadic`
Video: "Variadic Functions are `Funadic`," with Andrei Alexandrescu. This 1-1/2 hour talk provides a solid coverage of variadic fundamentals, including `typelists`, the archetypal "safe `printf`" mechanics, and tuple construction and access. It also discusses more advanced uses, such as structured argument lists.

`http://www.devx.com/cplus/Article/41533`
Article: "An Introduction to Variadic Templates in C++0x," by Anthony Williams. Discusses declaring a variadic template, type-safe variadic functions, and uses of variadic class templates.

`http://www.generic-programming.org/~dgregor/cpp/lib-variadics.pdf`
White paper: "Variadic Templates for the C++0x Standard Library," by Douglas Gregor and Jaakko Jarvi. Discusses the general utilities library, tuple creation functions, tuple helper class, element access, relational operators, function objects, requirements, function object return types and more.

24.11 Tuples

11 A common example of a variadic template is the C++11's new `tuple` class (from header `<tuple>`), which is a generalization of class template `pair` (introduced in Chapter 15). A **`tuple`** is a *fixed-size* collection of values that can be of *any* type. In a `tuple` declaration, every template type parameter specifies the type of a corresponding value in the `tuple`. So the number of `tuple` elements matches the number of type parameters.

Creating a `tuple`

You can create a `tuple` either by declaring a `tuple` and specifying its type parameters or by using the **`make_tuple` function** (also from header `<tuple>`), which infers the type parameters from the function arguments. The declaration

```
tuple<string, string, int, double> hammerInventory(
   "12345", "Hammer", 32, 9.95 );
```

might represent information about the hammer inventory for a hardware store (part number, part name, quantity in stock and price). You can also create this `tuple` as follows

```
auto hammerInventory =
   make_tuple( string("12345"), string("Hammer"), 32, 9.95 );
```

11 In this case, we use C++11's `auto` keyword to infer the type of the `hammerInventory` variable from the return value of `make_tuple`.

Using `get<index>` *to Obtain a* `tuple` *Member*

Recall that class template `pair` contains public members `first` and `second` for accessing the two members of a `pair`. The number of members in a `tuple` varies, so there are no such `public` data members in class template `tuple`. Instead, the `<tuple>` header provides the function template **`get<`*index*`>(`*tupleObject*`)`**, which returns a reference to the member of *tupleObject* at the specified *index*. As in an array or container, the first element has index 0. The returned reference is either an *rvalue* reference or an *lvalue* reference depending on whether the member is an *rvalue* or *lvalue*. To obtain a reference to the second element of the `hammerInventory` tuple, you'd write

```
string partName = get<1>( hammerInventory );
```

Other `tuple` *Features*

Fig. 24.15 shows several other features of class template `tuple`.

Feature	Description
<, <=, >, >=, ==, !=	`tuple`s that contain the same number of members can be compared to one another using the relational and equality operators. Corresponding members of each tuple are compared and must support < to work with relational `tuple` comparisons or == to work with equality `tuple` comparisons.
default constructor	Creates a `tuple` in which each member is value initialized—primitive type values are set to 0 or the equivalent of 0 and objects of class types are initialized with their default constructors.

Fig. 24.15 | Some features of class template `tuple`.

Feature	Description
copy constructor	Copies a `tuple`'s elements into a new `tuple` of the same type. The element types stored in the constructor argument must have a copy constructor.
move constructor	Moves a `tuple`'s elements into a new `tuple` of the same type.
copy assignment	Uses the assignment operator (=) to copy the elements of the `tuple` in the right operand into a `tuple` of the same type in the left operand. The element types stored in the constructor argument must be copy assignable.
move assignment	Uses the assignment operator (=) to move the elements of the `tuple` in the right operand into a `tuple` of the same type in the left operand. The element types stored in the constructor argument must be copy assignable.

Fig. 24.15 | Some features of class template `tuple`.

Web Resources for ***`tuple`s***

`http://en.cppreference.com/w/cpp/utility/tuple`
C++ Reference for `std::tuple`.

`http://stackoverflow.com/questions/10259351/what-are-good-use-cases-for-tuples-in-c11`
Stackoverflow: "What are good use-cases for tuples in C++11?"

`http://cpp-next.com/archive/2010/11/expressive-c-trouble-with-tuples/`
Blog: "Expressive C++: The Trouble with Tuples," by Eric Niebler.

`http://nealabq.com/blog/2008/12/10/tuples-and-structs/`
Tuples and Structs.

`http://mitchnull.blogspot.com/2012/06/c11-tuple-implementation-details-part-1.html`
Blog: "C++11 Tuple Implementation Details."

`http://arkaitzj.wordpress.com/2009/11/20/c-tuples-a-quick-overview/`
Blog: "Tuples a Quick Overview."

`http://yapb-soc.blogspot.com/2012/12/fun-with-tuples.html`
Blog: "Fun with Tuples."

`http://blog.codygriffin.com/2012/09/a-case-for-tuples.html`
Blog: "A Case for Tuples."

24.12 `initializer_list` Class Template

Previously, you learned how to use C++11's list initialization capabilities to initialize variables and containers. You can also define functions and constructors that receive list initializers as arguments. To do so, you specify a parameter that uses the **`initializer_list`** class template (from the header **`<initializer_list>`**). The program of Fig. 24.16 defines a function template `sum` (lines 8–18) that receives an `initializer_list` (line 9) and sums its elements. An initializer list can be used with the range-based `for` statement (lines 14–15) to iterate through all items in the `initializer_list`. In main, lines 24, 38 and 34 demonstrate passing a list initializer to `sum`'s `initializer_list` parameter.

```
// Fig. 24.16: fig24_16.cpp
// Summing the elements of an initializer list
#include <iostream>
#include <initializer_list>
using namespace std;

// sum the elements of an initializer_list
template <typename T>
T sum( initializer_list<T> list )
{
   T total{}; // value initialize total based on type T

   // sum all the elements in list; requires += operator for type T
   for ( auto item : list )
      total += item;

   return total;
} // end function template sum

int main()
{
   // display the sum of four ints contained in a list initializer
   cout << "The sum of the items in { 1, 2, 3, 4 } is: "
      << sum( { 1, 2, 3, 4 } ) << endl;

   // display the sum of three doubles contained in a list initializer
   cout << "The sum of the items in { 1.1, 2.2, 3.3 } is: "
      << sum( { 1.1, 2.2, 3.3 } ) << endl;

   // display the sum of two strings contained in a list initializer
   string s1{ "Happy " };
   string s2{ "birthday!" };
   cout << "The sum of the items in { s1, s2 } is: "
      << sum( { s1, s2 } ) << endl;
} // end main
```

Fig. 24.16 | Summing the elements of a list initializer.

In addition, the class template `initializer_list` provides the following member functions:

- `size`—returns an `initializer_list`'s number of elements.
- `begin`—returns an iterator pointing to the `intializer_list`'s first element.
- `end`—returns an iterator pointing to one past the `intializer_list`'s last element.

Many of the C++ Standard Library containers now include constructors that receive `initializer_lists`. For example, you could initialize a `vector<int>` as follows:

```
vector< int > integers{ 1, 2, 3, 4, 5, 6 };
```

or a `map<string, int>` as follows:

```
map< string, int, less< string > > pairs
   { { "hammer", 22 }, { "drill", 15 }, { "saw", 40 } };
```

Figure 24.16 was tested in GNU's C++ 4.7 and Apple's Xcode LLVM compilers. At the time of this writing, list initializers and the `initializer_list` class were not fully supported in Visual C++.

Web Resources for ***`initializer_list`***

`http://en.cppreference.com/w/cpp/utility/initializer_list`
The `initializer_list` page on `Cppreference.com`.

`http://stackoverflow.com/questions/9676538/using-a-c11-initializer-list-with-a-recursively-defined-type-using-constexpr`
The Stackoverflow discussion, "Using a C++11 `initializer_list` with a recursively defined type using `constexpr`."

`http://www.informit.com/articles/article.aspx?p=1852519`
The article, "Get to Know the New C++11 Initialization Forms," by Danny Kalev.

`http://allanmcrae.com/2012/06/c11-part-5-initialization/`
Allan McRae's blog, "C++11 – Part 5: Initialization."

`http://yoopscenities.net/2011/05/09/c0x-initializer-lists/`
The blog, "C++11: Initializer lists," by Ernesto Bascón Pantoja.

24.13 Inherited Constructors with Multiple Inheritance

11

In Section 11.4, we showed how a derived class in C++11 can inherit a base class's constructors, and in Sections 21.7–21.8 we discussed multiple inheritance. A class with *multiple base classes* can inherit constructors from *any* of its base classes. If a class inherits constructors with the *same signature* from two or more base classes, then the derived class *must* define its own version of that constructor; otherwise, a compilation error occurs. As you know, in any class that *explicitly* defines a constructor, the compiler will *not* define a default constructor. When a derived class is inheriting constructors from a base class and explicitly defines a constructor, the derived class will *not* inherit the default constructor from the base class. In that case, if a default constructor is needed, the derived class *must* define a default constructor either by using `= default` to tell the compiler to gerenate the default constructor or by explicitly defining a constructor that can be called with no arguments.

24.14 Regular Expressions with the `regex` Library

11

Regular expressions are specially formatted `string`s that are used to find patterns in text. They can be used to validate data to ensure that it is in a particular format. For example, a zip code must consist of five digits, and a last name must start with a capital letter.

The `regex` library (from header `<regex>`) provides several classes and algorithms for recognizing and manipulating regular expressions. Class template **`basic_regex`** represents a regular expression. The algorithm **`regex_match`** returns true if a `string` matches the regular expression. With `regex_match`, the entire string must match the regular expression. The `regex` library also provides the algorithm **`regex_search`**, which returns `true` if any part of an arbitrary `string` matches the regular expression.

Regular Expression Character Classes
The table in Fig. 24.17 specifies some **character classes** that can be used with regular expressions. A character class is not a C++ class—rather it's simply an escape sequence that represents a group of characters that might appear in a `string`.

Character class	Matches	Character class	Matches
\d	any decimal digit	\D	any non-digit
\w	any word character	\W	any non-word character
\s	any whitespace character	\S	any non-whitespace character

Fig. 24.17 | Character classes.

A **word character** is any alphanumeric character or underscore. A **whitespace** character is a space, tab, carriage return, newline or form feed. A **digit** is any numeric character. Regular expressions are not limited to the character classes in Fig. 24.17. In Fig. 24.18, you'll see that regular expressions can use other notations to search for complex patterns in `string`s.

24.14.1 Regular Expression Example

The program in Fig. 24.18 tries to match birthdays to a regular expression. For demonstration purposes, the expression in line 11 matches only birthdays that do not occur in April and that belong to people whose names begin with `"J"`. This example was tested with Microsoft's Visual C++ 2012 and Apple's Xcode LLVM compilers—GNU does not yet support regular expressions.

```
1  // Fig. 24.18: fig24_18.cpp
2  // Demonstrating regular expressions.
3  #include <iostream>
4  #include <string>
5  #include <regex>
6  using namespace std; // allows use of features in both std and std::tr1
7
8  int main()
9  {
10    // create a regular expression
11    regex expression( "J.*\\d[0-35-9]-\\d\\d-\\d\\d" );
12
13    // create a string to be tested
14    string string1 = "Jane's Birthday is 05-12-75\n"
15       "Dave's Birthday is 11-04-68\n"
16       "John's Birthday is 04-28-73\n"
17       "Joe's Birthday is 12-17-77";
18
19    // create an smatch object to hold the search results
20    smatch match;
21
22    // match regular expression to string and print out all matches
23    while ( regex_search( string1, match, expression,
24       regex_constants::match_not_dot_newline ) )
25    {
26       cout << match.str() << endl; // print the matching string
```

Fig. 24.18 | Regular expressions checking birthdays. (Part 1 of 2.)

```
27
28         // remove the matched substring from the string
29         string1 = match.suffix();
30      } // end while
31   } // end function main
```

```
Jane's Birthday is 05-12-75
Joe's Birthday is 12-17-77
```

Fig. 24.18 | Regular expressions checking birthdays. (Part 2 of 2.)

Creating the Regular Expression

Line 11 creates a **regex** object by passing a regular expression to the `regex` constructor. The name `regex` is a `typedef` of the `basic_regex` class template that uses `char`s. We precede each backslash character in the initializer string with an additional backslash. Recall that C++ treats a backslash in a string literal as the beginning of an escape sequence. To insert a literal backslash in a string, you must escape the backslash character with another backslash. For example, the character class `\d` must be represented as `\\d` in a C++ string literal.

The first character in the regular expression, `"J"`, is a literal character. Any `string` matching this regular expression is required to start with `"J"`. In a regular expression, the dot character `"."` matches any single character. When the dot character is followed by an asterisk, as in `".*"`, the regular expression matches any number of unspecified characters. In general, when the operator `"*"` is applied to a pattern, the pattern will match *zero or more* occurrences. By contrast, applying the operator `"+"` to a pattern causes the pattern to match *one or more* occurrences. For example, both `"A*"` and `"A+"` will match `"A"`, but only `"A*"` will match an empty `string`.

As indicated in Fig. 24.17, `"\d"` matches any decimal digit. To specify sets of characters other than those that belong to a predefined character class, characters can be listed in square brackets, `[]`. For example, the pattern `"[aeiou]"` matches any vowel. Ranges of characters are represented by placing a dash (-) between two characters. In the example, `"[0-35-9]"` matches only digits in the ranges specified by the pattern—i.e., any digit between `0` and `3` or between `5` and `9`; therefore, the pattern matches any digit except `4`. You can also specify that a pattern should match anything other than the characters in the brackets. To do so, place `^` as the first character in the brackets. It is important to note that `"[^4]"` is not the same as `"[0-35-9]"`; `"[^4]"` matches any non-digit and digits other than `4`.

Although the `"-"` character indicates a range when it is enclosed in square brackets, instances of the `"-"` character outside grouping expressions are treated as literal characters. Thus, the regular expression in line 11 searches for a `string` that starts with the letter "`J`", followed by any number of characters, followed by a two-digit number (of which the second digit cannot be `4`), followed by a dash, another two-digit number, a dash and another two-digit number.

Using the Regular Expression to Search for Matches

Line 20 creates an `smatch` (pronounced "ess-match"; a `typedef` for `match_results`) object. A **`match_results`** object, when passed as an argument to one of the `regex` algorithms, stores the regular expression's match. An **`smatch`** stores an object of type `string::const_iterator`

that you can use to access the matching `string`. There are `typedef`s to support other string representations such as `const char*` (`cmatch`).

The `while` statement (lines 23–30) searches `string1` for matches to the regular expression until none can be found. We use the call to `regex_search` as the `while` statement condition (lines 23–24). `regex_search` returns `true` if the `string` (`string1`) contains a match to the regular expression (`expression`). We also pass an `smatch` object to `regex_search` so we can access the matching `string`. The last argument, **`match_not_eol`**, prevents the `"."` character from matching a newline character. The body of the `while` statement prints the substring that matched the regular expression by callig the match object's `str` function (line 26) and removes it from the `string` being searched by calling the match object's `suffix` function and assigning its result back to `string1` (line 29). The call to the `match_results` member function **`suffix`** returns a `string` from the end of the match to the end of the `string` being searched. The output in Fig. 24.18 displays the two matches that were found in `string1`. Notice that both matches conform to the pattern specified by the regular expression.

Quantifiers

The asterisk (`*`) in line 11 of Fig. 24.18 is more formally called a **quantifier**. Figure 24.19 lists various quantifiers that you can place after a pattern in a regular expression and the purpose of each quantifier.

Quantifier	Matches
`*`	Matches zero or more occurrences of the preceding pattern.
`+`	Matches one or more occurrences of the preceding pattern.
`?`	Matches zero or one occurrences of the preceding pattern.
`{n}`	Matches exactly *n* occurrences of the preceding pattern.
`{n,}`	Matches at least *n* occurrences of the preceding pattern.
`{n,m}`	Matches between *n* and *m* (inclusive) occurrences of the preceding pattern.

Fig. 24.19 | Quantifiers used in regular expressions.

We've already discussed how the asterisk (`*`) and plus (`+`) quantifiers work. The question mark (`?`) quantifier matches zero or one occurrences of the pattern that it quantifies. A set of braces containing one number, `{n}`, matches exactly *n* occurrences of the pattern it quantifies. We demonstrate this quantifier in the next example. Including a comma after the number enclosed in braces matches at least *n* occurrences of the quantified pattern. The set of braces containing two numbers, `{n,m}`, matches between *n* and *m* occurrences (inclusively) of the pattern that it quantifies. All of the quantifiers are **greedy**—they'll match as many occurrences of the pattern as possible until the pattern fails to make a match. If a quantifier is followed by a question mark (`?`), the quantifier becomes **lazy** and will match as few occurrences as possible as long as there is a successful match.

24.14.2 Validating User Input with Regular Expressions

The program in Fig. 24.20 presents a more involved example that uses regular expressions to validate name, address and telephone number information input by a user. This example was tested with Microsoft's Visual C++ 2012 and Apple's Xcode LLVM compiler. At the time of this writing, Apple's Xcode LLVM compiler did not properly handle the *OR* (|) operation for the regular expression in line 21.

```
1  // Fig. 24.20: fig24_20.cpp
2  // Validating user input with regular expressions.
3  #include <iostream>
4  #include <string>
5  #include <regex>
6  using namespace std;
7
8  bool validate( const string&, const string& ); // validate prototype
9  string inputData( const string&, const string& ); // inputData prototype
10
11 int main()
12 {
13    // enter the last name
14    string lastName = inputData( "last name", "[A-Z][a-zA-Z]*" );
15
16    // enter the first name
17    string firstName = inputData( "first name", "[A-Z][a-zA-Z]*" );
18
19    // enter the address
20    string address = inputData( "address",
21       "[0-9]+\\s+([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
22
23    // enter the city
24    string city =
25       inputData( "city", "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
26
27    // enter the state
28    string state = inputData( "state",
29       "([a-zA-Z]+|[a-zA-Z]+\\s[a-zA-Z]+)" );
30
31    // enter the zip code
32    string zipCode = inputData( "zip code", "\\d{5}" );
33
34    // enter the phone number
35    string phoneNumber = inputData( "phone number",
36       "[1-9]\\d{2}-[1-9]\\d{2}-\\d{4}" );
37
38    // display the validated data
39    cout << "\nValidated Data\n\n"
40       << "Last name: " << lastName << endl
41       << "First name: " << firstName << endl
42       << "Address: " << address << endl
43       << "City: " << city << endl
44       << "State: " << state << endl
```

Fig. 24.20 | Validating user input with regular expressions. (Part 1 of 3.)

```
45         << "Zip code: " << zipCode << endl
46         << "Phone number: " << phoneNumber << endl;
47   } // end of function main
48
49   // validate the data format using a regular expression
50   bool validate( const string &data, const string &expression )
51   {
52      // create a regex to validate the data
53      regex validationExpression = regex( expression );
54      return regex_match( data, validationExpression );
55   } // end of function validate
56
57   // collect input from the user
58   string inputData( const string &fieldName, const string &expression )
59   {
60      string data; // store the data collected
61
62      // request the data from the user
63      cout << "Enter " << fieldName << ": ";
64      getline( cin, data );
65
66      // validate the data
67      while ( !( validate( data, expression ) ) )
68      {
69         cout << "Invalid " << fieldName << ".\n";
70         cout << "Enter " << fieldName << ": ";
71         getline( cin, data );
72      } // end while
73
74      return data;
75   } // end of function inputData
```

```
Enter last name: 12345
Invalid last name.
Enter last name: Blue
Enter first name: Betty
Enter address: 123
Invalid address.
Enter address: 123 Main Street
Enter city: SomeCity
Enter state: SomeState
Enter zip code: 1
Invalid zip code.
Enter zip code: 55555
Enter phone number: 555-555-123
Invalid phone number.
Enter phone number: 555-555-1234

Validated Data

Last name: Blue
First name: Betty
Address: 123 Main Street
```

Fig. 24.20 | Validating user input with regular expressions. (Part 2 of 3.)

```
City: SomeCity
State: SomeState
Zip code: 55555
Phone number: 555-555-1234
```

Fig. 24.20 | Validating user input with regular expressions. (Part 3 of 3.)

Function ***inputData***

The program first asks the user to input a last name (line 14) by calling the `inputData` function. The `inputData` function (lines 58–75) takes two arguments, the name of the data being input and a regular expression that it must match. The function prompts the user (line 63) to input the specified data. Then `inputData` checks whether the input is in the correct format by calling the `validate` function (lines 50–55). That function takes two arguments—the `string` to validate and the regular expression it must match. The function first uses the expression to create a `regex` object (line 53). Then it calls `regex_match` to determine whether the `string` matches the expression. If the input isn't valid, `inputData` prompts the user to enter the information again. Once the user enters a valid input, the data is returned as a `string`. The program repeats that process until all the data fields have been validated (lines 14–36). Then we display all the information (lines 39–46).

Matching an Entire String

In the previous example, we searched a `string` for substrings that matched a regular expression. In this example, we want to ensure that the entire `string` for each input conforms to a particular regular expression. For example, we want to accept `"Smith"` as a last name, but not `"9@Smith#"`. We use `regex_match` here instead of `regex_search`—`regex_match` returns `true` only if the entire `string` matches the regular expression. Alternatively, you can use a regular expression that begins with a `"^"` character and ends with a `"$"` character. The characters `"^"` and `"$"` represent the beginning and end of a `string`, respectively. Together, these characters force a regular expression to return a match only if the entire `string` being processed matches the regular expression.

Matching a Range of Characters

The regular expression in line 14 uses the square bracket and range notation to match an uppercase first letter followed by letters of any case—`a-z` matches any lowercase letter, and `A-Z` matches any uppercase letter. The `*` quantifier signifies that the second range of characters may occur zero or more times in the `string`. Thus, this expression matches any `string` consisting of one uppercase letter, followed by zero or more additional letters.

Matching Spaces and Digits; Using | to Match One String or Another

The notation `\s` matches a single white-space character (lines 21, 25 and 29). The expression `\d{5}`, used for the `zipCode string` (line 32), matches any five digits. The character `"|"` (lines 21, 25 and 29) matches the expression to its left *or* the expression to its right. For example, `Hi (John|Jane)` matches both `Hi John` and `Hi Jane`. In line 21, we use the character `"|"` to indicate that the address can contain a word of one or more characters *or* a word of one or more characters followed by a space and another word of one or more characters. Note the use of parentheses to group parts of the regular expression. Quantifiers may be applied to patterns enclosed in parentheses to create more complex regular expressions.

Purpose of Each Regular Expression in This Example
The `lastName` and `firstName` variables (lines 14 and 17) both accept `string`s of any length that begin with an uppercase letter. The regular expression for the `address string` (line 21) matches a number of at least one digit, followed by a space, then either one or more letters or else one or more letters followed by a space and another series of one or more letters. Therefore, `"10 Broadway"` and `"10 Main Street"` are both valid addresses. As currently formed, the regular expression in line 21 doesn't match an address that does not start with a number, or that has more than two words. The regular expressions for the `city` (line 25) and `state` (line 29) `string`s match any word of at least one character or, alternatively, any two words of at least one character if the words are separated by a single space. This means both `Waltham` and `West Newton` would match. Again, these regular expressions would not accept names that have more than two words. The regular expression for the `zipCode string` (line 32) ensures that the zip code is a five-digit number. The regular expression for the `phoneNumber string` (line 36) indicates that the phone number must be of the form `xxx-yyy-yyyy`, where the `x`s represent the area code and the `y`s the number. The first `x` and the first `y` cannot be zero, as specified by the range `[1-9]` in each case.

24.14.3 Replacing and Splitting Strings

Sometimes it's useful to replace parts of one `string` with another or to split a `string` according to a regular expression. For this purpose, the `regex` library provides the algorithm `regex_replace` and the `regex_token_iterator` class, which we demonstrate in Fig. 24.21. This example was tested with Microsoft's Visual C++ 2012 and Apple's Xcode LLVM compilers.

```
1  // Fig. 24.21: fig24_21.cpp
2  // Using regex_replace algorithm.
3  #include <iostream>
4  #include <string>
5  #include <regex>
6  using namespace std;
7
8  int main()
9  {
10    // create the test strings
11    string testString1 = "This sentence ends in 5 stars *****";
12    string testString2 = "1, 2, 3, 4, 5, 6, 7, 8";
13    string output;
14
15    cout << "Original string: " << testString1 << endl;
16
17    // replace every * with a ^
18    testString1 =
19       regex_replace( testString1, regex( "\\*" ), string( "^" ) );
20    cout << "^ substituted for *: " << testString1 << endl;
21
22    // replace "stars" with "carets"
23    testString1 =
24       regex_replace( testString1, regex( "stars" ), string( "carets" ) );
```

Fig. 24.21 | Using `regex_replace` algorithm. (Part 1 of 2.)

```
25      cout << "\"carets\" substituted for \"stars\": "
26         << testString1 << endl;
27
28      // replace every word with "word"
29      testString1 =
30         regex_replace( testString1, regex( "\\w+" ), string( "word" ) );
31      cout << "Every word replaced by \"word\": " << testString1 << endl;
32
33      // replace the first three digits with "digit"
34      cout << "\nOriginal string: " << testString2 << endl;
35      string testString2Copy = testString2;
36
37      for ( int i = 0; i < 3; ++i ) // loop three times
38      {
39         testString2Copy = regex_replace( testString2Copy,
40            regex( "\\d" ), "digit", regex_constants::format_first_only );
41      } // end for
42
43      cout << "Replace first 3 digits by \"digit\": "
44         << testString2Copy << endl;
45
46      // split the string at the commas
47      cout << "string split at commas [";
48
49      regex splitter( ",\\s" ); // regex to split a string at commas
50      sregex_token_iterator tokenIterator( testString2.begin(),
51         testString2.end(), splitter, -1 ); // token iterator
52      sregex_token_iterator end;  // empty iterator
53
54      while ( tokenIterator != end ) // tokenIterator isn't empty
55      {
56         output += "\"" + (*tokenIterator).str() + "\", ";
57         ++tokenIterator; // advance the iterator
58      } // end while
59
60      // delete the ", " at the end of output string
61      cout << output.substr( 0, output.size() - 2 ) << "]" << endl;
62   } // end of function main
```

```
Original string: This sentence ends in 5 stars *****
^ substituted for *: This sentence ends in 5 stars ^^^^^
"carets" substituted for "stars": This sentence ends in 5 carets ^^^^^
Every word replaced by "word": word word word word word word ^^^^^

Original string: 1, 2, 3, 4, 5, 6, 7, 8
Replace first 3 digits by "digit": digit, digit, digit, 4, 5, 6, 7, 8
string split at commas ["1", "2", "3", "4", "5", "6", "7", "8"]
```

Fig. 24.21 | Using `regex_replace` algorithm. (Part 2 of 2.)

Replacing Substrings with `regex_replace`

Algorithm **`regex_replace`** replaces text in a `string` with new text wherever the original `string` matches a regular expression. In line 19, `regex_replace` replaces every instance of

"*" in testString1 with "^". The regular expression ("*") precedes character "*" with a backslash, \. Typically, "*" is a quantifier indicating that a regular expression should match any number of occurrences of a preceding pattern. However, in this case we want to find all occurrences of the literal character "*"; to do this, we must escape character "*" with character "\". By escaping a special regular expression character with a \, we tell the regular expression matching engine to find the actual character "*" rather than use it as a quantifier. Also, the first and last arguments to this version of function regex_replace must be strings. Lines 23–24 use regex_replace to replace the string "stars" in testString1 with the string "carets". Lines 29–30 use regex_replace to replace every word in testString1 with the string "word".

Lines 37–41 replace the first three instances of a digit ("\d") in testString2 with the text "digit". We pass **regex_constants::format_first_only** as an additional argument to regex_replace (lines 39–40). This argument tells regex_replace to replace only the first substring that matches the regular expression. Normally regex_replace would replace all occurrences of the pattern. We put this call inside a for loop that runs three times; each time replacing the first instance of a digit with the text "digit". We use a copy of testString2 (line 35) so we can use the original textString2 for the next part of the example.

Obtaining Substrings with a ***regex_token_iterator***

Next we use a regex_token_iterator to divide a string into several substrings. A **regex_token_iterator** iterates through the parts of a string that match a regular expression. Lines 49 and 51 use sregex_token_iterator, which is a typedef that indicates the results are to be manipulated with a string::const_iterator. We create the iterator (lines 49–50) by passing the constructor two iterators (testString2.begin() and testString2.end()), which represent the beginning and end of the string to iterate over and the regular expression to look for. In our case we want to iterate over the parts of the string that *don't* match the regular expression. To do that we pass -1 to the constructor. This indicates that it should iterate over each substring that doesn't match the regular expression. The original string is broken at delimiters that match the specified regular expression. We use a while statement (lines 53–57) to add each substring to the string output. The regex_token_iterator end (line 51) is an empty iterator. We've iterated over the entire string when tokenIterator equals end (line 53).

24.15 Raw String Literals

Section 24.14 presented regular expressions. Because C++ treats a backslash in a string literal as the beginning of an escape sequence, we were required to precede each character class in a regular expression with another backslash. For example, the character class \d was represented as \\d. As of C++11, C++ now supports **raw string literals** that have the format

11

```
R"optionalDelimiter(characters)optionalDelimiter"
```

where the *optionalDelimiter* before the left parenthesis, (, and after the right parenthesis,), must be identical, if provided. The parentheses are required around the *characters* that compose the raw string literal. The compiler automatically inserts backslashes as necessary in a raw string literal to properly escape special characters like double quotes ("), backslashes (\), etc.

As an example, the string literal in line 11 in Fig. 24.18

```
"J.*\\d[0-35-9]-\\d\\d-\\d\\d"
```

could be written with a raw string literal as

```
R"(J.*\d[0-35-9]-\d\d-\d\d)"
```

which makes the regular expression more readable. The compiler converts the raw string literal into the original string literal from line 11 in Fig. 24.18.

The preceding raw string literal can include optional delimiters up to 16 characters long before the left parenthesis, `(`, and after the right parenthesis, `)`, as in

```
R"MYDELIMITER(J.*\d[0-35-9]-\d\d-\d\d)MYDELIMITER"
```

Raw string literals are not restricted to use with regular expressions. They may be used in any context that requires a string literal. They may also include line breaks, in which case the compiler inserts \n escape sequences. For example, the raw string literal

```
R"(multiple
lines
of
text)"
```

is treated as the string literal

```
"multiple\nlines\nof\ntext"
```

Web Resources for Raw String Literals

`http://solarianprogrammer.com/2011/10/16/cpp-11-raw-strings-literals-tutorial/`
The tutorial, "C++11 raw strings literals."

`http://stackoverflow.com/questions/3093632/why-must-c-c-string-literal-declarations-be-single-line`
The Stackoverflow discussion, "Why must C/C++ string literal declarations be single-line?"

`http://impactcore.blogspot.com/2011/03/c0x-raw-string-literals-simple-example.html`
The blog, "C++0x Raw String Literals: A simple example."

`http://en.cppreference.com/w/cpp/language/string_literal`
The string literal reference page.

`http://akrzemi1.wordpress.com/2012/08/12/user-defined-literals-part-i/`
The blog, "User-defined literals — Part I."

24.16 Wrap-Up

In this chapter we discussed various new language and library features of C++11. We showed how to use the other smart pointer library classes (`unique_ptr` was introduced in Chapter 17). You learned how to use the `shared_ptr` and `weak_ptr` classes to avoid memory leaks when using dynamically allocated memory. We demonstrated how to use custom deleter functions to allow `shared_ptrs` to manage resources that require special destruction procedures. We also explained how `weak_ptrs` can be used to prevent memory leaks in circularly referential data.

We introduced multithreading—one of the most significant updates in the C++11 standard. You learned how to use the `async` function template and `future` class template to execute a function in a separate thread and to obtain the function's results.

You learned how to use noexcept to indicate that a function does not throw any exceptions, which allows the compiler to perform additional optimizations. We introduced C++11's features for implementing move semantics to prevent unnecessary copying of objects that are about to be destroyed, and we showed how to implement a class's move constructor and move assignment operator using *rvalue* references. You also saw how to convert an *lvalue* to an *rvalue* reference with the Standard Library function `std::move`.

We discussed how to use a `static_assert` declaration to test constant integral expressions at compile time rather than runtime. This feature is typically used by library developers to report incorrect usage of a library at compile time.

You learned that the `decltype` operator enables the compiler to determine an expression's type at compile time, and that `decltype` is particularly useful when working with complex template types for which it's often difficult to provide, or even determine, the proper type declaration.

We introduced `constexpr` which can be used to declare variables, functions and constructors that are evaluated at compile time and that result in a constant. You learned how to use = default to indicate that the compiler should generate default versions of a class's special member functions—default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator and destructor.

We introduced the concept of variadic templates that can have any number of type parameters then discussed C++11's new tuple class template, which is a variadic template for creating a fixed-size collection of values in which the type of each value matches the type of the corresponding type parameter.

Finally, we discussed the `regex` library and the symbols that are used to form regular expressions. We provided examples of how to use regular-expression classes, including `regex`, `match_results` and `regex_token_iterator`. You learned how to find patterns in a string and match entire strings to patterns with algorithms `regex_search` and `regex_match`. We demonstrated how to replace characters in a string with `regex_replace` and how to split strings into tokens with a `regex_token_iterator`.