23.1 Introduction

Throughout the book, we've discussed many of the key features of the impending new C++ Standard (C++0x). In this chapter, we introduce the Boost C++ Libraries and Technical Report 1 (TR1), and consider additional C++0x features. The **Boost C++ Libraries** are free, open source libraries created by members of the C++ community. Boost provides useful, well-designed libraries that work well with the existing C++ Standard Library. Boost can be used on many platforms with many different compilers. We overview some of the popular Boost libraries and provide code examples for the regular expression and smart pointer libraries. **Technical Report 1** describes the proposed changes to the C++ Standard Library, many of which are based on current Boost libraries. These libraries add useful functionality to C++. **C++0x** is the working name for the next version of the C++ Standard. It includes some additions to the core language, many of the library additions described in TR1 and other library enhancements.

23.2 Deitel Online C++ and Related Resource Centers

We regularly post online Resource Centers on key programming, software, Web 2.0 and Internet business topics at `www.deitel.com/ResourceCenters.html`. We've created several online Resource Centers that provide links to key information on Boost and C++0x. Visit the C++ Boost Libraries Resource Center at `www.deitel.com/CPlusPlusBoostLibraries/` to find current information on the available libraries and new releases. You can find current information on TR1 and C++0x in the C++0x section of the C++ Resource Center at `www.deitel.com/cplusplus/` (click **C++0x** in the **Categories** list). We used GNU C++ 4.5 and Visual C++ 2010 Express Edition to compile the examples in this chapter.

23.3 Boost Libraries

The idea for an online repository of free, peer-reviewed, open source C++ libraries was first proposed in a paper by Beman Dawes in 1998.[1] He and Robert Klarer got the idea while

1. "Proposal for a C++ Library Repository Web Site," Beman G. Dawes, May 6, 1998, `www.boost.org/users/proposal.pdf`.

attending a C++ Standards Committee meeting. The paper suggested a website where C++ programmers could find and share libraries and foster further C++ development. That idea eventually developed into the Boost Libraries at `www.boost.org`. Boost has grown to over 100 libraries, with more being added frequently. Today there are thousands of programmers in the Boost community.

Adding a New Library to Boost

Boost accepts useful, well-designed, portable libraries from anyone willing to contribute. Potential Boost libraries should conform to the C++ Standard and use the C++ Standard Library—or other appropriate Boost libraries. There is a formal acceptance process to ensure that libraries meet Boost's high quality and portability standards.

The community's interest in a library is determined by posting to mailing lists and reading the responses. If there is interest in a library, a preliminary submission of the library is posted in the **Boost Sandbox** (`svn.boost.org/svn/boost/sandbox/`)—a code repository for libraries that are under development. The Sandbox allows other users to experiment with the library and provide feedback.

When the library is ready for a formal review, the code submission is posted to the Sandbox Vault and a review manager is selected from a list of approved volunteers. The review manager makes sure the code is ready for formal review, sets up the review schedule, reads all user reviews, and makes the final decision whether or not to accept the library. The review manager may accept the library with certain corrections or improvements that must be implemented before the library is officially added to Boost. Once a library has been accepted, the author is responsible for its maintenance.

The Boost Software License

The Boost Software License (`www.boost.org/users/license.html`) grants the rights to copy, modify, use and distribute the Boost source code and binaries for any commercial or noncommercial use. The only requirement is that the copyright and license information be distributed with any source code that is made public, though it isn't required that the source code be released. These conditions allow the Boost libraries to be used in any application. Every Boost library must conform to these conditions.

Installing the Boost Libraries

The Boost libraries can be used with minimal setup on many platforms and compilers. BoostPro Computing offers a free installer for using Boost with Visual Studio at `www.boostpro.com/download`. Most Linux distributions offer packages for Boost, though it is sometimes split up into separate packages for the headers and libraries. An installation guide available at `www.boost.org/more/getting_started/index.html` provides setup instructions for many compilers and platforms.

23.4 Boost Libraries Overview

There are many Boost libraries—too many to cover in this book. In this section, we overview some of the most useful and popular libraries. The ones listed here are part of the next C++ standard—C++0x. In the following sections, we demonstrate two of these libraries as implemented by using their implementations from the C++0x standard library.

Array [2]

Boost.Array is a wrapper for fixed-size arrays that enhances built-in arrays by supporting most of the STL container interface described in Section 22.1. Class `array` allows you to use fixed-size arrays in STL applications rather than `vectors` (dynamically sized arrays), which are not as efficient when there is no need for dynamic resizing. To use class `array` with compilers that support this C++0x feature, include the `<array>` header.

Bind [3]

Boost.Bind extends the functionality of the standard functions `std::bind1st` and `std::bind2nd`. The `bind1st` and `bind2nd` functions are used to adapt binary functions (i.e., functions that take two arguments) to be used with the standard algorithms which take unary functions (i.e., functions that take one argument). Class `bind` enhances that functionality by allowing you to adapt functions that take up to nine arguments. Class `bind` also makes it easy to reorder the arguments passed to the function using placeholders. To use class `bind` with compilers that support this C++0x feature, include the `<functional>` header.

Function [4]

Boost.Function allows you to store function pointers, member-function pointers and function objects in a function wrapper. A `function` can hold any function whose arguments and return type can be converted to match the signature of the function wrapper. For example, if the function wrapper was created to hold a function that takes a `string` and returns a `string`, it can also hold a function that takes a `char*` and returns a `char*`, because a `char*` can be converted to a `string`, using a conversion constructor. To use class `function` with compilers that support this C++0x feature, include the `<functional>` header.

Random [5]

Boost.Random allows you to create various random number generators and random number distributions. The `std::rand` and `std::srand` functions in the C++ Standard Library generate pseudo-random numbers. A **pseudo-random number generator** uses an initial state to produce seemingly random numbers—using the same initial state produces the same sequence of numbers. The `rand` function always uses the same initial state, therefore it produces the same sequence of numbers every time. The function `srand` allows you to set the initial state to vary the sequence. Pseudo-random numbers are often used in testing—the predictability enables you to confirm the results. `Boost.Random` provides pseudo-random number generators as well as generators that can produce **nondeterministic random numbers**—a set of random numbers that can't be predicted. Such random number generators are used in simulations and security scenarios where predictability is undesirable.

`Boost.Random` also allows you to specify the distribution of the numbers generated. A common distribution is the **uniform distribution**, which assigns the same probability to each number within a given range. This is similar to rolling a die or flipping a coin—each possible outcome is equally as likely. You can set this range at compile time. `Boost.Random`

---

2. Documentation for Boost.Array: `www.boost.org/doc/libs/1_45_0/doc/html/array.html`.
3. Documentation for Boost.Bind: `www.boost.org/doc/libs/1_45_0/libs/bind/bind.html`.
4. Documentation for Boost.Function: `www.boost.org/doc/libs/1_45_0/doc/html/function.html`.
5. Jens Maurer, "A Proposal to Add an Extensible Random Number Facility to the Standard Library," Document Number N1452, April 10, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1452.html`.

allows you to use a distribution in combination with any random number generator and even create your own distributions. To use these new random-number capabilities with compilers that support these C++0x features, include the `<random>` header.

Regex [6]

`Boost.Regex` provides support for processing **regular expressions** in C++. Regular expressions are used to match specific character patterns in text. Many modern programming languages have built-in support for regular expressions, but C++ does not. With `Boost.Regex`, you can search for a particular expression in a `string`, replace parts of a `string` that match a regular expression, and split a `string` into tokens using regular expressions to define the delimiters. These techniques are commonly used for text processing, parsing and input validation. To use regular expressions with compilers that support this C++0x feature, include the `<regex>` header. We discuss some regular expression capabilities in more detail in Section 23.5.

Smart_ptr [7]

`Boost.Smart_ptr` defines smart pointers that help you manage dynamically allocated resources (e.g., memory, files and database connections). Programmers often get confused about when to deallocate memory or simply forget to do it, especially when the memory is referenced by more than one pointer. Smart pointers take care of these tasks automatically. TR1 includes several smart pointers from the `Boost.Smart_ptr` library. We discussed the `unique_ptr` class in Chapter 16. **`shared_ptrs`** handle lifetime management of dynamically allocated objects. The memory is released when there are no `shared_ptrs` referencing it. **`weak_ptrs`** allow you to observe the value held by a `shared_ptr` without assuming any management responsibilities. We discuss the `shared_ptr` and `weak_ptr` in more detail in Section 23.6. To use the smart pointer classes with compilers that support these C++0x features, thisinclude the `<regex>` header.

Tuple [8]

A **tuple** is a set of objects. **`Boost.Tuple`** allows you to create sets of objects in a generic way and allows generic functions to act on those sets. The library allows you to create tuples of up to 10 objects; that limit can be extended. Class `tuple` is basically an extension to the STL's `std::pair` class template. Tuples are often used to return multiple values from a function. They can also be used to store sets of elements in an STL container where each set of elements is an element of the container. Another useful feature is the ability to set the values of variables using the elements of a tuple. To use class `tuple` with compilers that support this C++0x feature, include the `<tuple>` header.

Type_traits [9]

The **`Boost.Type_traits`** library helps abstract the differences between types to allow generic programming implementations to be optimized. The `type_traits` classes allow you

---

6. Documentation for `Boost.Regex`: `www.boost.org/doc/libs/1_45_0/libs/regex/doc/html/`.
7. Documentation for `Boost.Smart_ptr`: `www.boost.org/doc/libs/1_45_0/libs/smart_ptr/smart_ptr.htm`.
8. Documentation for `Boost.Tuple`: `www.boost.org/doc/libs/1_45_0/libs/tuple/doc/tuple_users_guide.html`.
9. Documentation for `Boost.Type_traits`, Steve Cleary, Beman Dawes, Howard Hinnant and John Maddock, `www.boost.org/doc/libs/1_45_0/libs/type_traits/doc/html/index.html`.

to determine specific traits of a type (e.g., is it a pointer or a reference type, or does the type have a `const` qualifier?) and perform type transformations to allow the object to be used in generic code. Such information can be used to optimize generic code. For example, sometimes it is more efficient to copy a collection of objects using the C function `memcpy` rather than by iterating through all the elements of the collection, as the STL `copy` algorithm does. With the `Boost.Type_traits` library, generic algorithms can be optimized by first checking the traits of the types being processed, then performing the algorithm accordingly. C++0x compilers that support these features include them in the `<type_traits>` header.

23.5 Regular Expressions with the regex Library

[*Note:* The C++0x library features used in this section's examples were not fully implemented in GNU C++ at the time of this writing. For now, if you wish to use these features in GNU C++, you can install the Boost version of the regular expressions library as discussed in Section 23.3]

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 `std::tr1::regex` library (from header `<regex>`) provides several classes and algorithms (in namespce `std::tr1`) 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. 23.1 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. 23.1 | 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. 23.1. In Fig. 23.2, you'll see that regular expressions can use other notations to search for complex patterns in `string`s.

23.5.1 Regular Expression Example

The program in Fig. 23.2 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".

```
1  // Fig. 23.2: fig23_02.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
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. 23.2 | Regular expressions checking birthdays.

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. 23.1, "\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 typedefs 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. 23.2 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. 23.2 is more formally called a **quantifier**. Figure 23.3 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. 23.3 | 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.

23.5.2 Validating User Input with Regular Expressions

The program in Fig. 23.4 presents a more involved example that uses regular expressions to validate name, address and telephone number information input by a user.

```
1  // Fig. 23.4: fig23_04.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
```

Fig. 23.4 | Validating user input with regular expressions. (Part 1 of 3.)

```
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
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 << ": ";
```

Fig. 23.4 | Validating user input with regular expressions. (Part 2 of 3.)

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

Fig. 23.4 | Validating user input with regular expressions. (Part 3 of 3.)

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).

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.

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.

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.

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.

23.5.3 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. 23.5.

```
1  // Fig. 23.5: fig23_05.cpp
2  // Using regex_replace algorithm.
3  #include <iostream>
4  #include <string>
5  #include <regex>
6  using namespace std;
7
8  int main()
9  {
```

Fig. 23.5 | Using `regex_replace` algorithm. (Part 1 of 3.)

```
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" ) );
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.length() - 2 ) << "]" << endl;
62 } // end of function main
```

Fig. 23.5 | Using `regex_replace` algorithm. (Part 2 of 3.)

```
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. 23.5 | Using regex_replace algorithm. (Part 3 of 3.)

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 `string`s. 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).

23.6 Smart Pointers

[*Note:* The C++0x library features used in this section's examples work in both Microsoft Visual C++ 2010 Express and GNU C++ 4.5. GNU C++ considers these features experimental and requires you to use the command line option `-std:c++0x` to compile the examples correctly.]

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 16.11 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++0x provides other smart pointer options with additional functionality.

23.6.1 Reference Counted `shared_ptr`

`shared_ptrs` (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_ptrs` to the same resource. `shared_ptrs` really do share the resource—if you change the resource with one `shared_ptr`, the changes also will be "seen" by the other `shared_ptrs`. The internal pointer is deleted once the last `shared_ptr` to the resource is destroyed. `shared_ptrs` use **reference counting** to determine how many `shared_ptrs` 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_ptrs` are useful in situations where multiple pointers to the same resource are needed, such as in STL containers. `shared_ptrs` can safely be copied and used in STL containers.

`shared_ptrs` 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`

Figures 23.6–23.7 define a simple class to represent a `Book` with a `string` to represent the title of the `Book`. The destructor for class `Book` (Fig. 23.7, lines 12–15) displays a message on the screen indicating that an instance is being destroyed. We use this class to demonstrate the common functionality of `shared_ptr`.

```
1  // Fig. 23.6: 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. 23.6 | Book header.

```
1  // Fig. 23.7: 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. 23.7 | Book member-function definitions.

Creating `shared_ptr`*s*

The program in Fig. 23.8 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. 23.8: fig23_08.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, ";
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";
```

Fig. 23.8 | `shared_ptr` example program. (Part 1 of 2.)

```
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
```

```
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. 23.8 | `shared_ptr` example program. (Part 2 of 2.)

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 use the `shared_ptr` constructor with the orig-

inal 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_ptrs 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_ptrs in an STL container. We create a vector of BookPtrs (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 Books 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_ptrs 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_ptrs 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_ptrs in it. The program output shows that each instance of class Book is destroyed automatically by the shared_ptrs. There is no need to delete each pointer placed in the vector.

23.6.2 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_ptrs pointing to it. When the last shared_ptr is destroyed, the resource is deleted and any remaining weak_ptrs are set to NULL. One use for weak_ptrs, 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_ptrs should be used in any situation where you need to observe the resource but don't want to assume any management responsibilities for it. The following example demonstrates the use of weak_ptrs in **circularly referential data**, a situation in which two objects refer to each other internally.

Example Using `weak_ptr`

Figures 23.9–23.12 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_ptrs and shared_ptrs to hold the cross reference to each class (Fig. 23.9 and 23.10, lines 20–21 in each figure). If we set the shared_ptrs, it creates a memory leak—we'll explain why soon and show how we can use the weak_ptrs to fix this problem.

```
1  // Fig. 23.9: 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. 23.9 | Author class definition.

```
1  // Fig. 23.10: 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 {
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. 23.10 | Book class definition.

Classes `Author` and `Book` define destructors that each display a message to indicate when an instance of either class is destroyed (Figs. 23.11 and 23.12, 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. 23.11: 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
```

Fig. 23.11 | `Author` member-function definitions. (Part 1 of 2.)

```
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    {
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. 23.11 | Author member-function definitions. (Part 2 of 2.)

```
1  // Fig. 23.12: 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
         shared_ptr< Author > authorPtr = weakAuthorPtr.lock()
25    {
```

Fig. 23.12 | Book member-function definitions. (Part 1 of 2.)

```
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. 23.12 | Book member-function definitions. (Part 2 of 2.)

Figure 23.13 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. 23.13: fig23_13.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
```

Fig. 23.13 | `shared_ptr`s cause a memory leak in circularly referential data. (Part 1 of 2.)

```
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
```

```
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. 23.13 | `shared_ptrs` cause a memory leak in circularly referential data. (Part 2 of 2.)

Memory Leak

At the end of `main`, the `shared_ptrs` 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 23.14 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_ptrs` 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_ptrs` 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

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. 23.14 | `weak_ptrs` used to prevent a memory leak in circularly referential data.

23.7 Technical Report 1

Technical Report 1 (TR1) describes proposed additions to the C++ Standard Library. Many of the libraries in TR1 will be accepted by the C++ Standards Committee but they are not considered part of the C++ standard until C++0x is finalized. The library additions provide solutions for many common programming problems. Most of the additions are based on 11 Boost libraries—the ones discussed in Section 23.4 and several other minor ones. Descriptions of the three additional TR1 libraries follow.

Visual Studio 2010 and recent versions of GNU C++ support most of TR1 already. Boost provides a compatibility layer that automatically falls back to the Boost implementation of each library if it was not supplied with the compiler.[10]

Many libraries didn't make it into TR1 due to time constraints. **Technical Report 2 (TR2)**, which will be released after C++0x, contains additional library proposals that weren't included in TR1. The release of TR2 will bring even more functionality to the standard library without having to wait for another new standard.

Unordered Associative Containers[11]

The Unordered Associative Containers library defines four new containers—`unordered_set`, `unordered_map`, `unordered_multiset` and `unordered_multimap`. These

10. Documentation for Boost.TR1: `www.boost.org/doc/libs/1_45_0/doc/html/boost_tr1.html`.
11. Matthew Austern, "A Proposal to Add Hash Tables to the Standard Library," Document Number N1456=03-0039, April 9, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1456.html`.

associative containers are implemented as hash tables. A **hash table** is split into sections sometimes called "**buckets**." A key is used to determine where to store an element in the container. The key is passed to a **hash function** which returns a `size_t`. The `size_t` returned by the hash function determines the "bucket" that the value is placed in. If two values are equal, so are the `size_t`s returned by the hash function. Multiple values can be placed in the same "bucket." You retrieve an element from the container using the key much as you do with a `set` or `map`. The key determines which "bucket" the value was placed in, then the "bucket" is searched for the value.

With **`unordered_set`** and **`unordered_multiset`**, the element itself is used as the key. **`unordered_map`** and **`unordered_multimap`** use a separate key to determine where to place the element—the arguments are passed as a `pair<const Key, Value>`. `unordered_set` and `unordered_map` require that all the keys used are unique; `unordered_multiset` and `unordered_multimap` don't enforce that restriction. The containers are defined in the `<unordered_set>` and `<undordered_map>` headers.

Mathematical Special Functions[12]
This library incorporates mathematical functions added to C99—the C standard published in 1999—that are missing in the C++ Standard. C99 supplies trigonometric, hyperbolic, exponential, logarithmic, power and special functions. This library adds those functions, among others, to C++ in the `<cmath>` header.

Increased Compatibility with C99[13]
C++ evolved from the C programming language. Most C++ compilers can also compile C programs, but there are some incompatibilities between the languages. The goal of this library is to increase compatibility between C++ and C99. Most of this library involves adding items to C++ headers to support C99 features—this is often accomplished by including the corresponding C99 headers.

23.8 C++0x

The C++ Standards Committee is currently revising the C++ Standard. The last standard was published in 1998. Work on the new standard, currently referred to as C++0x, began in 2003. The new standard, likely to be released in late 2011 or early 2012, includes the TR1 libraries and additions to the core language. Browse the C++0x section of the Deitel C++ Resource Center at `www.deitel.com/cplusplus/` and click **C++0x** in the **Categories** list to find current information on C++0x.

Standardization Process
The **International Organization for Standardization (ISO)** oversees the creation of international programming language standards, including those for C and C++. Every addition or change to the current C++ standard must be approved by the ISO/IEC JTC 1/SC 22 Working Group 21 (WG21), the committee that maintains the C++ standard. This com-

12. Walter E. Brown, "A Proposal to Add Mathematical Special Functions to the C++ Standard Library," Document Number N1422=03-0004, February 24, 2003, `std.dkuug.dk/jtc1/sc22/wg21/docs/papers/2003/n1422.html`.
13. P. J. Plauger, "Proposed Additions to TR-1 to Improve Compatibility With C99," Document Number N1568=04-0008, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1568.htm`.

mittee of volunteers from the C++ programming community meets twice a year to discuss issues pertaining to the standard. Smaller, unofficial meetings are held more frequently to consider proposals between official committee meetings. ISO requires at least 5 years between new drafts of a standard.

Goals for C++0x[14]

Bjarne Stroustrup, creator of the C++ programming language, has expressed his vision for the future of C++—the main goals for the new standard are to make C++ easier to learn, improve library building capabilities, and increase compatibility with the C programming language. He also provides an overview of C++0x's new features in his C++0x FAQ at `www2.research.att.com/~bs/C++0xFAQ.html`.

23.9 Core Language Changes

A listing of proposed changes to the core language can be found at `www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2869.html`. There are also links to the papers associated with each proposal. We briefly discuss some of the important core language changes that have been accepted into the working draft of the new standard. The number of proposals that make it into the working draft is likely to increase before the standard is finalized. The GNU C++ compiler has an optional C++0x mode which allows you to experiment with a number of the core language changes (`gcc.gnu.org/projects/cxx0x.html`). Visual Studio 2010 also supports some C++0x features—the Visual C++ Team Blog (`blogs.msdn.com/vcblog/`) contains updates on the status of C++0x in Visual Studio.

Rvalue *Reference*[15]

The ***rvalue* reference** type in C++0x allows you to bind an *rvalue* (temporary object) to a non-`const` reference. 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&` (now called an *lvalue* reference). An *rvalue* reference can be used to effectively implement move semantics—instead of being copied, the state of an object is moved, leaving the original with an empty value. For example, currently the following code creates a temporary `string` object and passes it to `push_back`, which then copies it into the vector.

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

If `push_back` were overloaded to take an *rvalue* reference, the storage allocated by the temporary `string` can be reused directly by the one in the `vector`. The temporary `string` will be destroyed anyway when the function returns, so there's no need for it to keep its value.

Rvalue references can also be used in "forwarding functions"—function objects that adapt a function to take fewer arguments (e.g., `std::bind1st` or function objects created using `Boost.Bind`). Normally, each reference parameter would need a `const` and non-`const` version to account for *lvalue*s, `const` *lvalue*s and *rvalue*s. With *rvalue* references you need only one forwarding function.

---

14. Bjarne Stroustrup, "The Design of C++0x," May 2005, `www.research.att.com/~bs/rules.pdf`.
15. Howard E. Hinnant, "A Proposal to Add an *rvalue* Reference to the C++ Language," October 19, 2006, Document Number N2118=06-0188, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2118.html`.

`static_assert`[16]

The **`static_assert` declaration** allows you to test certain aspects of the program at compile time. A `static_assert` declaration takes a constant integral expression and a `string` literal. If the expression evaluates to 0 (false), the compiler reports the error. The error message includes the `string` literal provided in the declaration. The `static_assert` declaration can be used at namespace, class or local scope.

The addition of `static_assert` makes learning C++ easier. The assertions can be used to provide more informative error messages when novices make common mistakes such as using the wrong type of argument in a function call or template instantiation. They're also useful in library development—incorrect usage of the library can be reported much more effectively.

Compatibility with New C99 Features

C++0x will incorporate many changes added in the 1999 C standard. These include changes to the preprocessor,[17] the addition of the `long long` integer type,[18] and imposing rules on extensions that add additional integer types[19] (for example, a 128-bit integer type). These changes allow modern C code to compile correctly as C++.

Delegating Constructors[20]

This feature allows a constructor to delegate to another of the class's constructors (i.e., call another of the class's constructors). This makes it easier to write overloaded constructors. Currently, an overloaded constructor must duplicate the code that is common to the other constructor. This leads to repetitive and error-prone code. A mistake in one constructor could cause inconsistency in object initialization. By calling another version of the constructor, the common code doesn't need to be repeated and the chance of error decreases.

Right Angle Brackets[21]

Currently, it's necessary to put a space between trailing right angle brackets (>) when using nested template types. Without the space, the compiler assumes that the two brackets are the right shift operator (>>). This means that writing `vector<vector<int>>` causes a compiler error. The statement would have to be written as `vector<vector<int> >`. Many novices stumble on this quirk of C++. In C++0x, the C++ compiler will recognize when >> is part of a template rather than the right-shift operator.

---

16. Robert Klarer, Dr. John Maddock, Beman Dawes and Howard Hinnant, "Proposal to Add Static Assertions to the Core Language," Document Number N1720, October 20, 2004, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1720.html`.
17. Clark Nelson, "Working Draft Changes for C99 Preprocessor Synchronization," Document Number N1653, July 16, 2004, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1653.htm`.
18. J. Stephen Adamczyk, "Adding the `long long` Type to C++," Document Number N1811, April 29, 2005, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1811.pdf`.
19. J. Stephen Adamczyk, "Adding Extended Integer Types to C++," Document Number N1988, April 19, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1988.pdf`.
20. Herb Sutter and Francis Glassborow, "Delegating Constructors," Document Number N1986=06-0056, April 6, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1986.pdf`.
21. Daveed Vandevoorde, "Right Angle Brackets," Document Number N1857=05-0017, January 14, 2005, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1757.html`.

Deducing the Type of Variable from Its Initializer[22]

This proposal defines new functionality for the keyword `auto`—it automatically determines variable types based on the initializer expression. `auto` can be used in place of long, complicated types that are unmanageable to type by hand. `auto` can also be used with `const` and `volatile` qualifiers. You can create pointers and references with `auto` as you would with the full type name. `auto` supports the declaration of multiple variables in one statement (e.g., `auto x = 1, y = 2`). The `auto` keyword is meant to save time, ease the learning process and improve generic programming. The following code creates a `vector` of instances of a hypothetical `Class< T >`.

```
vector< Class< T > > myVector;
vector< Class< T > >::const_iterator iterator = myVector.begin();
```

Using `auto`, the declaration of `iterator` can be written as

```
auto iterator = myVector.begin();
```

The type of `iterator` is `vector<Class<T>>::const_iterator`. You can also create two variables of the same type in one declaration. Both variables in

```
auto iteratorBegin = myVector.begin(), iteratorEnd = myVec-
tor.end();
```

are created with the type `vector<Class<T>>::const_iterator`. You can also use `auto` with `const` or `volatile` qualifiers and create pointers or references. The statement

```
const auto &iteratorRef = myVector.begin();
```

creates a `const` reference to a `vector<Class<T>>::const_iterator`. `auto` can save you a lot of time by automatically determining the type of the variable you're declaring—especially with complex template types, like those used in Section 22.6.

Variadic Templates[23]

Currently, each class or function template has a fixed number of template parameters. If you need a class or function template with different numbers of template parameters, you must 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 parameters. Many template libraries, such as `Boost.Bind`, `Boost.Tuple` and `Boost.Function`, include large amounts of duplicate code or make use of complex preprocessor macros to generate all the necessary template definitions. Variadic templates will make it easier to implement such libraries.

Template Aliases[24]

Libraries often use templates with many parameters to implement generic programming. There may be situations where it would be useful to be able to specify certain arguments

22. Jaakko Järvi, Bjarne Stroustrup and Gabriel Dos Reis, "Deducing the Type of Variable From Its Initializer Expression," Document Number N1984=06-0054, April 6, 2006, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1984.pdf`.
23. Douglas Gregor, Jaakko Järvi and Gary Powell, "Variadic Templates," Document Number N2080=06-0150, September 9, 2006, `www.osl.iu.edu/~dgregor/cpp/variadic-templates.pdf`.
24. Gabriel Dos Reis and Mat Marcus, "Proposal to Add Template Aliases to C++," Document # N1449-03-0032, April 7, 2003, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1449.pdf`.

for the template that remain consistent but still be able to vary the rest. This can be done with a template alias. A **template alias** is similar to a `typedef`—it introduces a name used to refer to a template. In a `typedef`, all the template parameters are specified. Using a template alias, certain parameters are specified and others may still vary. You can use a general-purpose template in a more specific role where many of the arguments are always the same by using a template alias to set the consistent parameters while still being able to vary those that change in each instantiation. For example, the following declares a template `MyStack` that uses `list<T>` as its underlying implementation instead of the `deque` used by default in `std::stack`.

```
template< typename T > using MyStack< T > =
   stack< T, list< T > >;
```

Initializer Lists for User-Defined Types[25]

Currently, initializer lists can be used only with arrays and `structs`. In C++0x, a class can define a constructor taking a parameter of type **`std::initializer_list<T>`**. An initializer list can then be used to initialize an object of the class, as in:

```
vector< int > second = { 4, 5, 6 }; // legal in C++0x
```

The initializer values are stored in an `initializer_list` object, which is passed to the class's constructor. All standard library container classes will be updated to have constructors taking an `initializer_list`.

Range-Based **`for`** *Statement*[26]

A common use of the `for` statement is to iterate over a container of elements. Currently, the syntax for built-in arrays and library containers is different—built-in arrays use an index or raw pointers, and container classes use iterators returned by the `begin` and `end` member functions. In addition to providing simpler syntax, the new **range-based `for` statement** allows you to use the same syntax for iterating over both arrays and containers. The following code iterates through a collection of `int` values.

```
for ( int &item : items ) // items can be an array or container
   item *= 2;
```

Lambda Expressions[27]

Many library functions receive function pointers or function objects as parameters. Currently, the functions or function objects must be defined before they can be passed to these library functions as arguments. **Lambda expressions** (or **lambda functions**) enable you to define function objects as they are being passed to a function. They are defined locally inside functions and can "capture" (by value or by reference) the local variables of the enclosing function then manipulate these variables in the lambda's body. Lambda

25. J. Stephen Adamczyk, Gabriel Dos Reis and Bjarne Stroustrup, "Initializer list WP wording (Revision 2)," Document Number N2531=08-0041, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2531.pdf`
26. Thorsten Ottosen, "Wording for range-based for-loop (revision 3)," Document Number N2934=07-0254, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2394.html`
27. Jaakko Järvi, John Freeman and Lawrence Crowl, "Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4)," Document Number N2550=08-0060, `www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf`

expressions are implemented in Visual Studio 2010 and GNU C++ 4.5. Figure 23.15 provides a simple lambda expression example that doubles the value of each element in an `int` array.

```
1  // Fig. 23.15: fig23_15.cpp
2  // Example of lambda expressions in C++0x.
3  #include <iostream>
4  #include <algorithm>
5  using namespace std;
6
7  int main()
8  {
9     const int size = 4; // size of array values
10    int values[ size ] = { 1, 2, 3, 4 }; // initialize values
11
12    // output each element multiplied by two
13    for_each( values, values + size,
14       []( int i ) { cout << i * 2 << endl; } );
15
16    int sum = 0; // initialize sum to zero
17
18    // add each element to sum
19    for_each( values, values + size,
20       [ &sum ]( int i ) { sum += i; } );
21
22    cout << "sum is " << sum << endl; // output sum
23 } // end main
```

```
2
4
6
8
sum is 10
```

Fig. 23.15 | Example of lambda expressions in C++0x.

Lines 9 and 10 declare and initialize a small array of integers. Lines 13–14 call the `for_each` algorithm on the elements of the `values` array. The third argument to `for_each` is a lambda expression. Lambdas begin with "lambda introducer" (`[]`), followed by a parameter list and function body. Return types can be inferred automatically if the body is a single statement of the form `return` *expression*;—otherwise, the return type is `void` by default. The lambda expression in line 14 receives an `int`, multiplies it by 2 and displays the result. The `for_each` algorithm passes each element of the array to the lambda.

The second call to the `for_each` algorithm (lines 19–20) calculates the sum of the array elements. The lambda introducer `[&sum]` indicates that this lambda expression is capturing the local variable `sum` by reference (note the use of the ampersand), so that the lambda can modify `sum`'s value. Without the ampersand, `sum` would be captured by value and the local variable would not be updated. The `for_each` algorithm passes each element of the array to the lambda, which adds the value to the `sum`. Line 22 then displays the value of `sum`.

23.10 Wrap-Up

In this chapter we discussed various aspects of the future of C++. We introduced the Boost C++ Libraries and described some of the most popular libraries.

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`.

We showed how to use the `Boost.Smart_ptr` library. 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_ptr`s to manage resources that require special destruction procedures. We also explained how `weak_ptr`s can be used to prevent memory leaks in circularly referential data.

We overviewed the upcoming revised standard, C++0x, discussing TR1 and the changes to the core language. We introduced the libraries accepted into TR1. We described the new core language features including the `auto` keyword, *rvalue* reference, improvements in compatibility with C99, initializer lists and lambda expressions. Remember that Boost, TR1 and C++0x are constantly changing—visit our Resource Centers to stay up to date with all three.

Summary

Section 23.2 Deitel Online C++ and Related Resource Centers

- Visit the C++ Boost Libraries Resource Center at `www.deitel.com/CPlusPlusBoostLibraries/` to find current information on the available libraries and new releases.
- Find current information on TR1 and C++0x (p. 937) in the **`C++0x`** category of the C++ Resource Center at `www.deitel.com/cplusplus/`.
- For more information on Visual C++, visit our Visual C++ Resource Center at `www.deitel.com/VisualCPlusPlus/`.

Section 23.3 Boost Libraries

- The Boost Libraries (p. 937) at `www.boost.org` provide free peer-reviewed C++ libraries.
- Boost libraries must conform to the C++ standard and use the C++ Standard Library—or other appropriate Boost libraries.
- A preliminary submission of each Boost library is posted in the Boost Sandbox Vault (p. 938).
- The review manager makes sure the code is ready for formal review, sets up the review schedule, reads all user reviews, and makes the final decision whether or not to accept the library.

Section 23.4 Boost Libraries Overview

- `Boost.Array` (p. 939) provides fixed-size arrays that support the STL container interface.
- `Boost.Bind` (p. 939) extends the functionality provided by the standard functions `bind1st` and `bind2nd`. It allows you to adapt functions that take up to nine arguments. It also makes it easy to reorder the arguments passed to the function.