Learning Boost C++ Libraries

Filled with dozens of working code examples that illustrate the use of over 40 popular Boost libraries, this book takes you on a tour of Boost, helping you to independently build the libraries from source and use them in your own code.

The first half of the book focuses on basic programming interfaces including generic containers and algorithms, strings, resource management, exception safety, and a miscellany of programming utilities that make everyday programming chores easy. Following a short interlude that introduces template metaprogramming and functional programming, the later chapters are devoted to systems programming interfaces, focusing on directory handling, I/O, concurrency, and network programming.

Who this book is written for

If you are a C++ programmer who has never used Boost libraries before, this book will get you up-to-speed with using them. Whether you are developing new C++ software or maintaining existing code written using Boost libraries, this hands-on introduction will help you decide on the right library and techniques to solve your practical programming problems.

What you will learn from this book

- Write efficient and maintainable code using expressive interfaces from Boost libraries
- Leverage a variety of flexible, practical, and highly efficient containers and algorithms beyond STL
- Solve common programming problems by applying a wide array of utility libraries
- Design and write portable multithreaded code that is easy to read and maintain
- Craft highly scalable and efficient TCP and UDP servers
- Build and deploy Boost libraries across a variety of popular platforms
- Use C++11 functionality and emulate C++11 language features in C++03 code

Prices do not include local sales tax or VAT where applicable


Packt Publishing
"Community Experience Distilled"
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 7 'Higher Order and Compile-time Programming'
- A synopsis of the book’s content
- More information on Learning Boost C++ Libraries
About the Author

Arindam Mukherjee is a senior principal software engineer at Symantec, Pune, India, where he is involved in the research and development of business continuity solutions for enterprises. He has used C++ extensively for developing large-scale distributed systems. He was a speaker at Dr. Dobb's Journal India Conference 2014 and is the organizer of regular meets for the Pune C++ and Boost Meetup. He believes that writing books and articles, speaking for interest groups, and engaging with the programming community are the best ways to develop a critical understanding of technology. He is also an amateur musician, dabbles in food photography, and loves profound discussions with his 4-year-old daughter, especially about dinosaurs and their diets.
Preface

Boost is not just a collection of useful, portable, generic C++ libraries. It is an important incubator for ideas and concepts that make their way to the ISO C++ Standard itself. If you are involved in the development of software written in C++, then learning to use the Boost libraries would save you from reinventing the wheel, improve the quality of your software, and very likely push up your productivity.

I first came across the Boost libraries a decade ago, while looking for a portable C++ regular expressions library. Over the next couple of days, porting Perl and Korn Shell text-processing code to C++ became a breeze, and I took an instant liking to Boost. In using many more Boost libraries to write software since then, I often found myself digging deep into the documentation, or asking questions on the mailing list and online forums to understand library semantics and nuances. As effective as that was, I always sorely missed a book that would get me started on the most useful Boost libraries and help me become productive faster. This is that book.

Boost has a wide array of libraries for solving various kinds of programming tasks. This book is a tutorial introduction to a selection of over of the most useful libraries from Boost to solve programming problems effectively. The chosen libraries represent the breadth of cross-cutting concerns from software development, including data structures and algorithms, text processing, memory management, exception safety, date and time calculations, file and directory management, concurrency, and file and network I/O, among others. You will learn about each library by understanding the kind of problems it helps solve, learning the fundamental concepts associated with it, and looking at a series of code examples to understand how the library is used. Libraries introduced earlier in this book are freely used in later examples, exposing you to the frequent synergies that occur in practice between the Boost libraries.
As a collection of peer-reviewed, open source libraries, Boost draws heavily from community expertise. I firmly believe that this book will give you a strong practical foundation in using the Boost libraries. This foundation will reflect in the quality of the software you write, and also give you the leverage to engage with the Boost community and make valuable contributions to it.

What this book covers

Chapter 1, Introducing Boost, discusses how to set up a development environment to use the Boost libraries. We cover different ways of obtaining Boost library binary packages, building them from source for different configurations, and using them in a development environment.

Chapter 2, The First Brush with Boost’s Utilities, explores a handful of Boost libraries for common programming tasks that include dealing with variant data types, handling command-line arguments, and detecting the configuration parameters of the development environment.

Chapter 3, Memory Management and Exception Safety, explains what is meant by exception safety, and shows how to write exception-safe code using the different smart pointer types provided by Boost and C++11.

Chapter 4, Working with Strings, explores the Boost String Algorithms library for performing various computations with character strings, the Boost Range library for elegantly defining subsequences, the Boost Tokenizer library to split strings into tokens using different strategies, and the Boost Regex library to search for complex patterns in text.

Chapter 5, Effective Data Structures beyond STL, deals with the Boost Container library focusing on containers not available in the C++ Standard Library. We see the Pointer Container library for storing dynamically-allocated objects in action, and use the Boost Iterator library to generate various value sequences from underlying containers.

Chapter 6, Bimap and Multi-index Containers, looks at bidirectional maps and multi-index containers—two nifty container templates from Boost.

Chapter 7, Higher Order and Compile-time Programming, delves into compile-time programming using Boost Type Traits and Template Metaprogramming libraries. We take a first look at Domain Specific Embedded Languages and use Boost Phoenix to build basic expression templates. We use Boost Spirit to build simple parsers using the Spirit Qi DSEL.
Chapter 8, *Date and Time Libraries*, introduces the Boost Date Time and Boost Chrono libraries to represent dates, time points, intervals, and periods.

Chapter 9, *Files, Directories, and IOStreams*, features the Boost Filesystem library for manipulating filesystem entries, and the Boost IOStreams library for performing type-safe I/O with rich semantics.

Chapter 10, *Concurrency with Boost*, uses the Boost Thread library and Boost Coroutine library to write concurrent logic, and shows various synchronization techniques in action.

Chapter 11, *Network Programming Using Boost Asio*, shows techniques for writing scalable TCP and UDP servers and clients using the Asio library.

Appendix, *C++11 Language Features Emulation*, summarizes C++11 move semantics and Boost's emulation of several C++11 features in C++03.
A number of Standard Library algorithms take callable entities called function objects (function pointers, functors, and so on) as parameters. They call these function objects on individual elements of containers to compute some value or perform some action. Thus, a part of the runtime logic of the algorithm is encapsulated in a function or functor and supplied as an argument to the algorithm. A function may also return function objects instead of data values. The returned function object can be applied on a set of parameters and may in turn return either a value or another function object. This gives rise to higher order transforms. This style of programming involving passing and returning functions is called higher order programming.

C++ templates enable us to write type generic code. Using templates, it is possible to execute branching and recursive logic at compile time and conditionally include, exclude, and generate code from simpler building blocks. This style of programming is called compile-time programming or template metaprogramming.

In the first part of this chapter, we will learn the applications of higher order programming in C++ using the Boost Phoenix Library and C++11 facilities like bind and lambda. In the next part of this chapter, we will learn C++ template metaprogramming techniques that execute at compile time to help generate more efficient and expressive code. In the last part of this chapter we look at domain-specific languages created within C++ by applying higher order programming techniques in combination with metaprogramming. The topics of this chapter are divided into the following sections:

- Higher order programming using Boost
- Compile-time programming using Boost
- Domain Specific Embedded Languages
Higher Order and Compile-time Programming

In this chapter, we will explore an alternate paradigm of programming, which is different from object-oriented and procedural programming and draws heavily from functional programming. We will also develop generic programming techniques that ultimately help us implement more efficient template libraries.

Higher order programming with Boost

Consider a type `Book` with three string fields: the ISBN, title, and author (for our purposes, assume that there is only one author). Here is how we can choose to define this type:

```cpp
1 struct Book
2 {
3   Book(const std::string& id,
4       const std::string& name,
5       const std::string& auth)
6     : isbn(id), title(name), author(auth)
7   {}
8
9   std::string isbn;
10  std::string title;
11  std::string author;
12};

13 bool operator< (const Book& lhs, const Book& rhs)
14 {  return lhs.isbn < rhs.isbn;  }
```

It is a `struct` with three fields and a constructor that initializes these three fields. The `isbn` field uniquely identifies the book and therefore is used to define an ordering of `Book` objects, using the overloaded `operator<` (line 14).

Now imagine that we have a list of these `Book` objects in a `std::vector`, and we want to sort these books. Thanks to the overloaded `operator<`, we can easily sort them using the Standard Library `sort` algorithm:

```cpp
1 #include <vector>
2 #include <string>
3 #include <algorithm>
4 #include <iostream>
5
6 // include the definition of struct Book
7
8 int main()
9 {
```

---

[250]
Chapter 7

```
10   std::vector<Book> books;
11   books.emplace_back("908..511..123", "Little Prince",
12       "Antoine St. Exupery");
13   books.emplace_back("392..301..109", "Nineteen Eighty Four",
14       "George Orwell");
15   books.emplace_back("872..610..176", "To Kill a Mocking Bird",
16       "Harper Lee");
17   books.emplace_back("392..301..109", "Animal Farm",
18       "George Orwell");
19
20   std::sort(books.begin(), books.end());
```

In the preceding code, we put four `Book` objects in the vector `books`. We do this by calling the `emplace_back` method (lines 11-18) rather than `push_back`. The `emplace_back` method (introduced in C++11) takes the constructor arguments for the stored type (`Book`) and constructs an object in the vector's layout rather than copying or moving in a pre-constructed object. We then sort the vector using `std::sort`, which ultimately uses the `operator<` for `Book` objects. Without this overloaded operator, `std::sort` would have failed to compile.

This is all great, but what if you wanted to sort the books in descending order of the ISBN? Or you could want to sort the books by their authors instead. Also, for two books with the same author, you might want to sort them further by their title. We will see a method to sort them this way in the next section.

### Function objects

There is a three-argument overload of `std::sort` algorithm that takes a function object for comparing two elements as the third argument. This function object should return true if the first argument appears before the second argument in the final ordering and false otherwise. So, even without an overloaded `operator<`, you can tell `std::sort` how to compare two elements and sort the vector. Here is how we do the sorting using an ordering function:

```
Listing 7.1: Passing functions to algorithms

1   bool byDescendingISBN(const Book& lhs, const Book& rhs)
2   {  return lhs.isbn > rhs.isbn;  }
3
4   ... 
5   std::vector<Book> books;
6   ...
7   std::sort(books.begin(), books.end(), byDescendingISBN);
```
The function `byDescendingISBN` takes const references to two books and returns true if the ISBN of the first book (`lhs`) is lexically greater than that of the second (`rhs`) and false otherwise. The signature of the function compatible with the function object that `std::sort` algorithm expects as its third argument. To sort the `books` vector in descending order, we pass to `std::sort`, a pointer to this function (line 7).

Function pointers are by no means the only callable entities you can pass around. A functor is a type that overloads the function call operator member (`operator()`). By applying or calling an instance of a functor on a set of arguments, you invoke the overloaded `operator()` member. In the following example, we define a functor to order books by author names, and in case of a tie with author names, by titles:

**Listing 7.2: Defining and passing functors to algorithms**

```
1 ...
2 struct CompareBooks
3 {
4   bool operator()(const Book& b1, const Book& b2) const {
5     return (b1.author < b2.author)
6     || (b1.author == b2.author
7       && b1.title < b2.title);
8   }
9 }
10 ...
11 ...
12 std::vector<Book> books;
13 ...
14 std::sort(books.begin(), books.end(), CompareBooks());
```

We define a functor called `CompareBooks` with an overloaded `operator()` that takes two `Book` objects to compare (line 4). It returns true if the name of the first book's author is lexicographically smaller than the name of second book's author. In case the authors of the two books are same, it returns true if the title of the first book is lexicographically smaller than that of the second. To use this functor as the sorting criterion, we pass a temporary instance of `CompareBooks` as the third argument of the `std::sort` algorithm (line 14). Functors like `CompareBooks`, that map one or more arguments to a Boolean truth value are called predicates.
A note on terminology

We use the term **function object** to refer to all callable entities that can be passed around and stored for later use by the application. These include function pointers and functors as well as other kinds of callable entities like unnamed functions or **lambdas**, which we will explore in this chapter.

A **functor** is simply a class or struct that defines an overloaded function call operator.

A function object that takes one or more arguments and maps them to a Boolean truth value is usually called a **predicate**.

The **arity** of a function object is the number of arguments it takes. A function with no arguments has 0-arity or is **nullary**, a function with one argument has 1-arity or is **unary**, a function with two arguments has 2-arity or is **binary**, and so on.

A **pure function** is a function whose return value depends solely on the values of the arguments passed to it and which has no side effects. Modifying states of objects not local to the function, performing I/O, or otherwise modifying the execution environment—all qualify as side effects.

Functors are especially useful when you want them to retain some state between calls. For example, imagine you have an unsorted list of names, and you just want to make a comma-separated list of all names, starting with a particular letter. Here is a way to do this:

Listing 7.3: Functors with states

```cpp
1 #include <vector>
2 #include <string>
3 #include <iostream>
4 #include <algorithm>
5
6 struct ConcatIfStartsWith {
7   ConcatIfStartsWith(char c) : startCh(c) {}
8
9   void operator()(const std::string& name) {
10      if (name.size() > 0 && name.at(0) == startCh) {
11         csNames += name + ", ";
12     }
13   }
```

We define a functor called `ConcatIfStartsWith` (line 6), which stores some state, namely the starting character to match (startCh) and a string to contain the comma-separated list of names (csNames). When the functor is invoked on a name, it checks whether it starts with the specified character, and if so, concatenates it to csNames (lines 10-11). We use the std::for_each algorithm to apply the ConcatIfStartsWith functor to each name in a vector of names (lines 30-31), looking for names starting with the letter G. The functor we pass is a temporary one (line 31), but we need a reference to it in order to access the concatenated string stored in it. The std::for_each algorithm actually returns a reference to the passed functor, which we then use to get the concatenated string. Here is the output, listing the names starting with G:

Guinnevere, Germaine, Gwynneth,

This illustrates an important point about functors; they are particularly useful when you want to maintain state that persists between successive calls to the function. They are also great if you need to use them at multiple places in your code. By naming them intuitively, their purpose can be made evident at the point of use:

```cpp
const auto& fe = std::for_each(names.begin(), names.end(),
                                   ConcatIfStartsWith('G'));
```
But sometimes, what a functor needs to do is trivial (for example, to check whether a number is even or odd). Often, we don't need it to maintain any state between calls. We may not even need to use it at multiple places. Sometimes, the functionality we are looking for may already be there in some form, maybe as a member function of the objects. In such cases, writing a new functor seems like overkill. C++11 introduced lambdas or unnamed functions to address precisely such cases.

**Lambdas – unnamed function literals**

The character string "hello" is a valid C++ expression. It has a well-defined type (const char[6]), can be assigned to variables of type const char*, and passed to functions that take arguments of type const char*. Likewise, there are numeric literals like 3.1415 or 64000U, Boolean literals like true and false, and so on. C++11 introduces **lambda expressions** for generating anonymous functions defined at the site, where they are invoked. Often, simply called **lambdas** (from Alonzo Church's λ-calculus), they consist of a function body not bound to a function name and are used to generate a function definition at any point in the lexical scope of a program, where you would expect to pass a function object. Let us first understand how this is done with the help of an example.

We have a list of integers, and we want to find the first odd number in the list using the std::find_if algorithm. The predicate passed to std::find_if is defined using a lambda.

**Listing 7.4: Using lambdas**

```cpp
#include <vector>
#include <algorithm>
#include <cassert>

int main() {
  std::vector<int> vec{2, 4, 6, 8, 9, 1};
  auto it = std::find_if(vec.begin(), vec.end(),
                         [](const int& num) -> bool
                         {  return num % 2 != 0; }
                         );
  assert(it != vec.end() && *it == 9);
}
```
Higher Order and Compile-time Programming

The lambda to compute whether a number is odd or even is a block of code passed as the third argument to `std::find_if` (lines 9-10). Let us look at the lambda in isolation to understand the syntax. First, consider what this function does; given an integer, it returns true if it is odd and false otherwise. So, we have an unnamed function that maps an `int` to a `bool`. The way to write this in lambda-land is as follows:

```cpp
[](const int& num) -> bool
{
    return num % 2 != 0;
}
```

We introduce the unnamed function with an empty pair of square brackets, and we describe the mapping by writing a parameter list like that of a conventional function, followed by an arrow and the return type. Following this, we write the body of the function just like you would for a normal function:

```cpp
[](const int& num) { return num % 2 != 0; }
```

The pair of square brackets, often called **lambda introducers**, need not be empty, as we will see shortly. There are several other variations possible with this syntax, but you can define a lambda using just this bit of syntax. The return type specification for lambdas is optional in simple cases, where the compiler can easily deduce the return type from the function body. Thus, we could have rewritten the lambda from the preceding example without the return type because the function body is really simple:

```cpp
[](const int& num) { return num % 2 != 0; }
```

**Lambda captures**

The lambda we defined in the previous example was a pure function without any state. In fact, how could a lambda conceivably store the state that persists between calls? Actually, lambdas can access local variables from the surrounding scope (in addition to global variables). To enable such an access, we can specify **capture clauses** in the lambda introducer to list which variables from the surrounding scope are accessible to the lambda and **how**. Consider the following example in which we filter out names longer than a user-specified length from a vector of names and return a vector containing only the shorter names:

**Listing 7.5: Lambdas with captures**

```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>

typedef std::vector<std::string> NameVec;

NameVec getNamesShorterThan(const NameVec& names,
```
Chapter 7

```cpp
size_t maxSize) {
    NameVec shortNames;
    std::copy_if(names.begin(), names.end(),
                 std::back_inserter(shortNames),
                 [maxSize](const std::string& name) {
                     return name.size() <= maxSize;
                 });
    return shortNames;
}
```

The `getNamesShorterThan` function takes two parameters: a vector called `names` and a variable `maxSize` that caps the size of strings to be filtered. It copies names shorter than `maxSize` from the `names` vector into a second vector called `shortNames`, using the `std::copy_if` algorithm from the standard library. We use a lambda expression (lines 12-14) to generate the predicate for `std::copy_if`. You can see that we name the `maxSize` variable from the surrounding lexical scope inside the square brackets (line 12), and access it inside the body of the lambda to compare the size of the passed string (line 13). This enables read-only access to the `maxSize` variable inside the lambda. If we wanted to potentially access any variable from the surrounding scope instead of a specific one, we could instead write the lambda with an equals sign in the square brackets; this would implicitly capture any variable used from the surrounding scope:

```cpp
[=](const std::string& name) {
    return name.size() <= maxSize;
}
```

You may want to modify a local copy of a variable from the surrounding scope, without affecting its value in the surrounding scope. To enable your lambda to do this, it must be declared as mutable:

```cpp
[=](const std::string& name) mutable -> bool {
    maxSize *= 2;
    return name.size() <= maxSize;
}
```

The `mutable` keyword trails the parameter list but appears before the return type if you specify one. This does not affect the value of `maxSize` in the surrounding scope.

You can also modify a variable from the surrounding scope inside a lambda. To do this, you must capture the variable by reference, by prefixing an ampersand to its name in the square brackets.
Here is listing 6.3 rewritten using a lambda:

**Listing 7.6: Reference captures in lambda**

```cpp
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>

int main() {
  std::string concat;
  char startCh = 'M';
  std::vector<std::string> names{"Meredith", "Guinnevere", "Mabel"
                      , "Myrtle", "Germaine", "Gwynneth", "Mirabelle"};

  std::for_each(names.begin(), names.end(),
                 [&concat, startCh](const std::string& name) {
                     if (name.size() > 0 && name[0] == startCh) {
                         concat += name + ", ";
                     }
                 });
  std::cout << concat << \n;
}
```

In the preceding example, we concatenate all names from the vector `names` that start with a specific character. The starting character is picked up from the variable `startCh`. The concatenated string is stored in the variable `concat`. We call `std::for_each` on the elements of the vector and pass a lambda, which explicitly captures `concat` as a reference (with a leading ampersand) and `startCh` as a read-only value from the surrounding scope (line 13). Thus, it is able to append to `concat` (line 15). This code prints the following output:

```
Meredith, Mabel, Myrtle, Mirabelle
```

In the latest revision of the C++ Standard, dubbed C++14, lambdas get a little niftier. You can write a **generic lambda** whose parameter types are deduced based on the context. For example, in C++14, you can write the call to `std::for_each` in the previous example, as follows:

```cpp
std::for_each(names.begin(), names.end(),
              [&concat, startCh](const auto& name) {
                  if (name.size() > 0 && name[0] == startCh) {
                      concat += name + ", ";
                  }
              });
```
The type of the argument to lambda is written as `const auto&`, and the compiler deduces it as `const std::string&` based on the type of elements in the iterated sequence.

### Delegates and closures

Let us suppose you are writing a high-level C++ API for reading incoming messages on a message queue. The client of your API must register for the types of messages it is interested in and pass a callback—a function object that will be invoked when messages of your interest arrive. Your API could be a member of a `Queue` class. Here is one possible API signature:

```cpp
class Queue
{
public:
    ...
    template <typename CallbackType>
    int listen(MsgType msgtype, CallbackType cb);
    ...
};
```

The `listen` member template takes two parameters: the message type `msgtype`, which identifies the messages of interest, and a callback function object `cb` that will be called when a new message arrives. Since we want the client to be able to pass function pointers, pointer to member functions, functors, as well as lambdas for the callback, we make `listen` a member template parameterized on the type of the callback. Of course, the callback should have a specific signature. Let us suppose it should be compatible with the signature of the following function:

```cpp
void msgRead(Message msg);
```

Here, `Message` is the type of messages read from the queue. The `listen` member template is a little too permissive because it can be instantiated with function objects that do not conform to the preceding signature. For a signature-incompatible callback, a compilation error occurs at the point where the callback is invoked inside `listen` rather than the point where the nonconforming callback is passed. This can make debugging the compiler errors more difficult.

The Boost.Function library and its C++11 incarnate `std::function` offer function object wrappers that are tailor-made to fix such problems. We can write the type of the function `msgRead` as `void (Message)`. The general syntax for the type of a function of arity N is as follows:

```cpp
return-type(param1-type, param2-type, ..., paramN-type)
```
Higher Order and Compile-time Programming

The more familiar **function pointer type** corresponding to the preceding **function type** would be:

```
return-type (*)(param1-type, param2-type, ..., paramN-type)
```

Thus, the type of a function `int foo(double, const char*)` would be:

```
int(double, const char*);
```

A pointer to `int` will be of type:

```
int (*)(double, const char*);
```

Using `std::function` with the appropriate function type, we can declare `listen` so that it accepts only function objects that conform to the correct signature:

```
#include <boost/function.hpp>

class Queue
{
  public:
    ...
    int listen(MsgType msgtype, boost::function<void(Message)> cb);
    ...
};
```

The callback is now declared to be of type `boost::function<void(Message)>`. You can now call `listen` with a pointer to a global function, a functor, or even a lambda, and it will only compile if the function object has a conforming signature. We could have used `std::function` in place of `boost::function` if we were using a C++11 compiler. On pre-C++11 compilers, `boost::function` supports signatures with up to ten arguments, while `std::function` does not have any such limitation as it uses C++11 *variadic templates*. For more features of `boost::function` and its differences from `std::function` (which are minor), you can refer to the online documentation.

Passing a nonstatic member function as a callback requires a little bit more work, because a non-static member must be called on an instance of its class. Consider the following class `MessageHandler` with a member `handleMessage`:

```
class MessageHandler
{
  public:
    ...
    void handleMessage(Message msg);
};
```
The `handleMessage` member function is implicitly passed a pointer to the `MessageHandler` object on which it is invoked as its first parameter; so its effective signature is:

```cpp
void(MessageHandler*, Message);
```

When we want to pass this as a callback to `Queue::listen`, we probably already know which object we want `handleMessage` to be called on, and it would be great if we could somehow attach that object instance too in the call to `listen`. There are a couple of ways in which this can be done.

The first method involves wrapping the call to `handleMessage` in a lambda and passing it to `listen`. The following snippet illustrates this:

**Listing 7.7: Member function callbacks using closures**

```cpp
MessageHandler *handler = new MessageHandler(...);
Queue q(...);
...
q.listen(msgType, [handler](Message msg)
    { handler->handleMessage(msg); });
```

Here, the second argument to `listen` is generated using a lambda expression, which also captures a pointer to the `handler` object from the surrounding scope. In this example, `handler` is a local variable in the calling scope, but the lambda captures it and binds it into the function object it generates. This function object is not invoked immediately on it but delayed until a message of interest is received on the queue, when it forwards the call to the `handleMessage` method on the `handler` object pointer.

The `handler` pointer is created in the calling scope but becomes indirectly accessible in another scope via the lambda capture. This is referred to as **dynamic scoping**, and functions of this kind that bind to variables in the lexical scope, in which they are created, are called **closures**. Of course, the `handler` pointer must still point to a valid `MessageHandler` object at the time when `handleMessage` is called on it, not just when the lambda is created.

More often than not, such lambdas would be generated from inside a member function, like a member function of the `MessageHandler` class and would capture the `this` pointer with some consequent syntactic simplifications:

**Listing 7.8: Capturing this-pointer in lambdas**

```cpp
class MessageHandler
{
  public:
```
In the preceding example, we create a closure using a lambda expression that captures the `this` pointer (line 6). The call to `handleMsg` inside the lambda automatically binds to the `this` pointer, just as it would in a member function. Callbacks, especially when bound to specific objects, as mentioned earlier, are sometimes called *delegates*.

The `boost::function`/`std::function` wrapper provides an effective and type-checked way of passing and returning function objects as callbacks or delegates. They are sometimes called polymorphic function wrappers because they completely abstract the type of the underlying callable entity (function pointer, functor, and so on) from the caller. Most implementations allocate memory dynamically though, so you should pay due diligence to assess their impact on runtime performance.

**Partial function application**

Given the Standard Library function `pow`:

```cpp
double pow(double base, double power);
```

Consider the effect of the line of code `x = pow(2, 3)`. When this line is encountered, the function `pow` is immediately called with two arguments, the values 2 and 3. The function `pow` computes 2 raised to 3 and returns the value 8.0, which is then assigned to `x`.

Now, say you have a list of numbers, and you want to put their cubes into another list. The Standard Library algorithm `std::transform` is a perfect fit for this. We just need to find the right functor to raise the numbers to their cubic power. The following functor takes a single numeric argument and raises it to a specific power, using the `pow` function:

```cpp
#include <cmath>

struct RaiseTo {
    RaiseTo(double power) : power_(power) {}

    double operator()(double base) const {
        return pow(base, power_);
    }

    double power_;

};
```
We could also have used a lambda expression to generate the function object, as shown in listing 7.7 and 7.8 in the last section. Using `RaiseTo` with the `std::transform` algorithm, the following code does the job:

```cpp
std::vector<double> nums, raisedToThree;
...  
std::transform(nums.begin(), nums.end(),
               std::back_inserter(raisedToThree),
               RaiseTo(3));
```

The core computation in `RaiseTo` is done by the `pow` function. The `RaiseTo` functor provides a way to fix the power through the constructor argument and a call signature compatible with what `std::transform` expects.

Imagine if you could do this in C++ without functors or lambdas. What if using the following *imaginary* syntax, you could do the same thing?

```cpp
std::transform(nums.begin(), nums.end(),
               std::back_inserter(raisedToThree),
               pow(_, 3));
```

It is as if you are passing the `pow` function with one of its two arguments fixed at 3 and asking the `transform` algorithm to fill in the blank; supply the number to raise to. The expression `pow(_, 3)` would have evaluated to a function object, taking one argument instead of 2. We essentially achieved this using the `RaiseTo` functor, but the Boost Bind library and its C++11 incarnate `std::bind` help us do this with less syntax. Formally, what we have just done is referred to as **partial function application**.

To create a partially applied function object for `pow` using `bind`, you would need to write:

```cpp
boost::bind(pow, _1, 3)
```

The preceding expression generates an unnamed functor which takes a single argument and returns its value raised to the power of 3, using the standard library function `pow`. The similarity with our imaginary syntax should be evident. The value to be cubed is passed as the sole argument of the generated functor and is mapped to the special placeholder `_1`. 
Listing 7.9: Using Boost Bind

```cpp
#include <boost/bind.hpp>

std::vector<double> nums, raisedToThree;
std::transform(nums.begin(), nums.end(),
               std::back_inserter(raisedToThree),
               boost::bind(pow, _1, 3));
```

If the generated functor takes more arguments, then they could be mapped to the placeholders \_2, \_3, and so on, based on their positions in the argument list. In general, the nth argument maps to the placeholder \_n. Boost Bind by default supports maximum nine positional placeholders (\_1 through \_9); std::bind might support more (varies from one compiler to the next), but you will need to access them from the std::placeholders namespace, using one of the following directives:

```cpp
using std::placeholders::_1;
using std::placeholders::_2;
// etc. OR
using namespace std::placeholders;
```

You may adapt functions by reordering their arguments without changing function arity to achieve a new functionality. For example, given the functor std::less that returns true if its first argument is less than its second argument, we can generate a functor, which returns true if its first argument is greater than its second argument by swapping the arguments. The following expression generates this:

```cpp
boost::bind(std::less<int>(), _2, _1)
```

Here, std::less<int> takes two arguments, and we generate a wrapper function object, which also takes two arguments but swaps their positions before passing them to std::less. We can directly call the generated functor in-place, like this:

```cpp
boost::bind(std::less<int>(), _2, _1)(1, 10)
```

We can safely assert that 1 is not greater than 10 but is, in fact, less:

```cpp
assert( std::less<int>()(1, 10) );
ansest( !boost::bind(std::less<int>(), _2, _1)(1, 10) );
```

Boost Bind is also useful for generating delegates, and other methods of generating delegates were illustrated in listing 7.7 and 7.8. Here is Listing 7.8 rewritten using boost::bind:

Listing 7.10: Generating delegates with Boost Bind

```cpp
class MessageHandler
{
  public:
```
void listenOnQueue(Queue& q, MessageType msgType) {
    q.listen(msgType, boost::bind(&MessageHandler::handleMsg,
                                  this, _1));
}

void handleMsg(Message msg) { ... }
};

We must bind a member function to an object instance. We do this by binding this
to the first argument of MessageHandler::handleMsg (lines 6-7). This technique
is generally useful for invoking member functions on each object in a collection.
Moreover, boost::bind / std::bind intelligently deal with objects, pointers,
smart pointers, and so on, so you do not need to write different binders, depending
on whether it is a copy of an object, a pointer, or a smart pointer. In the following
example, we take a vector of std::string objects, compute their lengths using the size
member function, and put them in a vector of lengths:

Listing 7.11: Generating delegates with Boost Bind

```cpp
#include <functional>
... std::vector<std::string> names{"Groucho", "Chico", "Harpo"};
std::vector<std::string::size_type> lengths;
using namespace std::placeholders;

std::transform(names.begin(), names.end(),
               std::back_inserter(lengths),
               std::bind(&std::string::size, _1));
```

The lengths are computed by calling the size member function on each std::string
object. The expression std::bind(&std::string::size, _1) generates an unnamed
functor, which calls the size member on the string object passed to it.

Even if names was a vector of pointers to std::string objects, or smart pointers,
the bind expression (line 9) would not need to change. The bind function takes its
parameters by value. Thus, in the preceding example, each string is copied into the
generated functor—a source of potential performance issue.

Another function template called boost::mem_fn and its Standard Library counterpart
std::mem_fn make it a tad easier to call member functions on objects and generate
delegates. The mem_fn function template creates a wrapper around pointers to class
members. For a member function $f$ of arity $N$ in class $X$, mem_fn($&X::f$) generates
a functor of arity $N+1$, whose first argument must be a reference, pointer, or smart
pointer to the object on which the member function is invoked.
We can write listing 7.11 to use mem_fn instead:

```cpp
#include <boost/mem_fn.hpp> // <functional> for std
...
std::transform(names.begin(), names.end(), 
std::back_inserter(lengths), 
boost::mem_fn(&std::string::size));
```

Because `std::string::size` is nullary, the functor generated by `boost::mem_fn` is unary and can be readily used with `transform`, without additional binding. The savings are in not having to write the `_1` placeholder, and thus have less syntactic noise.

When we generate a function object using `bind`, it does not immediately check whether the type and number of arguments match the signature of the function being bound to. Only when the generated function object is invoked, does the compiler detect parameter type and arity mismatch:

```cpp
std::string str;
auto f = boost::bind(&std::string::size, 5); // binds to literal 5
auto g = boost::bind(&std::string::size, _1, 20); // binds two args
```

For example, the preceding code would compile even though you cannot call the `size` member function of `std::string` on a numeric literal 5 (line 2). Nor does the `size` member function take an additional numeric argument (line 3). But as soon as you try to call these generated function objects, you will get errors due to type and arity mismatch:

```cpp
f(); // error: operand has type int, expected std::string
g(str); // error: std::string::size does not take two arguments
```

Binding member functions that are overloaded requires more syntactic effort. Generating functions of even moderate complexity with `bind` is an exercise in nesting `bind`s, which more often than not produces unmaintainable code. In general, with the availability of C++11 lambda and its further refinement in C++14, lambdas rather than `bind` should be the preferred mechanism of generating unnamed functors. Use `bind` only when it makes your code more expressive than a lambda can.
Compile-time programming with Boost

Templates allow us to write C++ code that is independent of specific types of
operands and can thus work unchanged with a large family of types. We can create
both function templates and class templates (or struct templates), which take
type parameters, nontype parameters (like constant integers), as well as template
parameters. When a specialization of a class template is instantiated, member functions
that are not directly or indirectly called are never instantiated.

The power of C++ templates goes beyond the ability to write generic code though.
C++ templates are a powerful computation subsystem using which we can introspect
C++ types, glean their properties, and write sophisticated recursive and branching
logic that executes at compile time. Using these capabilities, it is possible to define
generic interfaces to implementations that are highly optimized for each type they
operate upon.

Basic compile-time control flow using templates

In this section, we briefly look at branching and recursive logic generated
using templates.

Branching

Consider the function template boost::lexical_cast, introduced in Chapter 2, The
First Brush with Boost's Utilities. To convert a string to a double, we would write
code like the following:

```cpp
std::string strPi = "3.141595259";
double pi = boost::lexical_cast<double>(strPi);
```

The primary template of lexical_cast is declared this way:

```cpp
template <typename Target, typename Source>
Target lexical_cast(const Source&);
```
The default implementation of `lexical_cast` (called the **primary template**) writes the source object to a memory buffer via an interface like `ostringstream` and reads back from it via another interface like `istringstream`. This conversion may incur some performance overhead but has an expressive syntax. Now let us suppose that for a particularly performance-intensive application, you want to improve the performance of these string-to-double conversions, but do not want to replace `lexical_cast` with some other function calls. How would you do it? We can create an **explicit specialization** of the `lexical_cast` function template to perform a branching at compile time based on the types involved in the conversion. Since we want to override the default implementation for `string` to `double` conversions, this is how we would write the specialization:

**Listing 7.12: Explicit specialization of function templates**

```cpp
namespace boost {
  template <>
  double lexical_cast<double, std::string>(
      const std::string& str)
  {
    const char *numstr = str.c_str();
    char *end = nullptr;
    double ret = strtod(numstr, &end);

    if (end && *end != '\0') {
      throw boost::bad_lexical_cast();
    }

    return ret;
  }
} // boost
```

The `template` keyword with an empty argument list (`template<>`) indicates that this is a specialization for specific type arguments (line 2). The `template identifier` `lexical_cast <double, std::string>` lists the specific types for which the specialization takes effect (line 3). With this specialization available, the compiler invokes it whenever it sees code like this:

```cpp
std::string strPi = "3.14159259";
double pi = boost::lexical_cast<double>(strPi);
```

Note that it is possible to **overload function templates** (not just functions). For example:

```cpp
template<typename T> void foo(T);  // 1
template<typename T> void foo(T*); // 2
template<typename T> T foo(T, T);  // 3
```
void foo(int);                        // 4
template<> void foo<double>(double);  // 5

int x;
foo(&x);   // calls 2
foo(4, 5); // calls 3
foo(10);   // calls 4
foo(10.0); // calls 5

In the preceding example, foo is a function template (1) that is overloaded (2 and 3). The function foo itself is overloaded (4). The function template foo (1) is also specialized (5). When the compiler encounters a call to foo, it first looks for a matching non-template overload, failing which it looks for the most specialized template overload. In the absence of a matching specialized overload, this would simply resolve to the primary template. Thus, the call to foo(&x) resolves to template<typename T> void foo(T*). If such an overload was not present, it would resolve to template<typename T> void foo(T).

It is possible to create specializations for class templates too. In addition to explicit specializations, which specialize a class template for a fixed set of type and non-type arguments, we can also create partial specializations of class templates that specialize a class template for a family or category of types:

```cpp
template <typename T, typename U>
class Bar { /* default implementation */ };

template <typename T>
class Bar<T*, T> { /* implementation for pointers */ };  
```

In the preceding example, the primary template Bar takes two type arguments. We create a partial specialization for Bar for those cases, where the first of these two arguments is a pointer-type and the second argument is the pointer-type for the first. Thus, instantiating Bar<int, float> or Bar<double, double*> will instantiate the primary template, but Bar<float*, float>, Bar<Foo*, Foo>, etc. will instantiate the partially specialized template. Note that functions cannot be partially specified.

**Recursion**

Recursion using templates is best illustrated using an example of calculating factorials at compile time. Class templates (as well as function templates) can take integer arguments as long as the values are known at compile time.
Listing 7.13: Compile-time recursion using templates

```
1 #include <iostream>
2
3 template <unsigned int N>
4 struct Factorial
5 {
6   enum {value = N * Factorial<N-1>::value};
7   
8   template <>
9   struct Factorial<0>
10   {
11     enum {value = 1}; // 0! == 1
12   };
13
14   int main()
15   {
16     std::cout << Factorial<8>::value << '
'; // prints 40320
17   }
```

The primary template for calculating factorials defines a compile-time constant enum value. The value enum in Factorial<N> contains the value of the factorial of N. This is calculated recursively by instantiating the Factorial template for N-1 and multiplying its nested value enum with N. The stopping condition is provided by the specialization of Factorial for 0. These calculations happen at compile time, as the Factorial template gets instantiated with successively smaller arguments until Factorial<0> stops further instantiation. Thus, the value 40320 is computed completely at compile time and baked into the binary that is built. For example, we could have written the following and it would have compiled and generated an array of 40320 integers on the stack:

```
int arr[Factorial<8>::value]; // an array of 40320 ints
```

Boost Type Traits

The Boost Type Traits library provides a set of templates used to query types for properties and generate derivative types at compile time. They are useful in generic code, that is, code which uses parameterized types, for purposes such as choosing an optimal implementation based on the properties of a type parameter.
Consider the following template:

```cpp
#include <iostream>

template <typename T>
struct IsPointer {
    enum { value = 0 };  
};

template <typename T>
struct IsPointer <T*> {
    enum { value = 1 };  
};

int main() {
    std::cout << IsPointer<int>::value << '
';
    std::cout << IsPointer<int*>::value << '
';
}
```

The `IsPointer` template has a nested enum called `value`. This is set to 0 in the primary template. We also define a partial specialization of `IsPointer` for pointer-type arguments and set the nested `value` to 1. How is this class template useful? For any type `T`, `IsPointer<T>::value` is 1 if and only if `T` is a pointer-type and 0 otherwise. The `IsPointer` template maps its type argument to a compile-time constant value 0 or 1, which can be used for further branching decisions at compile time.

The Boost Type Traits library is chock full of such templates (including `boost::is_pointer`) that can glean information about types and also generate new types at compile time. They can be used for selecting or generating the optimal code for the types at hand. Boost Type Traits was accepted for the C++ TR1 release in 2007 and as of C++11, there is a Type Traits library in the Standard Library.

Each type trait is defined in its own header so that you can include only those type traits that you need. For example, `boost::is_pointer` would be defined in `boost/type_traits/is_pointer.hpp`. The corresponding `std::is_pointer` (introduced in C++11) is defined in the standard header `type_traits`, where there being no separate standard header for it. Each type trait has an embedded type called `type`, and in addition, it may have a member `value` of type `bool`. Here is an example of using a few type traits.

**Listing 7.14: Using type traits**

```cpp
#include <boost/type_traits/is_pointer.hpp>
#include <boost/type_traits/is_array.hpp>
```
3 #include <boost/type_traits/rank.hpp>
4 #include <boost/type_traits/extent.hpp>
5 #include <boost/type_traits/is_pod.hpp>
6 #include <string>
7 #include <iostream>
8 #include <cassert>
9
10 struct MyStruct {
11   int n;
12   float f;
13   const char *s;
14  }
15
16 int main()
17 {
18   typedef int* intptr;
19   std::cout << "intptr is "
20     << (boost::is_pointer<intptr>::value ?"true" : "false")
21     << "pointer type\n";
22 // introspect arrays
23   int arr[10], arr2[10][15];
24   if (boost::is_array<decltype(arr)>::value) {
25     assert(boost::rank<decltype(arr)>::value == 1);
26     assert(boost::rank<decltype(arr2)>::value == 2);
27     assert(boost::extent<decltype(arr)>::value == 10);
28     assert(boost::extent<decltype(arr2)>::value == 10);
29     assert((boost::extent<decltype(arr2), 1>::value) == 15);
30     std::cout << "arr is an array\n";
31   }
32
33 // POD vs non-POD types
34   std::cout << "MyStruct is "
35     << (boost::is_pod<MyStruct>::value ?"true" : "false")
36     << "pod type." << '\n';
37   std::cout << "std::string is "
38     << (boost::is_pod<std::string>::value ?"true" : "false")
39     << "pod type." << '\n';
40 }

In this example, we use a number of type traits to query information about types. We define a type intptr as an integer pointer (line 18). Applying boost::is_pointer to intptr yields true (line 20).
The `decltype` specifier used here was introduced in C++ 11. It generates the type of the expression or entity it is applied to. Thus, `decltype(arr)` (line 24) yields the declared type of arr, including any `const` or `volatile` qualifiers. It is a useful means of computing the type of an expression. We apply the `boost::is_array` trait to an array type, which obviously yields true (line 24). To find the number of dimensions or the rank of an array, we use the trait `boost::rank` (lines 25 and 26). The rank of `arr[10]` is 1 (line 25), but the rank of `arr2[10][15]` is 2 (line 26). The `boost::extent` trait is used to find the extent of an array's rank. It must be passed the array's type and rank. If the rank is not passed, it defaults to 0 and returns the extent for one-dimensional arrays (line 27) or the zeroth dimension of multi-dimensional arrays (line 28). Otherwise, the rank should be explicitly specified (line 29).

The `boost::is_pod` trait returns whether a type is a Plain Old Data type or not. It returns true for a simple struct without any constructors or destructors like `MyStruct` (line 34) and false for `std::string`, which is obviously not a POD type (line 38).

As mentioned before, there is also an embedded type in these traits called `type`. This is defined as `boost::true_type` or `boost::false_type`, depending on whether the trait returned true or false. Now consider that we are writing a generic algorithm to copy arrays of arbitrary objects into an array on the heap. For POD-types, a shallow copy or `memcpy` of the whole array is good enough, while for non-POD types, we need to perform element by element copies.

**Listing 7.15: Leveraging type traits**

```cpp
#include <boost/type_traits/is_pod.hpp>
#include <cstring>
#include <iostream>
#include <string>

struct MyStruct {
  int n; float f;
  const char *s;
};

template <typename T, size_t N>
T* fastCopy(T(&arr)[N], boost::true_type podType) {
  std::cerr << "fastCopy for POD\n";
  T *cpyarr = new T[N];
  memcpy(cpyarr, arr, N*sizeof(T));
  return cpyarr;
}
```
Higher Order and Compile-time Programming

20
template <typename T, size_t N>
T* fastCopy(T(&arr)[N], boost::false_type nonPodType)
{
24   std::cerr << "fastCopy for non-POD\n";
25   T *cpyarr = new T[N];
26   std::copy(&arr[0], &arr[N], &cpyarr[0]);
27   return cpyarr;
29 }
30
31 template <typename T, size_t N>
T* fastCopy(T(&arr)[N])
33 {
34   return fastCopy(arr, typename boost::is_pod<T>::type());
35 }
36
37 int main()
38 {
39   MyStruct podarr[10] = {};
40   std::string strarr[10];
41   auto* cpyarr = fastCopy(podarr);
42   auto* cpyarr2 = fastCopy(strarr);
43   delete [] cpyarr;
44   delete [] cpyarr2;
46 }

The fastCopy function template creates a copy of the array on the heap (lines 31-35). We create two overloads of it: one for copying POD-types (lines 11-12) and the other for copying non-POD types (lines 21-22), by adding a second parameter of type boost::true_type in the first case and boost::false_type in the second case. We create two arrays: one of the POD-type MyStruct and the other of the non-POD type std::string (lines 42-43). We call fastCopy on both, which are resolved to the one argument overload (line 32). This forwards the call to the two argument overloads of fastCopy, passing an instance of boost::is_pod<T>::type as the second argument (line 34). This automatically routes the call to the correct overload, depending on whether the stored type T is POD-type or not.

There are many, many more type traits than we can cover in the scope of this book. You have type traits to check whether one type is a base class of another (boost::is_base), whether a type is copy constructible (boost::is_copy_constructible), has specific operators (for example, boost::has_pre_increment), is same as another type (boost::is_same), and so on. The online documentation is a good place to go dig traits and see which ones fit a job at hand.
SFINAE and enable_if / disable_if

Each time a compiler encounters a call to a function with the same name as a function template, it creates an overload resolution set of matching template and non-template overloads. The compiler deduces template arguments as needed to determine which function template overloads (and specializations thereof) qualify, and the qualifying template overloads are instantiated in the process. If substitution of the deduced type arguments in the template's argument list or the function parameter list causes an error, this does not cause the compilation to abort. Instead, the compiler removes the candidate from its overload resolution set. This is referred to as Substitution Failure Is Not An Error or SFINAE. The compiler only flags an error if, at the end of the process, the overload resolution set is empty (no candidates) or has multiple equally good candidates (ambiguity).

Using a few clever tricks involving compile-time type computation, it is possible to leverage SFINAE to conditionally include templates or exclude them from the overload resolution set. The most succinct syntax to do this is provided by the boost::enable_if / boost::disable_if templates that are part of the Boost Utility library.

Let us write a function template to copy an array of elements into another array. The signature of the primary template is as follows:

```cpp
template <typename T, size_t N>
void copy(T (&lhs)[N], T (&rhs)[N]);
```

Thus, you pass two arrays of same size storing the same type of elements, and the elements of the second arguments are copied into the first array in the correct order. We also assume that the arrays never overlap; this keeps the implementation simple. Needless to say this is not the most general setting in which such an assignment can take place, but we will relax some of these restrictions a little later. Here is a generic implementation for this template:

```cpp
1 template <typename T, size_t N>
2 void copy(T (&lhs)[N], T (&rhs)[N])
3 {
4   for (size_t i = 0; i < N; ++i) {
5     lhs[i] = rhs[i];
6   }
7 }
```
The first opportunity for optimization here is when T is a POD-type and a bitwise copy is good enough and possibly faster. We will create a special implementation for POD-types and use SFINAE to choose this implementation only when we are dealing with arrays of POD-types. Our technique should exclude this overload from the overload set when dealing with non-POD type arrays. Here is the special implementation for POD-types:

```cpp
1 // optimized for POD-type
2 template <typename T, size_t N>
3 void copy(T (&lhs)[N], T (&rhs)[N])
4 {
5   memcpy(lhs, rhs, N*sizeof(T));
6 }
```

If you noticed, the two implementations have identical signature and obviously cannot coexist. This is where the `boost::enable_if` template comes in. The `boost::enable_if` template takes two parameters: a type T and a second type E, which defaults to `void`. `enable_if` defines an embedded type called `type`, which is typedef'd to E only when T has an embedded type called `type` and T::`type` is `boost::true_type`. Otherwise, no embedded type is defined. Using `enable_if`, we modify the optimized implementation:

```cpp
Listing 7.16: Using enable_if
#include <boost/utility/enable_if.hpp>
#include <boost/type_traits/is_pod.hpp>

// optimized for POD-type
template <typename T, size_t N>
  typename boost::enable_if<boost::is_pod<T>::type>
  copy(T (&lhs)[N], T (&rhs)[N])
  {
    memcpy(lhs, rhs, N*sizeof(T));
  }
```

The `typename` keyword is required because otherwise the compiler has no way of knowing whether the expression `boost::enable_if<boost::is_pod<T>::type` names a type or a member.

If we now instantiate an array of a non-POD type, it will resolve to the default implementation:

```cpp
std::string s[10], s1[10];
copy(s1, s);  // invokes the generic template
```
The call to \texttt{copy} causes the compiler to instantiate both templates but \texttt{boost::is\_pod<std::string>::type} is \texttt{boost::false\_type}. Now \texttt{enable\_if<false\_type>} does not have a nested type as required by the return type specification of the version of \texttt{copy} optimized for POD-arrays. Therefore, there is a substitution failure, and this overload is removed from the overload resolution set, and the first or generic implementation is invoked. Now consider what happens in the following case, where we try to copy an array of POD-types (double):

\begin{verbatim}
  double d[10], d1[10];
  copy(d1, d);
\end{verbatim}

In the current state of affairs, the POD-optimized version will no longer encounter a substitution failure, but the default implementation would also be signature-compatible with this call. Thus, there would be ambiguity and this would result in a compiler error. To fix this, we would have to make sure that the generic implementation excuses itself from the overload set this time. This is done using \texttt{boost::disable\_if} (which is really \texttt{boost::enable\_if} negated) in the return type of the generic implementation.

**Listing 7.17: Using disable\_if**

\begin{verbatim}
1 template <typename T, size_t N>
2 typename boost::disable\_if<boost::is\_pod<T>>::type
3   copy(T (&lhs)[N], T (&rhs)[N])
4 {
5   for (size_t i = 0; i < N; ++i) {
6     lhs[i] = rhs[i];
7   }
8 }
\end{verbatim}

When \(T\) is a POD-type, \texttt{is\_pod<T>::type} is \texttt{boost::true\_type}. \texttt{boost::disable\_if<true\_type>} does not have a nested type and thus a substitution failure occurs with the generic implementation. This way, we build two mutually exclusive implementations that are correctly resolved at compile time.

We can also use the \texttt{boost::enable\_if\_c<>} template which takes a Boolean parameter instead of a type. \texttt{boost::enable\_if\_c<true>} has an embedded type, while \texttt{boost::enable\_if\_c<false>} does not. With these, the return type in listing 7.17 would look like this:

\begin{verbatim}
1 template <typename T, size_t N>
2 typename boost::disable\_if\_c<boost::is\_pod<T>::value>::type
\end{verbatim}

The Standard Library, as of C++11, has \texttt{std::enable\_if} only, and it behaves like \texttt{boost::enable\_if\_c}, taking a Boolean argument rather than a type. It is available from the standard header \texttt{type\_traits}.
The Boost Metaprogramming Library (MPL)

The **Boost Metaprogramming Library**, MPL for short, is a general purpose library for template metaprogramming. It is ubiquitous in the Boost codebase, and most libraries use some metaprogramming facility from MPL. Some libraries like Phoenix, BiMap, MultIndex, and Variant use it very heavily. It is used heavily for type manipulation and optimization through conditional selection of specific template implementations. This section is a short overview of some of the concepts and techniques involving MPL.

**Metafunctions**

The heart of the MPL library is a **metafunction**. Formally, a metafunction is either a class template with only type parameters or a class, which exposes a single embedded type called `type`. In effect, type parameters if any are analogous to parameters to a function and the embedded `type`, which is computed at compile time based on the parameters, is analogous to the return value of a function.

Type traits provided by Boost Type Traits library are first-class metafunctions. Consider the `boost::add_pointer` type trait:

```cpp
template <typename T>
struct add_pointer;
```

The type `add_pointer<int>::type` is `int*`. The `add_pointer` template is a unary metafunction with a single type parameter and an embedded type called `type`.

Sometimes, the effective result of a type computation is numeric – case in point `boost::is_pointer<T>` (Boolean truth value) or `boost::rank<T>` (a positive integer). In such cases, the embedded `type` will have a static member called `value` containing this result, and it will also be directly accessible from the metafunction as a non-type member called `value`. Thus, `boost::is_pointer<T>::type::value` and `boost::is_pointer<T>::value` are both valid, the latter being more concise.

**Using MPL metafunctions**

The MPL working in conjunction with Boost Type Traits makes a lot of metaprogramming jobs easy. For this, the MPL provides a number of metafunctions to compose existing metafunctions together.
Like type traits, MPL facilities are partitioned into independent, highly granular header files. All metafunctions are in the boost::mpl namespace. We can compose unnamed metafunctions together into composite metafunctions using the MPL library. This is not unlike lambdas and bind at runtime. The following snippet uses boost::mpl::or_ metafunction to check whether a type is either an array or a pointer:

**Listing 7.18: Using MPL metafunctions**

```cpp
#include <boost/mpl/or.hpp>
#include <boost/type_traits.hpp>

if (boost::mpl::or_<
    boost::is_pointer<int*>,
    boost::is_array<int*>
> ::value) {
    std::cout << "int* is a pointer or array type\n";
}

if (boost::mpl::or_<
    boost::is_pointer<int[]>,
    boost::is_array<int[]>
> ::value) {
    std::cout << "int* is a pointer or array type\n";
}
```

The boost::mpl::or_ metafunction checks whether any of its argument metafunctions evaluates to true. We can create our own reusable metafunction that packages the preceding logic by using a technique called metafunction forwarding:

**Listing 7.19: Creating your own metafunction**

```cpp
#include <boost/mpl/or.hpp>
#include <boost/type_traits.hpp>

template <typename T>
struct is_pointer_or_array
    : boost::mpl::or_<boost::is_pointer<T>,
    boost::is_array<T>>
{};
```

We combine the existing type trait metafunctions using boost::mpl::or_ and inherit from the composed entity, as shown in the preceding listing (line 6). We can now use is_pointer_or_array like any type trait.
Sometimes, we need to pass numeric arguments, which are clearly non-type, to metafunctions. For example, to compare whether the size of a type \( T \) is smaller than that of another type \( U \), we ultimately need to compare two numeric sizes. Let us write the following trait to compare the size of two types:

\[
\text{template } \langle \text{typename } T, \text{typename } U \rangle \text{ struct is_smaller;}
\]

\text{is_smaller\text{\( T, U \)}}\cdot \text{value will be true if and only if } \text{sizeof}(T) \text{ is less than } \text{sizeof}(U), \text{ and will be false otherwise.}

\textbf{Listing 7.20: Using integral wrappers and other metafunctions}

\begin{verbatim}
1 \#include <boost/mpl/and.hpp>
2 \#include <boost/mpl/int.hpp>
3 \#include <boost/mpl/integral_c.hpp>
4 \#include <boost/mpl/less.hpp>
5 \#include <iostream>
6 namespace mpl = boost::mpl;
7
8 template <typename L, typename R>
9 struct is_smaller : mpl::less<
10   mpl::integral_c<size_t, sizeof(L)>
11             , mpl::integral_c<size_t, sizeof(R)>
12 >;
13
14 int main()
15 {
16   if (is_smaller<short, int>::value) {
17     std::cout << "short is smaller than int\n";
18   } else { ... }
19 }
\end{verbatim}

MPL provides a metafunction \texttt{boost::mpl::integral\_c} to wrap integral values of a specified type (size\_t, short, etc.). We use it to wrap the sizes of the two types. The \texttt{boost::mpl::less} metafunction compares the two sizes and its nested \texttt{value} is set to true only if the first argument is numerically less than the second. We can use it like any other trait.

We will now try to write something slightly less trivial. We want to write a function to assign arrays. Here is the function template signature:

\[
\text{template } \langle \text{typename } T, \text{size\_t } M, \text{typename } S, \text{size\_t } N \rangle \text{ void arrayAssign}(T(&lhs)[M], S(&rhs)[N]);
\]
The type \( T(\&)[M] \) is a reference to an array of \( M \) elements of type \( T \); likewise for \( S(\&)[N] \). We want to assign the second argument \( \text{rhs} \) to the first argument \( \text{lhs} \).

You can assign an array of type \( S[] \) to an array of type \( T[] \) as long as \( S \) and \( T \) are the same types, or the conversion from \( S \) to \( T \) is allowed and does not cause loss of information. Also, \( M \) must not be smaller than \( N \). We will define a trait \( \text{is\_array\_assignable} \) which captures these constraints. Thus, \( \text{is\_array\_assignable}<T(\&)[M], S(\&)[N]>::\text{value} \) will be true only if the preceding constraints are met.

First, we need to define three helper metafunctions: \( \text{is\_floating\_assignable} \), \( \text{is\_integer\_assignable} \), and \( \text{is\_non\_pod\_assignable} \). The \( \text{is\_floating\_assignable}<T, S> \) metafunction checks whether it is possible to assign a numeric value of type \( S \) to a floating point type \( T \). The \( \text{is\_integer\_assignable}<T, S> \) metafunction checks whether both \( T \) and \( S \) are integers, and an assignment for \( T \) and \( S \) does not cause any potential loss or narrowing. Thus, signed integers cannot be assigned to unsigned integers, unsigned integers can only be assigned to larger signed integer types, and so on. The \( \text{is\_non\_pod\_assignable}<T, S> \) trait checks whether at least one of \( S \) and \( T \) is non-POD type and whether an assignment operator from \( S \) to \( T \) exists.

We will then define \( \text{is\_array\_assignable} \) using these and other metafunctions.

**Listing 7.21: Defining useful type traits using MPL**

```cpp
#include <boost/type_traits.hpp>
#include <type_traits>
#include <boost/mpl/and.hpp>
#include <boost/mpl/or.hpp>
#include <boost/mpl/not.hpp>
#include <boost/mpl/greater.hpp>
#include <boost/mpl/greater_equal.hpp>
#include <boost/mpl/equal.hpp>
#include <boost/mpl/if.hpp>
#include <boost/mpl/integral_c.hpp>
#include <boost/utility/enable_if.hpp>
#include <iostream>

namespace mpl = boost::mpl;

template <typename T, typename S>
struct is_larger
    : mpl::greater<mpl::integral_c<size_t, sizeof(T)>,
                   mpl::integral_c<size_t, sizeof(S)>>
{};
```
Higher Order and Compile-time Programming

21 template <typename T, typename S>
22 struct is_smaller_equal
23   : mpl::not_<is_larger<T, S>>
24 {};
25
26 template <typename T, typename S>
27 struct is_floating_assignable
28   : mpl::and_<
29     boost::is_floating_point<T>
30     , boost::is_arithmetic<S>
31     , is_smaller_equal<S, T>
32   >
33 {};
34
35 template <typename T, typename S>
36 struct is_integer_assignable
37   : mpl::and_<
38     boost::is_integral<T>
39     , boost::is_integral<S>
40     , is_smaller_equal<S, T>
41     , mpl::if_<boost::is_signed<S>
42       , boost::is_signed<T>
43       , mpl::or_<boost::is_unsigned<T>
44         , mpl::and_<boost::is_signed<T>
45           , is_larger<T, S>>
46     >
47     >
48 {};
49
50 template <typename T, typename S>
51 struct is_non_pod_assignable
52   : mpl::and_<
53      mpl::not_<mpl::and_<boost::is_pod<T>
54                       , boost::is_pod<S>>
55   >
56      , std::is_assignable<T, S>
57   >
58 {};
59
60 template <typename T, typename U>
61 struct is_array_assignable
Chapter 7

63 : boost::false_type
64 {
65 }
66 template <typename T, size_t M, typename S, size_t N>
67 struct is_array_assignable<T (&)[M], S (&)[N]>
68 : mpl::and_<
69     mpl::or_<
70         boost::is_same<T, S>
71         , is_floating_assignable<T, S>
72         , is_integer_assignable<T, S>
73         , is_non_pod_assignable<T, S>
74     >
75     , mpl::greater_equal<mpl::integral_c<size_t, M>
76         , mpl::integral_c<size_t, N>>
77 >
78 {
79 }
80 
81 template <typename T, size_t M, typename S, size_t N>
82 typename boost::enable_if<is_array_assignable<T(&)[M],
83 S(&)[N>>::type
84 assignArray(T (&target)[M], S (&source)[N])
85 { /* actual copying implementation */ }

The primary template of the is_array_assignable metafunction always returns false (lines 61-64). The partial specialization of is_array_assignable (line 66-78) is the heart of the implementation. It uses the mpl::or_ metafunction to check whether any one of the following conditions is met:

- The source and target types are the same (line 70)
- The target type is a floating point, the source type is numeric, and an assignment is possible without narrowing (line 71)
- The target type is integral (signed or unsigned), the source type is integral, and an assignment is possible without narrowing (line 72)
- At least one of the source and target types is a non-POD type and a conversion from the source to the target type is possible (line 73)

The mpl::or_ metafunction is analogous to the logic or operator of C++, and its static member value is set to true if any one of the passed conditions is true. Along with this composite condition being true, the following condition must also hold:

The number of elements in the target array should be at least as much as the elements in the source array.
We use the `mpl::greater_equal` metafunction to compare these two values \( M \) and \( N \). Since the metafunction needs to take type parameters, we generate type parameters corresponding to \( M \) and \( N \) using `boost::mpl::integral_c` wrapper (lines 75-76). We compute the logical-OR of conditions 1-4 and its logical-AND with condition 5 using the `mpl::and_` metafunction (line 61).

We use `boost::enable_if` that leverages SFINAE to disable `assignArray` when `is_array_assignable` returns false.

Let us now look at the implementation of the `is_integer_assignable`. It checks if the target and source types are both integral, (lines 38-39) and the source type is not bigger than the target type (line 40). In addition, we use `boost::mpl::if_` metafunction, which takes three metafunctions; if the first metafunction evaluates to `true`, the second metafunction is returned, otherwise the third metafunction is returned. Using `mpl::if_`, we express the constraints on the source and target types (lines 41-47). If the source type is a signed integer (line 41), then the target type must also be a signed integer (line 42). But if the source type be an unsigned integer, then the target type must either be an unsigned integer (line 43) or a signed integer larger than the source type (lines 44-45). The rest of the traits are similarly defined using Boost MPL library facilities.

Metaprogramming is not just a tool for choosing optimal implementations or catching violations at compile time. It actually helps create expressive libraries like `boost::tuple` or `boost::variant`, involving significant type manipulation. We introduced only a few basic abstractions from the Boost MPL library to help you ease into template metaprogramming. If you have worked through the examples in this chapter, you should have no problems exploring MPL further on your own.

**Domain Specific Embedded Languages**

In the last third of this chapter, we look at the applications of higher order and compile-time programming mainly in the area Domain Specific Embedded Languages.

**Lazy evaluation**

In C++, when we see the following code:

\[
    z = x + y();
\]
We know that the value of $z$ is immediately computed when the control reaches past the statement $z = x + y()$. In fact, the act of computing the sum involves evaluating the expressions $x$ and $y()$ themselves. Here, $y$ is presumably a function or a functor instance, so the call to $y()$ will in turn trigger more evaluations. Irrespective of whether $z$ is ever used for anything later, its value would still be computed. This is the model of **eager evaluation** that a lot of programming languages follow. The actual story is slightly more complex because compilers can reorder and optimize away computations but there is little control the programmer has on the process.

What if we could defer the evaluation of such expressions and any of their sub-expressions until we have to make use of the result? This is the **lazy evaluation** model seen in a lot of functional programming languages, like Haskell. If we could construct arbitrary language expressions that are lazily evaluated, then such expressions could be passed around just like functors and evaluated where necessary. Imagine a function called `integrate` that evaluates definite integrals of arbitrary functions, given boundary values:

```c++
double integrate(std::function<double(double)> func, double low, double high);
```

Imagine being able to evaluate the integral $\int_1^{10} \frac{1}{x} dx$ by calling the following code:

```c++
double result = integrate(x + 1/x, 1, 10);
```

The key would be to not evaluate the expression $x + 1/x$ eagerly but pass it to the `integrate` function as a lazy expression. Now C++ does not have any built-in mechanism to create lazy expressions like these using regular variables. But we can quite easily write a lambda to get our job done:

```c++
result = integrate([](double) { return x + 1/x; }, 1, 10);
```

This works albeit with some syntactic noise, but in many applications, lambda and `bind` just do not scale with complexity. In this section, we briefly study **expression templates** and more generally, **Domain Specific Embedded Languages (DSELs)**, which are the means of constructing lazily evaluated function objects within C++ that get your job done without sacrificing on expressive syntax.

### Expression templates

So, how do we express a function $f(x)=x+1/x$ in the language of the domain rather than through a syntactic compromise within the confines of C++? To create a generic solution, we must be able to support a variety of algebraic expressions. Let us start with the most basic function—a constant function, such as $f(x)=5$. Irrespective of the value of $x$, this function should always return 5.
The following functor can be used for this purpose:

**Listing 7.22a: An expression template mini-library – lazy literals**

```cpp
#include <iostream>

struct Constant {
  Constant(double val = 0.0) : val_(val) {}
  double operator()(double) const { return val_; }

  const double val_;  
};

Constant c5(5);
std::cout << c5(1.0) << '
';  // prints 5
```

The `operator()` returns the stored `val_` and ignores its argument, which is unnamed. Now let us see how we can represent a function like \( f(x) = x \), using a similar functor:

**Listing 7.22b: An expression template mini-library – lazy variables**

```cpp
struct Variable {
  double operator()(double x) { return x; }
};

Variable x;
std::cout << x(8) << '
';  // prints 8
std::cout << x(10) << '
';  // prints 10
```

We now have a functor that yields whatever value is passed to it; exactly what \( f(x) = x \) does. But how do we express an expression like \( x + 1/x \)? The general form of a functor that represents an arbitrary function of a single variable should be as follows:

```cpp
struct Expr {
  ...
  double operator()(double x) {
    return (value computed using x);
  }
};
```

Both `Constant` and `Variable` conform to this form. But consider a more complex expression like \( f(x) = x + 1/x \). We can break it down to two sub-expressions \( x \) and \( 1/x \) acted upon by the binary operation \(+\). The expression \( 1/x \) can be further broken down to two sub-expressions \( 1 \) and \( x \) acted upon by the binary operation \(/\).
This can be represented by an **Abstract Syntax Tree (AST)**, as shown here:

The non-leaf nodes in the tree represent operations. Binary operation nodes have two children: the left operand is the left child and the right operand is the right child. The AST has an operation (+) at the root and two sub-expressions as two children. The left sub-expression is \( x \), while the right sub-expression is \( 1/x \). This \( 1/x \) is further deconstructed in a sub-tree with operation (/) at the root, 1 as the left child, and \( x \) as the right child. Notice that values like 1 and \( x \) only appear at the leaf level and correspond to the **Constant** and **Variable** classes we defined. All non-leaf nodes represent operators.

We can model a complex expression as one that is composed of two sub-expressions with an operator:

**Listing 7.22c: An expression template mini-library – complex expressions**

```cpp
1 template <typename E1, typename E2, typename OpType>
2 struct ComplexExpression {
3   ComplexExpression(E1 left, E2 right) : left_(left),
4       right_(right)
5   {}
6
7   double operator()(double x) {
8     return OpType()(left_(x), right_(x));
9   }
10
11   E1 left_; E2 right_;  // E1 and E2 are placeholders for complex expressions
12);
```

When the `ComplexExpression` functor is invoked, that is, when it evaluates its left and right sub-expressions and then applies the operator on them (line 7), this in turn triggers the evaluation of the left and right sub-expressions. If they are `ComplexExpression`s themselves, then they trigger further evaluations that traverse down the tree, depth-first. This is definitive **lazy evaluation**.
Now, in order to easily generate complex expression functors, we need to overload the arithmetic operators to combine sub-expressions of type `Constant, Variable, ComplexExpression<>`, or primitive arithmetic types. To do this better, we create an abstraction for all kinds of expressions called `Expr`. We also modify our definition of `ComplexExpression` to use `Expr`.

Listing 7.22d: An expression template mini-library – generic expressions

```cpp
1 template <typename E, typename Enable = void>
2 struct Expr {
3   Expr(E e) : expr_(e) {}  
4     
5   double operator()(double x) { return expr_(x); }  
6   
7 private:
8     E expr_;  
9   
10 };
11 template <typename E1, typename E2, typename Op>
12 struct ComplexExpression  
13 {
14   ComplexExpression(Expr<E1> left, Expr<E2> right) :
15     left_(left), right_(right) {}  
16   
17   double operator()(double d) {  
18     return Op()(left_(d), right_(d));  
19   }  
20   
21 private:
22     Expr<E1> left_;  
23     Expr<E2> right_;  
24   
```

We will pass around all kinds of expressions wrapped in `Expr`, for example, `Expr<Constant>, Expr<ComplexExpression>`, and so on. If you are unsure why we need the second template parameter `Enable`, then hang on for the answer in a bit. Before that, we will define the arithmetic operators between any two `Expr`s, starting with `operator+`:

Listing 7.22e: An expression template mini-library – overloaded operators

```cpp
1 #include <functional>
2 
```
Any binary operation will produce a ComplexExpression. Since we will pass everything abstracted as Expr, we return Expr<ComplexExpression<...>> from the arithmetic operators. It is easy to write an operator-, operator*, or operator/ on the same lines. We can replace std::plus with std::minus, std::multiples, or std::divides in the preceding implementation.

There is only one more detail to take care of. With the preceding code, we can write expressions of the following form:

Variable x;
Constant c1(1);
integrate(x + c1/x, 1, 10);

But we cannot write $x + 1/x$ using numeric literals. To do this, we must automatically convert numeric literals to Constant. For this, we will create a partial specialization of Expr and use boost::enable_if to enable it for numeric types. This is where the Enable argument of the Expr template comes in handy. It defaults to void for the primary template, but it helps us write the partial specialization for wrapping arithmetic-type literals.

**Listing 7.22f: An expression template mini-library – a small trick**

```cpp
#include <boost/utility/enable_if.hpp>
#include <boost/type_traits/is_arithmetic.hpp>

template <typename E>
struct Expr<E, typename boost::enable_if<boost::is_arithmetic<E>::type>::type

struct Expr<E, typename boost::enable_if<boost::is_arithmetic<E>::type>::type
```

```cpp
#include <boost/utility/enable_if.hpp>
#include <boost/type_traits/is_arithmetic.hpp>

3 template <typename E1, typename E2>
4 Expr<ComplexExpression<E1, E2, std::plus<double>>>
5       operator+ (E1 left, E2 right)
6 {
7   typedef ComplexExpression <E1, E2,
8   std::plus<double>> ExprType;
9   return ExprType(Expr<E1>(left), Expr<E2>(right));
10 }
```

```cpp
Any binary operation will produce a ComplexExpression. Since we will pass everything abstracted as Expr, we return Expr<ComplexExpression<...>> from the arithmetic operators. It is easy to write an operator-, operator*, or operator/ on the same lines. We can replace std::plus with std::minus, std::multiples, or std::divides in the preceding implementation.

There is only one more detail to take care of. With the preceding code, we can write expressions of the following form:

Variable x;
Constant c1(1);
integrate(x + c1/x, 1, 10);

But we cannot write $x + 1/x$ using numeric literals. To do this, we must automatically convert numeric literals to Constant. For this, we will create a partial specialization of Expr and use boost::enable_if to enable it for numeric types. This is where the Enable argument of the Expr template comes in handy. It defaults to void for the primary template, but it helps us write the partial specialization for wrapping arithmetic-type literals.

**Listing 7.22f: An expression template mini-library – a small trick**

```cpp
1 #include <boost/utility/enable_if.hpp>
2 #include <boost/type_traits/is_arithmetic.hpp>
3
4 template <typename E>
5 struct Expr<E, typename boost::enable_if
6   boost::is_arithmetic<E>::type>
7 {
8   Expr(E& e) : expr_(Constant(e)) {}
9
10   double operator()(double x) { return expr_(x); }
11
12   Constant expr_;
13 };
```
This partial specialization is invoked only when \( E \) is an arithmetic type (\texttt{int}, \texttt{double}, \texttt{long}, etc.). This stores the arithmetic value as a \texttt{Constant}. With this change, we can use numeric literals in our expressions, and as long as there is a single \texttt{Variable} in the expression, the literals would get wrapped in a \texttt{Constant} via the partial specialization in listing 7.22f. We can now generate a functor using just natural algebraic expressions:

**Listing 7.22g: An expression template mini-library – using the expressions**

```cpp
Variable x;
std::cout << (x + 1/x)(10) << '\n';
std::cout << ((x*x - x + 4)/(2*x))(10) << '\n';
```

We can add many more refinements to this very basic expression template library of not even a hundred lines of code. But it already allows us to generate arbitrary algebraic functions of a single variable using very simple syntax. This is an example of a Domain Specific Language. Also, specifically, because we use valid C++ syntax to do all this instead of defining a new syntax, it is specifically called Domain Specific Embedded Language (DSEL) or sometimes Embedded Domain Specific Language (EDSL). We will now look at Boost Phoenix, an elaborate library of lazy expressions.

**Boost Phoenix**

Boost Phoenix is a library for enabling functional programming constructs in C++. It defines an elaborate and very readable DSEL with scores of functors and operators, which can be used to generate fairly involved lambdas. It provides a comprehensive library for constructing lazy expressions and an excellent example of what expression templates can achieve. This section features a very short introduction to using Phoenix expressions as lambdas, and we will see some examples of using Phoenix with Boost Spirit Parser Framework. It is too extensive a library to cover in a single chapter, let alone a subsection of it, but this introduction should still provide enough tail wind to master Phoenix, with the benefit of the excellent online documentation.

Phoenix expressions are composed of \texttt{actors}, which are abstractions for lazy functions. Actors are used to generate unnamed functions or lambdas. They support partial function application by binding some arguments to values and keeping others unspecified. They can be composed to generate more complex functors. In that sense, Phoenix is a lambda language library.
Actors are categorized based on functionality and exposed through a set of header files. The most basic actor is `val` which represents a lazy immutable value (not unlike the `Constant` functor in our expression template example). The `ref` actor is used to create a lazy mutable variable reference, and the `cref` actor generates a lazy immutable reference. There is a whole set of actors that define lazy operators, including arithmetic (`+`, `-`), comparison (`,` `==`, `>`) logical (`&&`, `||`), bitwise operators (`|`, `^`, `&`), and other kinds of operators. Using just these, we can construct algebraic expressions, as we do in the following example:

**Listing 7.23: Lazy algebraic expressions with Phoenix**

```cpp
#include <boost/phoenix/core.hpp>
#include <boost/phoenix/operator.hpp>
#include <iostream>

int main() {
    namespace phx = boost::phoenix;
    double eX;
    auto x = phx::ref(eX);

    eX = 10.0;
    std::cout << (x + 1/x)() << '
';              // prints 10.1
    std::cout << ((x*x -x + 4) / (2*x))() << '
';  // prints 4.7
}
```

Using `boost::phoenix::ref`, we generate an actor for lazily evaluating the variable `eX` (`e` for eager) and cache it in a variable `x`. The expressions `x + 1/x` and `x*x - x + 4` generate anonymous functors just like the expression templates from listing 7.22, except that `x` is already bound to the variable `eX`. The actor `x` is said to **infect** the numeric literals in the expressions by its presence; the literals get wrapped in `boost::phoenix::val`. The operators `+`, `-`, `*`, and `/` used in the expression are lazy operators from Phoenix (just like the operators we defined for our expression template in listing 7.22e) and generate anonymous functors.

Writing simple lambdas can sometimes be extremely succinct using Phoenix. Look at how we can print each element in a vector using `std::for_each` and Phoenix's lazy operator `<<`:

**Listing 7.24: Simpler lambdas with Phoenix**

```cpp
#include <boost/phoenix/core.hpp>
#include <boost/phoenix/operator.hpp>
#include <vector>

int main() {
    namespace phx = boost::phoenix;
    double eX;
    auto x = phx::ref(eX);

    eX = 10.0;
    std::cout << (x + 1/x)() << '
';              // prints 10.1
    std::cout << ((x*x -x + 4) / (2*x))() << '
';  // prints 4.7
}
```
The expression `std::cout << arg1` is actually a lambda that generates a functor. The actor `arg1` (boost::phoenix::arg_names::arg1) represents the first argument to the functor and is lazily evaluated. The presence of `arg1` in the expression `std::cout << arg1` invokes the lazy operator `<<` and infects the entire expression to generate an unnamed function that prints its argument to the standard output. In general, you can use `arg1` through `argN` to refer to the lazy arguments of an N-ary functor generated with Phoenix. By default, up to ten argument actors (`arg1` through `arg10`) are supported. These are akin to `_1`, `_2`, etc. for boost::bind. You can also use boost::phoenix::placeholders::_1, _2, etc.

Phoenix actors are not limited to expressions involving operators. We can generate actors that lazily evaluate entire blocks of code with branching and looping constructs. Let us say we have a vector of the names of personnel in a band's lineup, and we want to print whether a person is a vocalist or instrumentalist:

**Listing 7.25: Lazy control structures with Phoenix**

```cpp
#include <boost/phoenix/core.hpp>
#include <boost/phoenix/statement/if.hpp>
#include <boost/phoenix/operator.hpp>
#include <algorithm>
#include <vector>
#include <iostream>

int main() {
    namespace phx = boost::phoenix;
    using namespace phx;
    using phx::arg_names::arg1;

    std::vector<std::string> names = make_pair("Daltrey", "Townshend",
                                              "Entwistle", "Moon");
    std::for_each(names.begin(), names.end(),
                  if_(arg1 == "Daltrey") {
                      std::cout << arg1 << 'n';
                  }
    }
```
We want to run through the vector of last names of the four legendary members of *The Who* and list them with their roles. For (Roger) Daltrey, the role would be of a vocalist and for the others, instrumentalist. We use `std::for_each` to iterate the list of names. We pass a unary functor to it generated using Phoenix's statement actors, specifically `boost::phoenix::if_`.

The syntax is intuitive enough to look at and understand what is going on. The actual statements in the `if_` and `else_` blocks are put in square brackets instead of braces (which cannot be overloaded) and are lazily evaluated. If there were multiple statements, they would need to be separated by commas. Notice how the `else_` is a member call invoked with a dot on the preceding expression (line 18). The presence of `arg1` is said to *infect* the statements, that is, it invokes the lazy operator `<<` and causes the literal character strings to be automatically wrapped in `boost::phoenix::val` (lines 16, 17, 19). Running this code prints the following:

```
Daltrey, vocalist
Townshend, instrumentalist
Entwistle, instrumentalist
Moon, instrumentalist
```

The power of Phoenix should be evident already. It defines an expressive sub-language using standard C++ operator overloading and functors that easily generates unnamed functions or lambdas as needed, and starts to mimic the host language itself. There is more to the Phoenix library. It is chock-full of actors for lazy evaluation of STL container member functions and STL algorithms. Let us look at an example to understand this better:

**Listing 7.26: Actors for STL algorithms and container member functions**

```cpp
#include <vector>
#include <string>
#include <iostream>
#include <boost/phoenix/core.hpp>
#include <boost/phoenix/stl/algorithm.hpp>
#include <boost/phoenix/stl/container.hpp>
#include <cassert>

int main() {
```

We have a vector `greets` of hello greetings in different languages (English, Spanish, Swahili, and German), and we want to search for a specific greet. We want to do so lazily using Phoenix. Phoenix provides actors for generating lazy versions of most STL algorithms. We use the lazy form of the `std::find` algorithm available via the header `boost/phoenix/stl/algorithm.hpp` (line 5), and call the `boost::phoenix::find` actor to generate a unary functor named `finder` (line 14). The `finder` functor takes as its only argument, the string to look for in `greets`. The call `boost::phoenix::find(greets, arg1)` takes two arguments and generates a unary functor. The first argument is a reference to the vector `greets`, which is automatically wrapped in a `cref` actor and stored for lazy evaluation later. The second argument to `find` is the Phoenix placeholder `arg1`.

When `finder` is called with the string to lookup as its sole argument, it evaluates the `arg1` actor to get this string argument. It also evaluates the `cref` actor it stored earlier to get a reference to `greets`. It then calls `std::find` on the `greets` vector, looking for the string passed, which returns an iterator. We look for the string `Hujambo` which is the present in the vector (line 15).

To check whether the iterator returned is valid, we need to compare it against `greets.end()`. Just to show that it can be done, we generate the lazy version of the `end` member function call using the `boost::phoenix::end` actor available from the header `boost/phoenix/stl/algorithm.hpp`. The call `boost::phoenix::end(greets)` generates a functor, and we call it in-place by suffixing parentheses. We compare the result with the iterator returned by `finder` (line 17). We print the greeting pointed by the iterator returned by `find` and the element after that (lines 18-20):

```
Hujambo
Hallo
```
Actors from Phoenix are polymorphic. You can apply `boost::phoenix::find` on any kind of container that supports searching via `std::find`, and you can look up an object of any type that the underlying container can store.

In the final example on Phoenix, we look at how we can define our own actors, which can fit in with the rest of Phoenix. We have a vector of names from which we print the first name in each entry, using `std::for_each` and functors generated using Phoenix. We extract first names from a name string by looking up the first space character in the string and extracting the prefix up to that point. We can use the `find` actor to locate the space but to extract the prefix, we need a lazy way to call the `substr` member of `std::string`. There is no `substr` actor currently available in Phoenix, so we need to roll out our own:

**Listing 7.27: User defined actors and STL actors**

```cpp
#include <vector>
#include <string>
#include <iostream>
#include <algorithm>
#include <boost/phoenix/core.hpp>
#include <boost/phoenix/function.hpp>
#include <boost/phoenix/operator.hpp>
#include <boost/phoenix/stl/container.hpp>
#include <boost/phoenix/stl/algorithm.hpp>

struct substr_impl {
    template<typename C, typename F1, typename F2>
    struct result {
        typedef C type;
    };  

    template<typename C, typename F1, typename F2>
    C operator()(const C& c, const F1& offset, const F2& length) const
    {  return c.substr(offset, length); }
};

int main() {
    namespace phx = boost::phoenix;
    using phx::arg_names::arg1;

    std::vector<std::string> names = {"Pete Townshend", "Roger Daltrey", "Keith Moon", "John Entwistle");
    phx::function<substr_impl> const substr = substr_impl();
```
We write the \texttt{substr\_impl} functor, which has a member template \texttt{operator()} (line 17) and a metafunction called \texttt{result} (line 12). The \texttt{operator()} is a template used to make \texttt{substr\_impl} polymorphic. Any type \texttt{C} with a member function called \texttt{substr}, which takes two parameters of type \texttt{F1} and \texttt{F2} (which may or may not be of different types) can be covered by this single implementation (lines 17-20). The embedded type in the \texttt{result} metafunction is the return type of the wrapped function (\texttt{substr}). The actual \texttt{substr} actor is an instance of type \texttt{boost::phoenix::function<substr\_impl>} (line 29). We use the \texttt{substr} actor, we just defined, to generate a unary functor, which we pass to the \texttt{std::for\_each} algorithm (lines 32-33). Since we want to extract the first name from each string in the \texttt{names} vector, the first argument is \texttt{arg1} (the name passed to the function), the second offset argument is 0, while the third length argument is the offset of the first space character in the string. The third argument is calculated lazily as the expression \texttt{boost::phoenix::find(arg1, ' ') \texttt{} - boost::phoenix::begin(arg1)}. The \texttt{find(arg1, ' ')} is an actor that looks up the first space in the string passed to it using the generic find actor from Phoenix that we also used in listing 7.26. The \texttt{begin(arg1)} is an actor that returns the begin iterator of its argument (in this case the string). The difference between them returns the length of the first name.

**Boost Spirit Parser Framework**

Boost Spirit is a very popular DSEL used for generating lexers and parsers, which uses Boost Phoenix. Writing custom lexers and parsers used to be heavily reliant on specialized tools like lex/\texttt{flex}, yacc/bison, and ANTLR that generated C or C++ code from a language neutral specification in the Extended Backus-Naur Form (EBNF). Spirit eliminates the need for creating such a specification outside the language, and for tools to translate from such specifications. It defines a declarative DSEL with intuitive syntax in C++ and uses only the C++ compiler to generate parsers. Spirit makes heavy use of template metaprogramming, resulting in slower compile times but generates parsers that are efficient at runtime.

Spirit is a rich framework that includes Spirit Lex – a lexer, Spirit Qi – a parser, and Spirit Karma – a generator. You can use these separately, or use them all in collaboration to build powerful data translation engines.
In this book we only look at Spirit Qi. It is used primarily to parse text data according to some specified grammar that the data is supposed to obey, with the following objectives:

- Verifying that the input conforms to the grammar
- Decomposing a conforming input into meaningful semantic components

For example, we can parse some input text to verify whether it is a valid timestamp, and if it is, extract the components of the timestamp, such as year, month, day, hours, minutes, and so on. For this, we need to define a grammar for the timestamp, and we need to define the actions to be taken, as we parse the data in terms of its semantic constituents. Let us see a concrete example.

**Using Spirit Qi**

Spirit provides predefined parsers, which can be combined using parser operators defined by Spirit, to define a parser for our needs. Once defined, we can store the parser or its components as rules that can be combined with other rules. Or we can directly pass it to a Qi parsing API, such as parse or phrase_parse, along with the input to parse.

**Predefined parsers**

Qi provides a number of predefined parsers that can be used to parse basic pieces of data. The parsers are available or aliased under the namespace boost::spirit::qi. Here is a listing of these parsers with their purpose:

<table>
<thead>
<tr>
<th>Input class</th>
<th>Parsers</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integers</td>
<td>int_, uint_, short_, ushort_, long_, ulong_, long_long, ulong_long</td>
<td>Parse signed and unsigned integers</td>
</tr>
<tr>
<td>Real numbers</td>
<td>float_, double_, long_double</td>
<td>Parse real numbers with decimal points</td>
</tr>
<tr>
<td>Boolean</td>
<td>bool_, true_, false_</td>
<td>Parse either or both the strings, true and false</td>
</tr>
<tr>
<td>Characters</td>
<td>char_, alpha, lower, upper, digit, xdigit, alnum, space, blank, punct, cntrl, graph, print</td>
<td>Parse characters of different classes, like letters, digits, hexadecimal digits, punctuation, etc.</td>
</tr>
<tr>
<td>Strings</td>
<td>String</td>
<td>Parse specific strings</td>
</tr>
</tbody>
</table>
The parsers listed in the preceding table are predefined objects rather than types. There are generic parser templates corresponding to each of these parsers. For example, the template `boost::spirit::qi::int_parser` can be used to define custom parsers for signed integers. There are many other templates, including `boost::spirit::qi::uint_parser`, `boost::spirit::qi::bool_parser`, and so on.

**The parsing API**

Qi provides two function templates, `parse` and `phrase_parse`, that are used to parse text input. Each takes a pair of iterators that define the input range and a parser expression. In addition, `phrase_parse` takes a second parser expression that is used to match and skip whitespace. The following short example shows you the essence of using Spirit:

**Listing 7.28: A simple Spirit example**

```cpp
#include <boost/spirit/include/qi.hpp>
#include <cassert>
namespace qi = boost::spirit::qi;

int main()
{
  std::string str = "Hello, world!";

  auto iter = str.begin();
  bool success = qi::parse(iter, str.end(), qi::alpha);

  assert(!success);
  assert(iter - str.begin() == 1);
}
```

We include the header file `boost/spirit/include/qi.hpp` in order to access Spirit Qi functions, types, and objects. Our input is the string "Hello, world!", and using the predefined parser `alpha`, we want to enforce that the first character is a letter from the Latin alphabet, as opposed to a digit or a punctuation symbol. For this, we use the `parse` function, passing it a pair of iterators defining the input and the `alpha` parser (line 10). The `parse` function returns `true` if the parser successfully parses the input and `false` otherwise. The iterator to the start of the range is incremented to point to the first unparsed character in the input. Since the first character of Hello, world! is H, the `alpha` parser parses it successfully, incrementing the `iter` by 1 (line 13) and `parse` returns `true` (line 12). Note that the first iterator is passed as a non-const reference to `parse` and is incremented by `parse`; the reason we pass a copy of `str.begin()`.
## Parser operators and expressions

Spirit defines a number of overloaded operators called **parser operators** which can be used to compose a complex parser expression out of simpler parsers, including the predefined ones. The following table summarizes some of these operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;&gt;</code> (Sequence operator)</td>
<td>Binary, infix</td>
<td>Two parsers serially parse two tokens</td>
<td><code>string(&quot;Hello&quot;) &gt;&gt; string(&quot;world&quot;)</code>  Matches Helloworld.</td>
</tr>
<tr>
<td>`</td>
<td>` (Disjunction operator)</td>
<td>Binary, infix</td>
<td>Any one of the two parsers is able to parse the token, but not both</td>
</tr>
<tr>
<td><code>*</code> (Kleene operator)</td>
<td>Unary, prefix</td>
<td>Parses the empty string or one or more matching tokens</td>
<td><code>*string(&quot;Hello&quot;)</code>  Matches the empty string, Hello, HelloHello, and so on.</td>
</tr>
<tr>
<td><code>+</code> (Plus operator)</td>
<td>Unary, prefix</td>
<td>Parses one or more matching tokens</td>
<td><code>+string(&quot;Hello&quot;)</code>  Matches Hello, HelloHello, and so on, but not the empty string.</td>
</tr>
<tr>
<td><code>~</code> (Negation operator)</td>
<td>Unary, prefix</td>
<td>Parses a token that does not match the parser</td>
<td><code>-xdigit</code>  Will parse any character that is not a hexadecimal digit.</td>
</tr>
<tr>
<td><code>-</code> (Optional operator)</td>
<td>Unary, prefix</td>
<td>Parses the empty string or a single matching token</td>
<td><code>-string(&quot;Hello&quot;)</code>  Matches Hello or the empty string.</td>
</tr>
<tr>
<td><code>-</code> (Difference operator)</td>
<td>Binary, infix</td>
<td><code>P1 - P2</code> parses any token that P1 can parse and P2 cannot</td>
<td><code>uint_ - ushort_</code>  Matches any unsigned int that is not also an unsigned short. Matches 65540 but not 65530 on a system with 2-byte short.</td>
</tr>
<tr>
<td><code>%</code> (List operator)</td>
<td>Binary, infix</td>
<td><code>P1 % D</code> splits the input into tokens that match P1 at delimiters that match D</td>
<td>`alnum % +(space</td>
</tr>
<tr>
<td>`</td>
<td></td>
<td>` (Sequential OR operator)</td>
<td>Binary, infix</td>
</tr>
</tbody>
</table>
Higher Order and Compile-time Programming

Note that there is a unary operator -, which is the optional operator, and binary operator -, which is the difference operator.

The boost::spirit::qi::parse function template does not skip any characters whitespaces while parsing. Sometimes, it is convenient to ignore intervening spaces between tokens while parsing, and the boost::spirit::qi::phrase_parse does this. For example, the parser string("Hello") >> string("world") would parse Helloworld when we use boost::spirit::qi::parse, but not Hello, world!. But if we used phrase_parse and ignored spaces and punctuation, then it would parse Hello, world! too.

Listing 7.29: Using phrase_parse

```cpp
#include <boost/spirit/include/qi.hpp>
#include <cassert>
namespace qi = boost::spirit::qi;

int main()
{
    std::string str = "Hello, world!";
    auto iter = str.begin();
    bool success = qi::parse(iter, str.end(),
              qi::string("Hello") >> qi::string("world"));
    assert(!success);
    iter = str.begin();
    success = qi::phrase_parse(iter, str.end(),
              qi::string("Hello") >> qi::string("world"),
              +(qi::space|qi::punct));
    assert(success);
    assert(iter - str.begin() == str.size());
}
```

Note that we pass +(space|punct) as the fourth argument to phrase_parse, which tells it which characters to ignore; spaces and punctuation.
 Parsing directives

Parsing directives are modifiers that can be used to alter the behavior of parsers in some way. For example, we can perform case-insensitive parses using the `no_case` directive, as shown in the following snippet:

```cpp
1 std::string str = "Hello, WORLD!";
2 iterator = str.begin();
3 success = qi::phrase_parse(iterator, str.end(),
4     qi::string("Hello") >>
5     qi::no_case(qi::string("world")),
6     +(qi::space|qi::punct));
7 assert(success);
```

The `skip` directive can be used to skip whitespace over a section of the input:

```cpp
1 std::string str = "Hello world";
2 auto iter = str.begin();
3 bool success = qi::parse(iter, str.end(),
4     qi::skip(qi::space)[qi::string("Hello") >>
5     qi::string("world")]);
6 assert(success);
```

The directive `qi::skip(qi::space)[parser]` ignores spaces even though we called `parse` and not `phrase_parse`. It can be selectively applied to parser sub-expressions.

Semantic actions

More often than not, while using Spirit, we are not just looking to verify that a piece of text conforms to a certain grammar; we want to extract the tokens and perhaps use them in some kind of calculation or store them away. We can associate some action to a parser instance to be run when it successfully parses text, and this action can perform the necessary computation using the result of the parse. Such actions are defined using a function object enclosed in square brackets, trailing the parser it is associated with.

Listing 7.30: Defining actions associated with parsers

```cpp
1 #include <boost/spirit/include/qi.hpp>
2 #include <iostream>
3 namespace qi = boost::spirit::qi;
4
5 void print(unsigned int n) {
```
In the preceding example, we parse a list of unsigned integers separated by spaces (line 10) using the uint_ parser (line 14). We define a function print (line 5) to print unsigned integers and associate it as an action with the uint_ parser (line 14). For each unsigned integer parsed, the preceding code prints it on a new line by invoking the specified action. Actions can also be specified using functors, including those generated by Boost Bind and Boost Phoenix.

Each parser, from the primitive to the most complex, has an associated attribute, which is set to the result of a successful parse, that is, the text it matches when it is applied to some input converted to the appropriate type. For a simple parser like uint_, this attribute would be of type unsigned int. For complex parsers, this could be an ordered tuple of attributes of its constituent parsers. When an action associated with a parser is invoked, it is passed the value of the parser's attribute.

The expression +qi::uint_[print] associates the print function with the uint_ parser. If instead we wanted to associate an action with the composite parser +qi::uint_, then we would need to use a function with a different signature—one with a parameter of type std::vector<unsigned int> that would contain all the parsed numbers:

```cpp
#include <vector>

void printv(std::vector<unsigned int> vn)
{
  for (const int& n: vn) {
    std::cout << n << 'n';
  }
```
int main() {
    std::string str = "10 20 30 40 50 60";
    auto iter = str.begin();
    bool success = qi::phrase_parse(iter, str.end(),
                                      (*qi::uint_)[printv],
                                      qi::space);
}

We can use Boost Bind expressions and Phoenix actors too for generating the action. Thus, we could have written +qi::uint_[boost::bind(print, ::_1)] to call print on each parsed number. The placeholders ::_1 through ::_9 are defined by the Boost Bind library in the global namespace. Spirit provides Phoenix actors that can be used for a variety of actions. The following snippet shows a way to add parsed numbers into a vector:

#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>
#include <boost/spirit/include/phoenix_stl.hpp>

int main() {
    using boost::phoenix::push_back;
    std::string str = "10 20 30 40 50 60";
    std::vector<unsigned int> vec;
    auto iter = str.begin();
    bool status = qi::phrase_parse(iter, str.end(),
                                    (+qi::uint_[push_back(boost::phoenix::ref(vec),
                                      qi::_1)],
                                    qi::space);
}

The action expression push_back(boost::phoenix::ref(vec), qi::_1) uses the boost::phoenix::push_back actor to append each parsed number (represented by the placeholder qi::_1) to the vector vec.
There are overloads of the `parse` and `phrase_parse` function templates that take an attribute argument in which you can directly store the data parsed by the parser. Thus, we can pass a `vector` of `unsigned int` as the attribute argument, while parsing the list of unsigned integers:

```cpp
std::vector<unsigned int> result;
bool success = qi::phrase_parse(iter, str.end(),
                                 +qi::uint_, result,
                                 qi::space);
for (int n: result) {
    std::cout << n << '
';
}
```

**Rules**

So far, we have generated parsers using inline expressions. When dealing with more complex parsers, it is useful to cache the components and reuse them. For this purpose, we use the `boost::spirit::qi::rule` template. The rule template takes up to four arguments of which the first, that is, the iterator type for the input, is mandatory. Thus, we can cache a parser that parses spaces in `std::string` objects, as shown here:

```cpp
qi::rule<std::string::iterator> space_rule = qi::space;
```

Notice that `space_rule`, defined as above, is a parser that follows the same grammar as `qi::space`.

More often than not, we are interested in consuming the value parsed by the parser. To define a rule containing such a parser, we need to specify the signature of a method that would be used to obtain the parsed value. For example, the `boost::spirit::qi::double_parser`'s attribute is of type `double`. So, we consider a function taking no arguments and returning a `double` as the appropriate signature `double()` to use. This signature is passed as the second template argument to the rule:

```cpp
qi::rule<std::string::iterator, double()> double_rule =
    qi::double_;
```

If the rule is meant to skip spaces, we specify the type of parser that is used to identify the characters to skip as the third template argument to `rule`. Thus, to define a parser for a list of doubles separated by spaces, we can use the following rule with `qi::space_type`, specifying the type of the space parser:

```cpp
qi::rule<std::string::iterator, std::vector<double>(),
         qi::space_type> doubles_p = +qi::double_;
```
When a rule is defined in terms of a combination of parsers, the value parsed by the rule is synthesized from the values parsed by the individual component parsers. This is called the synthesized attribute of the rule. The signature argument to the rule template should be compatible with the type of the synthesized attribute. For example, the parser `+qi::double_` returns a sequence of doubles, and therefore, the type of the synthesized attribute is `std::vector<std::double>`:

```cpp
qi::rule<std::string::iterator, std::vector<double>(),
         qi::space_type> doubles_p;

doubles_p %= +qi::double_;```

Notice that we assign the parser to the rule on a separate line, using operator `%=`. If we did not use the `%=` operator and used the plain assignment operator instead, then the result of a successful parse using `+qi::double_` would not be propagated to the synthesized attribute of `doubles_p`. Thanks to the `%=` operator, we can associate a semantic action with `doubles_p` to access its synthesized value as a `std::vector<double>`, as shown in the following example:

```cpp
std::string nums = "0.207879576 0.577215 2.7182818 3.14159259";
std::vector<double> result;
qi::phrase_parse(iter1, iter2,
                 doubles_p[boost::phoenix::ref(result) == qi::_1],
                 qi::space);
```

### Parsing timestamps

Consider timestamps of the form YYYY-mm-DD HH:MM:SS.ff, in which the date part is mandatory and the time part is optional. Moreover, the seconds and fractional seconds part of the time are also optional. We need to define a suitable parser expression.

The first thing we require is a way to define parsers for fixed-length unsigned integers. The `boost::spirit::qi::int_parser` template comes in handy for this purpose. Using template parameters of `int_parser`, we specify the base integral type to use, the radix or base of the number system, and the minimum and maximum number of digits to allow. Thus, for 4-digit years, we can use a parser type `int_parser<unsigned short, 10, 4, 4>`, both the minimum and maximum width being 4, as we need fixed-length integers. The following are the rules constructed using `int_parser`:

```cpp
#include <boost/spirit/include/qi.hpp>

namespace qi = boost::spirit::qi;

qi::int_parser<unsigned short, 10, 4, 4> year_p;
```
qi::int_parser<unsigned short, 10, 2, 2> month_p, day_p, hour_p,
    min_p, sec_p;

qi::rule<std::string::iterator> date_p =
    year_p >> qi::char_('\-') >> month_p >> qi::char_('\-') >> day_p;

qi::rule<std::string::iterator> seconds_p =
    sec_p >> -(qi::char_('\.') >> qi::ushort_);

qi::rule<std::string::iterator> time_p =
    hour_p >> qi::char_('\:') >> min_p
    >> -(qi::char_('\:') >> seconds_p);

qi::rule<std::string::iterator> timestamp_p = date_p >> -
    (qi::space >> time_p);

Of course, we need to define actions to capture the components of the timestamp.
For simplicity, we will associate actions with the component parsers. We will define
a type to represent timestamps and associate actions with parsers to set attributes
of an instance of this type.

**Listing 7.31: Simple date and time parser**

```cpp
1 #include <boost/spirit/include/qi.hpp>
2 #include <boost/bind.hpp>
3 #include <cassert>
4 namespace qi = boost::spirit::qi;
5
6 struct timestamp_t
7 {
8   void setYear(short val) { year = val; }
9   unsigned short getYear() { return year; }
10   // Other getters / setters
11 private:
12   unsigned short year, month, day,
13       hours, minutes, seconds, fractions;
14};
15
16 timestamp_t parseTimeStmp(std::string input)
17 {
```
timestamp_t ts;
qi::int_parser< unsigned short, 10, 4, 4> year_p;
qi::int_parser< unsigned short, 10, 2, 2> month_p, day_p,
        hour_p, min_p, sec_p;
qi::rule< std::string:: iterator> date_p =
    year_p [boost::bind(&timestamp_t::setYear, &ts, ::_1)]
    >> qi::char_('-')
    >> month_p [boost::bind(&timestamp_t::setMonth, &ts, ::_1)]
    >> qi::char_('-')
    >> day_p [boost::bind(&timestamp_t::setDay, &ts, ::_1)];
qi::rule< std::string:: iterator> seconds_p =
    sec_p [boost::bind(&timestamp_t::setSeconds, &ts, ::_1)]
    >> -(qi::char_('.'))
    >> qi::ushort_<
        [boost::bind(&timestamp_t::setFractions, &ts, ::_1)];
qi::rule< std::string:: iterator> time_p =
    hour_p [boost::bind(&timestamp_t::setHours, &ts, ::_1)]
    >> qi::char_(':')
    >> min_p [boost::bind(&timestamp_t::setMinutes, &ts, ::_1)]
    >> -(qi::char_(':')) >> seconds_p;
qi::rule< std::string:: iterator> timestamp_p = date_p >> -
        (qi::space >> time_p);
auto iterator = input.begin();
bool success = qi::phrase_parse(iterator, input.end(),
    timestamp_p, qi::space);
assert(success);
return ts;
}

The timestamp_t type (line 6) represents a timestamp, with getters and setters for each of its fields. We have omitted most of the getters and setters for conciseness. We define actions associated with parsers for individual fields of the timestamp, setting appropriate attributes of a timestamp_t instance using boost::bind (lines 25, 27, 29, 32, 35, 38, 40).
Self-test questions

For multiple choice questions, choose all the options that apply:

1. Which of the following overloads/specializations does the call `foo(1.0, std::string("Hello"))` resolve to?
   a. template<typename T, typename U> foo(T, U);
   b. foo(double, std::string&);
   c. template <> foo<double, std::string>
   d. There is ambiguity

2. What is the interface that a metafunction must satisfy?
   a. It must have a static value field
   b. It must have an embedded type called `type`
   c. It must have a static `type` field
   d. It must have an embedded type called `result`

3. What does the following statement do: `boost::mpl::or_<boost::isFloatingPoint<T>, boost::is_signed<T>>`?
   a. Checks whether type T is signed and a floating point type
   b. Generates a metafunction that checks (a)
   c. Checks whether type T is signed or a floating point type
   d. Generates a metafunction that checks (b)

4. We have a template declared as: `template <typename T, typename Enable = void> class Bar` and does not use the `Enable` parameter in any way. How do you declare a partial specialization of Bar that would be instantiated only when T is a non-POD type?
   a. template <T> class Bar<T, boost::is_non_pod<T>>
   b. template <T> class Bar<T, boost::enable_if<is_non_pod<T>>::type>
   c. template <T> class Bar<T, boost::mpl::not<boost::is_pod<T>>>
   d. template <T> class Bar<T, boost::disable_if<is_pod<T>>::type>
5. Which of the following is true of C++ lambda expressions and Boost Phoenix actors?

   a. Lambda expressions are unnamed, Phoenix actors are not
   b. Phoenix actors are polymorphic, while polymorphic lambda expressions are only available from C++14
   c. Phoenix actors can be partially applied, while lambda expressions cannot
   d. Lambda expressions can be used as closures but Phoenix actors cannot

**Summary**

This chapter was an interlude in our exploration of the Boost libraries. There were two key underlying themes: more expressive code and faster code. We saw how higher order programming helps us achieve more expressive syntaxes using functors and operator overloading. We saw how template metaprogramming techniques allow us to write code that executes at compile time and chooses the most optimal implementations for the task at hand.

We covered a diverse amount of material in a single chapter and introduced a paradigm of programming that may be new to some of you. We solved a few problems with different functional patterns and saw the power of C++ functors, templates, and operator overloading put together. Understanding the subject of this chapter will be of immediate help if you are reading the implementation of most Boost libraries or trying to write a fast general purpose library that is efficient, expressive, and extensible.

There is a lot that we did not cover in this chapter and do not cover in this book, including many, but the most basic details of Boost Spirit, a DSEL construction kit, Boost Proto; an expression template-based fast regular expression library, Boost Xpressive; and a more advanced tuple library, Boost Fusion. Hopefully, this chapter gives you enough of a head start to explore them further. Starting with the next chapter, where we cover Boost libraries for date and time calculations, we switch gears to focus on systems programming libraries in Boost.

**References**

- *C++ Common Knowledge*, Stephen C. Dewhurst, Addison Wesley Professional
- *Modern C++ Design*, Andrei Alexandrescu, Addison Wesley Professional
- *C++ Template Metaprogramming*, David Abrahams and Aleksey Gurtovoy, Addison Wesley Professional
- **Fusion**: [www.boost.org/libs/fusion](http://www.boost.org/libs/fusion)
Where to buy this book
You can buy Learning Boost C++ Libraries from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.