The Boost C++ Libraries

Boris Schäling

September 22, 2014

The Boost C++ Libraries
by Boris Schäling

Edition Second English
Published September 22, 2014
Copyright © 2008 - 2014 Boris Schäling

Contents

Preface

What you will learn

This book is an introduction to the Boost C++ Libraries. The Boost C++ Libraries complement the standard library. Because the Boost C++ Libraries are based on the standard, they are implemented using state-of-the-art C++. They are platform independent and are supported on many operating systems, including Windows and Linux, by a large developer community.
The Boost C++ Libraries enable you to boost your productivity as a C++ developer. For example, you can benefit from smart pointers that help you to write more reliable code or use one of the many libraries to develop platform-independent network applications. Since the Boost libraries partly anticipate developments in the standard, you can benefit earlier from tools without having to wait for them to become available in the standard library.

What you should know

Since the Boost libraries are based on, and extend, the standard, you should know the standard well. You should understand and be able to use containers, iterators, and algorithms, and ideally you should have heard of concepts such as RAII, function objects, and predicates. The better you know the standard, the more you will benefit from the Boost libraries.
In general, you don't need any knowledge of template meta programming to use the libraries introduced in this book. The main focus is on libraries that can be learned quickly and easily and that can be immediately of great benefit in your work as a C++ developer.
Many examples use features that were added to the standard with C++11. For example, the keyword `auto` is used to avoid specifying types explicitly. Constructors are called through uniform initialization: variables are initialized, if possible, with a pair of curly brackets instead of parentheses. Many examples use lambda functions to make code shorter and more compact. While you can understand many examples without detailed knowledge of C++11, this book is based on the current standard.

Typographical Conventions

The following text styles are used in this book:

`Monospace font` A monospace font is used for class names, function names, and keywords — basically for any C++ code. It is also used for code examples, command line options, and program output. For example: `int i =0;`

`Monospace bold font` A monospace bold font is used for variable names, objects, and user input. For example: The variable **`i`** is initialized with 0.

Bold Commands are marked in bold. For example: The Boost libraries are compiled with a program called **bjam**.

Italic An italic font is used when a new concept is introduced and mentioned for the first time. For example: *RAII* is the abbreviation for Resource Acquisition Is Initialization – a concept smart pointers are based on.

Examples

This book contains more than 430 examples. Every example is complete and can be compiled and executed. You can download all examples from `http://theboostcpplibraries.com/examples/` for a quick start.

All examples have been tested with the following compilers: Microsoft Visual Studio Professional 2013 Update 1 (64-bit Windows 7 Professional with Service Pack 1), GCC 4.8.3 (64-bit Cygwin 1.7.30), GCC 4.6.3 (32-bit Ubuntu 12.04.4), and Clang 3.3 (32-bit Ubuntu 12.04.4).

All of the examples in this book are based on the C++11 standard. During testing, all of the compilers were configured to enable support for C++11. Most examples will work on Windows, Linux, and OS X, but a few are platform dependent. The exceptions are noted in the example descriptions.

The examples are provided with NO WARRANTY expressed or implied. They are licensed, like this book, under a Creative Commons Attribution - NonCommercial - NoDerivatives 4.0 International License.

Introduction

The Boost C++ Libraries are a collection of modern libraries based on the C++ standard. The source code is released under the Boost Software License, which allows anyone to use, modify, and distribute the libraries for free. The libraries are platform independent and support most popular compilers, as well as many that are less well known.

The Boost community is responsible for developing and publishing the Boost libraries. The community consists of a relatively large group of C++ developers from around the world coordinated through the web site www.boost.org as well as several mailing lists. GitHub is used as the code repository. The mission statement of the community is to develop and collect high-quality libraries that complement the standard library. Libraries that prove of value and become important for the development of C++ applications stand a good chance of being included in the standard library at some point.

The Boost community emerged around 1998, when the first version of the standard was released. It has grown continuously since then and now plays a big role in the standardization of C++. Even though there is no formal relationship between the Boost community and the standardization committee, some of the developers are active in both groups. The current version of the C++ standard, which was approved in 2011, includes libraries that have their roots in the Boost community.

The Boost libraries are a good choice to increase productivity in C++ projects when your requirements go beyond what is available in the standard library. Because the Boost libraries evolve faster than the standard library, you have earlier access to new developments, and you don't need to wait until those developments have been added to a new version of the standard library. Thus, you can benefit from progress made in the evolution of C++ faster, thanks to the Boost libraries.

Due to the excellent reputation of the Boost libraries, knowing them well can be a valuable skill for engineers. It is not unusual to be asked about the Boost libraries in an interview because developers who know these libraries are usually also familiar with the latest innovations in C++ and are able to write and understand modern C++ code.

Development Process

The development of the Boost libraries is only possible because individual developers and organizations vigorously support them. Because Boost only accepts libraries that solve existing problems, exhibit a convincing design, are developed using modern C++, and are documented in an understandable way, each Boost library has a lot of work behind it.

C++ developers can participate in the Boost community and propose new libraries. However, a lot of time and effort is required to convert an idea into a Boost library. Thus, it is vitally important to discuss requirements and possible solutions with other developers and potential users on the Boost mailing lists.

Besides new libraries, it is also possible to nominate existing C++ libraries for inclusion into Boost. However, because the requirements for these libraries are the same as for libraries explicitly developed for Boost, changes may be required before new libraries are accepted.

Whether or not a library gets accepted into Boost depends on the outcome of the review process. Developers of libraries can apply for a review, which usually takes about 10 days. During the review, other developers are asked to rate the library. Based on the number of positive and negative reviews, the review manager decides whether or not to accept the library into Boost. Since some reviewers may be seeing the proposed new library for the first time, it is not uncommon for modifications to be required.

If a library is rejected for technical reasons, it is still possible to revise the library and request a new review for an updated version. However, if the reason a library was rejected is because it does not solve any practical problem or because it provides an unconvincing solution to an existing problem, there is a good chance that it will be rejected in another review.

Because new libraries can be accepted at any time, a new version of the Boost libraries is released every three months. This makes sure that developers benefit from improvements in the Boost libraries regularly and promptly.

> Note
>
> This book is based on the Boost libraries 1.55.0 and 1.56.0. Version 1.55.0 was released in November 2013. Because the Boost community switched from Subversion to Git at the end of 2013 and needed time to adapt the release process, the next version, 1.56.0, was only released in August 2014. This happened two weeks before this book went into production. Since 1.55.0 was the latest version for nine months, this book is based on Boost 1.55.0. However, all examples have been tested with Boost 1.56.0, and all examples can be compiled with either Boost 1.55.0 or 1.56.0.

Installation

The Boost libraries come as source code. While most of the libraries consist solely of header files that can be used directly, some of the libraries require compilation. In order to make installation as easy as possible, an automated installation process based on Boost.Build is available. Instead of validating and compiling individual libraries separately, Boost.Build installs the complete set automatically. Boost.Build can be used with many operating systems and compilers and is able to compile each individual library based on appropriate configuration files.
To automatically install the Boost libraries with Boost.Build, the command-line program **bjam** is used. The Boost libraries ship this program as source code and not as an executable. Therefore, two steps are required to build and install the Boost libraries. After you download the Boost libraries, change to the Boost directory and enter the following commands on the command line:

1. Enter **`bootstrap`** on Windows and **`./bootstrap.sh`** on other platforms, such as Linux, to compile **bjam**. The script automatically searches for a C compiler to build **bjam**.

2. Then, enter **`bjam`** on Windows and **`./bjam`** on other platforms to start installing the Boost libraries.

You use **bootstrap** only once to build **bjam**. However, you might need to use **bjam** more often because **bjam** supports command-line options to build the Boost libraries in different ways. If you start **bjam** without any command-line options, a default configuration will be used. Because the default configuration is not always appropriate, you should know the most important command-line options:

- The command-line options `stage` and `install` specify whether the Boost libraries are installed in a subdirectory called `stage` or are made available system wide. The meaning of system wide depends on the operating system. On Windows, the target directory is `C:\Boost`; on Linux it is `/usr/local`. The target directory can also be specified with the `--prefix` option. Starting **bjam** without command-line options always means `stage`.

- If **bjam** is called without any command-line options, it will search for a suitable C++ compiler. A specific compiler can be selected using the `--toolset` option. To select Visual C++ 2013 on Windows, call **bjam** with `--toolset=msvc-12.0`. To select the GCC compiler on Linux, use `--toolset=gcc`.

- The command-line option `--build-type` determines which build types of the libraries are created. By default, this option is set to `minimal`, meaning that only release builds are created. This may become an issue for developers who want to create debug builds of their projects with Visual C++ or GCC. Because these compilers automatically try to link against the debug builds of the Boost libraries, an error message will be displayed. In this case the option `--build-type` should be set to `complete` to generate both debug and release builds of the Boost libraries. This can take quite some time, which is why `complete` is not the default.

- Boost libraries that have to be compiled are made available on Windows with file names that contain version numbers and various tokens. They make it possible, for example, to tell whether a library has been built as a debug or release variant. `libboost_atomic-vc120-mt-gd-1_56` is such a file name. This library was built with Visual C++ 2013. It belongs to the Boost libraries 1.56.0. It is a debug variant and can be used in multithreaded programs. With the command-line option `--layout`, **bjam** can be told to generate other file names. For example, if you set it to `system`, the same file would be called `libboost_atomic`. On Linux, `system` is the default setting. If you want file names on Linux to be the same as those generated on Windows by default, set `--layout` to `versioned`.

To create both debug and release builds of the Boost libraries with Visual C++ 2013 and install them in the directory `D:\Boost`, enter the following command:

```
bjam --toolset=msvc-12.0 --build-type=complete --prefix=D:\Boost install
```

To build them on Linux and install them in the default directory, the command would be:

```
bjam --toolset=gcc --build-type=complete install
```

There are many other command-line options that you can use to specify in detail how to compile the Boost libraries. Have a look at the following command:

```
bjam --toolset=msvc-12.0 debug release link=static runtime-link=shared ↩
   install
```

The `debug` and `release` options cause both debug and release builds to be generated. `link=static` only creates static libraries. `runtime-link=shared` specifies that the C++ runtime library is dynamically linked, which is the default setting for projects in Visual C++ 2013.

Overview

There are more than a hundred Boost libraries. This book discusses the following libraries in detail:

Table 1: Covered libraries

Boost library	**Standard**	**Short description**
Boost.Accumulators		Boost.Accumulators provides accumulators to which numbers can be added to get, for example, the mean or the standard deviation.
Boost.Algorithm		Boost.Algorithm provides various algorithms that complement the algorithms from the standard library.
Boost.Any		Boost.Any provides a type called `boost::any`, which can store objects of arbitrary types.
Boost.Array	TR1, C++11	Boost.Array makes it possible to treat C++ arrays like containers from the standard library.
Boost.Asio		Boost.Asio allows you to develop applications, such as network applications, that process data asynchronously.
Boost.Assign		Boost.Assign provides helper functions to add multiple values to a container without having to call member functions like `push_back()` repeatedly.
Boost.Atomic	C++11	Boost.Atomic defines the class `boost::atomic` to perform atomic operations on integral values. The library is used in multithreaded applications that need to share integral values between threads.
Boost.Bimap		Boost.Bimap provides a class called `boost::bimap`, which is similar to `std::map`. The crucial difference is that `boost::bimap` allows you to search for both key and value.
Boost.Bind	TR1, C++11	Boost.Bind is an adapter for passing functions as template parameters, even if the function signature is incompatible with the expected template parameter.
Boost.Chrono	C++11	Boost.Chrono defines numerous clocks to get values such as the current time or the CPU time.

Table 1: (continued)

Boost library	Standard	Short description
Boost.CircularBuffer		Boost.CircularBuffer offers a circular container with a constant memory size.
Boost.CompressedPair		Boost.CompressedPair provides the data structure `boost::compressed_pair`, which is similar to `std::pair` but needs less memory when template parameters are empty classes.
Boost.Container		Boost.Container defines all of the containers from the standard library, plus additional containers like `boost::container::slist`.
Boost.Conversion		Boost.Conversion provides two cast operators to perform downcasts and cross casts.
Boost.Coroutine		Boost.Coroutine makes it possible to use coroutines in C++. Coroutines are often used in other programming languages using the keyword `yield`.
Boost.DateTime		Boost.DateTime can be used to process, read, and write date and time values.
Boost.DynamicBitset		Boost.DynamicBitset provides a structure similar to `std::bitset`, except that it is configured at run time.
Boost.EnableIf	C++11	Boost.EnableIf makes it possible to overload functions based on type properties.
Boost.Exception		Boost.Exception allows you to add additional data to thrown exceptions so you can provide more data to `catch` handlers.
Boost.Filesystem		Boost.Filesystem provides a class to process paths and several functions to access files and directories.
Boost.Flyweight		Boost.Flyweight makes it easy to use the design pattern of the same name.
Boost.Foreach		Boost.Foreach provides a macro that is similar to the range-based `for` loop introduced with C++11.
Boost.Format		Boost.Format replaces the function `std::printf()` with a type-safe and extensible class, `boost::format`.
Boost.Function	TR1, C++11	Boost.Function simplifies the definition of function pointers.
Boost.Fusion		Boost.Fusion allows you to create heterogeneous containers – containers that can store elements of different types.
Boost.Graph		Boost.Graph provides algorithms for operations such as finding the shortest path between two points in a graph.
Boost.Heap		Boost.Heap provides many variants of the class `std::priority_queue` from the standard library.
Boost.Integer	C++11	Boost.Integer defines specialized types for integers that have been available to C developers since the standard C99 was released in 1999.
Boost.Interprocess		Boost.Interprocess uses shared memory to help applications communicate quickly and efficiently.
Boost.Intrusive		Boost.Intrusive defines containers that provide higher performance than containers from the standard library, but which also have special requirements for the objects they contain.
Boost.IOStreams		Boost.IOStreams provides streams and filters. Filters can be connected with a stream to, for example, write compressed data.
Boost.Lambda		Boost.Lambda allows you to define anonymous functions without C++11.

Table 1: (continued)

Boost library	Standard	Short description
Boost.LexicalCast		Boost.LexicalCast provides a cast operator to convert numbers to a string and vice versa.
Boost.Lockfree		Boost.Lockfree defines thread-safe containers that multiple threads may access concurrently.
Boost.Log		Boost.Log is the logging library in Boost.
Boost.MetaStateMachine		Boost.MetaStateMachine makes it possible to develop state machines as they are defined in the UML.
Boost.MinMax	C++11	Boost.MinMax provides an algorithm that can find the smallest and largest values in a container without calling `std::min()` and `std::max()`.
Boost.MPI		Boost.MPI provides a C++ interface for the MPI standard.
Boost.MultiArray		Boost.MultiArray simplifies working with multidimensional arrays.
Boost.MultiIndex		Boost.MultiIndex allows you to define new containers that can support multiple interfaces, such as the ones from `std::vector` and `std::map`.
Boost.NumericConversion		Boost.NumericConversion provides a cast operator to safely convert between values of different numeric types without generating an overflow condition.
Boost.Operators		Boost.Operators allows many operators to be automatically overloaded with the help of already defined operators.
Boost.Optional		Boost.Optional provides a class to denote optional return values. Functions that can't always return a result don't need to use special values like -1 or a null pointer anymore.
Boost.Parameter		Boost.Parameter lets you pass parameters to functions as name/value pairs like you can with programming languages like Python.
Boost.Phoenix		Boost.Phoenix makes it possible to create lambda functions without C++11. Unlike the C++11 lambda functions, the lambda functions from this library can be generic.
Boost.PointerContainer		Boost.PointerContainer provides containers that are optimized for managing dynamically allocated objects.
Boost.Pool		Boost.Pool is a library to manage memory. For example, Boost.Pool defines an allocator optimized for situations where you need to create and destroy many objects, all of the same size.
Boost.ProgramOptions		Boost.ProgramOptions allows an application to define and evaluate command-line options.
Boost.PropertyTree		Boost.PropertyTree provides a container that stores key/value pairs in a tree-like structure. This makes it easier to manage the kind of configuration data used by many applications.
Boost.Random	TR1, C++11	Boost.Random provides random number generators.
Boost.Range		Boost.Range introduces a concept called range that replaces the iterators usually received from containers with `begin()` and `end()`. Ranges makes it so you don't have to pass a pair of iterators to algorithms.
Boost.Ref	TR1, C++11	Boost.Ref provides adapters that allow you to pass references to objects that can't be copied to functions that pass parameters by copy.
Boost.Regex	TR1, C++11	Boost.Regex provides functions to search strings with regular expressions.

Table 1: (continued)

Boost library	Standard	Short description
Boost.ScopeExit		Boost.ScopeExit provides macros to define code blocks that are executed when the current scope ends. That way, resources can be released at the end of the current scope without having to use smart pointers or other classes.
Boost.Serialization		Boost.Serialization allows you to serialize objects and store them in files to be reloaded later.
Boost.Signals2		Boost.Signals2 is a framework for event handling based on the signal/slot concept, which associates functions with signals and automatically calls the appropriate function(s) when a signal is triggered.
Boost.SmartPointers	TR1, C++11 (partly)	Boost.SmartPointers provides a set of smart pointers that simplify managing dynamically allocated objects.
Boost.Spirit		Boost.Spirit allows you to generate parsers using a syntax similar to EBNF (Extended Backus-Naur-Form).
Boost.StringAlgorithms		Boost.StringAlgorithms provides many stand-alone functions to facilitate string handling.
Boost.Swap		Boost.Swap defines `boost::swap()`, which has the same function as `std::swap()`, but is optimized for many Boost libraries.
Boost.System	C++11	Boost.System offers a framework to process system- and application-specific error codes.
Boost.Thread	C++11	Boost.Thread allows you to develop multithreaded applications.
Boost.Timer		Boost.Timer defines clocks that let you measure code performance.
Boost.Tokenizer		Boost.Tokenizer allows you to iterate over tokens in a string.
Boost.Tribool		Boost.Tribool provides a type that, unlike bool, distinguishes three, rather than two, states.
Boost.Tuple	TR1, C++11	Boost.Tuple provides a generalized version of `std::pair` that can store an arbitrary number of values, not just two.
Boost.TypeTraits	TR1, C++11	Boost.TypeTraits provides functions to check properties of types.
Boost.Unordered	TR1, C++11	Boost.Unordered provides two hash containers: `boost::unordered_set` and `boost::unordered_map`.
Boost.Utility		Boost.Utility is a collection of various tools that are too small to have their own libraries and don't fit in another library.
Boost.Uuid		Boost.Uuid defines the class `boost::uuids::uuid` and generators to create UUIDs.
Boost.Variant		Boost.Variant permits the definition of types that, like `union`, group multiple types.
Boost.Xpressive		Boost.Xpressive makes it possible to search strings with regular expressions. Regular expressions are encoded as C++ code rather than as strings.

Presumably, the next version of the standard will be C++14. There are many project groups working on various topics for C++14. These activities are known as Technical Specifications (TS). For example, the File System TS works on an extension of the standard based on Boost.Filesystem to access files and directories. You can find more information on C++14 and the standardization of C++ at isocpp.org.

Part I

RAII and Memory Management

RAII stands for Resource Acquisition Is Initialization. The idea behind this idiom: for any resource acquired, an object should be initialized that will own that resource and close it in the destructor. Smart pointers are a prominent example of RAII. They help avoid memory leaks. The following libraries provide smart pointers and other tools to help you manage memory more easily.

- Boost.SmartPointers defines smart pointers. Some of them are provided by the C++11 standard library. Others are only available in Boost.
- Boost.PointerContainer defines containers to store dynamically allocated objects – objects that are created with `new`. Because the containers from this library destroy objects with `delete` in the destructor, no smart pointers need to be used.
- Boost.ScopeExit makes it possible to use the RAII idiom for any resources. While Boost.SmartPointers and Boost.PointerContainer can only be used with pointers to dynamically allocated objects, with Boost.ScopeExit no resource-specific classes need to be used.
- Boost.Pool has nothing to do with RAII, but it has a lot to do with memory management. This library defines numerous classes to provide memory to your program faster.

Chapter 1

Boost.SmartPointers

The library Boost.SmartPointers provides various smart pointers. They help you manage dynamically allocated objects, which are anchored in smart pointers that release the dynamically allocated objects in the destructor. Because destructors are executed when the scope of smart pointers ends, releasing dynamically objects is guaranteed. There can't be a memory leak if, for example, you forget to call `delete`.
The standard library has included the smart pointer `std::auto_ptr` since C++98, but since C++11, `std::auto_ptr` has been deprecated. With C++11, new and better smart pointers were introduced in the standard library. `std::shared_ptr` and `std::weak_ptr` originate from Boost.SmartPointers and are called `boost::shared_ptr` and `boost::weak_ptr` in this library. There is no counterpart to `std::unique_ptr`. However, Boost.SmartPointers provides four additional smart pointers – `boost::scoped_ptr`, `boost::scoped_array`, `boost::shared_array`, and `boost::intrusive_ptr` – which are not in the standard library.

1.1 Sole Ownership

`boost::scoped_ptr` is a smart pointer that is the sole owner of a dynamically allocated object. `boost::scoped_ptr` cannot be copied or moved. This smart pointer is defined in the header file `boost/scoped_ptr.hpp`.
A smart pointer of type `boost::scoped_ptr` can't transfer ownership of an object. Once initialized with an address, the dynamically allocated object is released when the destructor is executed or when the member function `reset()` is called.
Example 1.1 uses a smart pointer **p** with the type boost::scoped_ptr<int>. **p** is initialized with a pointer to a dynamically allocated object that stores the number 1. Via `operator*`, **p** is de-referenced and `1` written to standard output.

Example 1.1 Using `boost::scoped_ptr`

```
#include <boost/scoped_ptr.hpp>
#include <iostream>

int main()
{
  boost::scoped_ptr<int> p{new int{1}};
  std::cout << *p << '\n';
  p.reset(new int{2});
  std::cout << *p.get() << '\n';
  p.reset();
  std::cout << std::boolalpha << static_cast<bool>(p) << '\n';
}
```

With `reset()` a new address can be stored in the smart pointer. That way the example passes the address of a newly allocated int object containing the number 2 to **p**. With the call to `reset()`, the currently referenced object in **p** is automatically destroyed.
`get()` returns the address of the object anchored in the smart pointer. The example de-references the address returned by `get()` to write `2` to standard output.
`boost::scoped_ptr` overloads the operator `operator bool`. `operator bool` returns `true` if the smart

pointer contains a reference to an object – that is, if it isn't empty. The example writes `false` to standard output because **p** has been reset with a call to `reset()`.
The destructor of `boost::scoped_ptr` releases the referenced object with `delete`. That's why `boost::scoped_ptr` must not be initialized with the address of a dynamically allocated array, which would have to be released with `delete[]`. For arrays, Boost.SmartPointers provides the class `boost::scoped_array`.

Example 1.2 Using `boost::scoped_array`

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

int main()
{
  boost::scoped_array<int> p{new int[2]};
  *p.get() = 1;
  p[1] = 2;
  p.reset(new int[3]);
}
```

The smart pointer `boost::scoped_array` is used like `boost::scoped_ptr`. The crucial difference is that the destructor of `boost::scoped_array` uses the operator `delete[]` to release the contained object. Because this operator only applies to arrays, a `boost::scoped_array` must be initialized with the address of a dynamically allocated array.
`boost::scoped_array` is defined in `boost/scoped_array.hpp`.
`boost::scoped_array` provides overloads for `operator[]` and `operator bool`. Using `operator[]`, a specific element of the array can be accessed. Thus, an object of type `boost::scoped_array` behaves like the array it owns. Example 1.2 saves the number 2 as the second element in the array referred to by **p**.
Like `boost::scoped_ptr`, the member functions `get()` and `reset()` are provided to retrieve and reinitialize the address of the contained object.

1.2 Shared Ownership

The smart pointer `boost::shared_ptr` is similar to `boost::scoped_ptr`. The key difference is that `boost::shared_ptr` is not necessarily the exclusive owner of an object. Ownership can be shared with other smart pointers of type `boost::shared_ptr`. In such a case, the shared object is not released until the last copy of the shared pointer referencing the object is destroyed. Because `boost::shared_ptr` can share ownership, the smart pointer can be copied, which isn't possible with `boost::scoped_ptr`.
`boost::shared_ptr` is defined in the header file `boost/shared_ptr.hpp`.

Example 1.3 Using `boost::shared_ptr`

```
#include <boost/shared_ptr.hpp>
#include <iostream>

int main()
{
  boost::shared_ptr<int> p1{new int{1}};
  std::cout << *p1 << '\n';
  boost::shared_ptr<int> p2{p1};
  p1.reset(new int{2});
  std::cout << *p1.get() << '\n';
  p1.reset();
  std::cout << std::boolalpha << static_cast<bool>(p2) << '\n';
}
```

Example 1.3 uses two smart pointers, **p1** and **p2**, of the type `boost::shared_ptr`. **p2** is initialized with **p1** which means both smart pointers share ownership of the same int object. When `reset()` is called on **p1**, a new int object is anchored in **p1**. This doesn't mean that the existing int object is destroyed. Since it is also anchored in **p2**, it continues to exist. After the call to `reset()`, **p1** is the sole owner of the int object with the number 2 and **p2** the sole owner of the int object with the number 1.
`boost::shared_ptr` uses a reference counter internally. Only when `boost::shared_ptr` detects that the last copy of the smart pointer has been destroyed is the contained object released with `delete`.

Like `boost::scoped_ptr`, `boost::shared_ptr` overloads `operator bool()`, `operator*()`, and `operator->()`. The member functions `get()` and `reset()` are provided to retrieve the currently stored address or store a new one.
As a second parameter, a *deleter* can be passed to the constructor of `boost::shared_ptr`. The deleter must be a function or function object that accepts as its sole parameter a pointer of the type `boost::shared_ptr` was instantiated with. The deleter is called in the destructor instead of `delete`. This makes it possible to manage resources other than dynamically allocated objects in a `boost::shared_ptr`.

Example 1.4 `boost::shared_ptr` with a user-defined deleter

```
#include <boost/shared_ptr.hpp>
#include <Windows.h>

int main()
{
  boost::shared_ptr<void> handle(OpenProcess(PROCESS_SET_INFORMATION, FALSE,
    GetCurrentProcessId()), CloseHandle);
}
```

In Example 1.4 `boost::shared_ptr` is instantiated with void. The first parameter passed to the constructor is the return value from `OpenProcess()`. `OpenProcess()` is a Windows function to get a handle to a process. In the example, `OpenProcess()` returns a handle to the current process – to the example itself.
Windows uses handles to refer to resources. Once a resource isn't used anymore, the handle must be closed with `CloseHandle()`. The only parameter expected by `CloseHandle()` is the handle to close. In the example, `CloseHandle()` is passed as a second parameter to the constructor of `boost::shared_ptr`. `CloseHandle()` is the deleter for **`handle`**. When **`handle`** is destroyed at the end of `main()`, the destructor calls `CloseHandle()` to close the handle that was passed as a first parameter to the constructor.

> Note
>
> Example 1.4 only works because a Windows handle is defined as void*. If `OpenProcess()` didn't return a value of type void* and if `CloseHandle()` didn't expect a parameter of type void*, it wouldn't be possible to use `boost::shared_ptr` in this example. The deleter does not make `boost::shared_ptr` a silver bullet to manage arbitrary resources.

Example 1.5 Using `boost::make_shared`

```
#include <boost/make_shared.hpp>
#include <typeinfo>
#include <iostream>

int main()
{
  auto p1 = boost::make_shared<int>(1);
  std::cout << typeid(p1).name() << '\n';
  auto p2 = boost::make_shared<int[]>(10);
  std::cout << typeid(p2).name() << '\n';
}
```

Boost.SmartPointers provides a helper function `boost::make_shared()` in `boost/make_shared.hpp`. With `boost::make_shared()` you can create a smart pointer of type `boost::shared_ptr` without having to calling the constructor of `boost::shared_ptr` yourself.
The advantage of `boost::make_shared()` is that the memory for the object that has to be allocated dynamically and the memory for the reference counter used by the smart pointer internally can be reserved in one chunk. Using `boost::make_shared()` is more efficient than calling new to create a dynamically allocated object and calling new again in the constructor of `boost::shared_ptr` to allocate memory for the reference counter.
You can use `boost::make_shared()` for arrays, too. With the second call to `boost::make_shared()` in Example 1.5, an int array with ten elements is anchored in **`p2`**.
`boost::shared_ptr` has only supported arrays since Boost 1.53.0. `boost::shared_array` provides a smart pointer that is analogous to `boost::shared_ptr` in the same way that `boost::scoped_array` is analogous to

boost::scoped_ptr. When built with Visual C++ 2013 and Boost 1.53.0 or newer, Example 1.5 prints class boost::shared_ptr<int [0]> for **p2**.
Since Boost 1.53.0, boost::shared_ptr supports single objects and arrays and detects whether it has to release resources with delete or delete[]. Because boost::shared_ptr also overloads operator[] (since Boost 1.53.0), this smart pointer is an alternative for boost::shared_array.
boost::shared_array complements boost::shared_ptr: Since boost::shared_array calls delete[] in the destructor, this smart pointer can be used for arrays. For versions older than Boost 1.53.0, boost::shared_array had to be used for arrays because boost::shared_ptr didn't support arrays.
boost::shared_array is defined in boost/shared_array.hpp.

Example 1.6 Using boost::shared_array

```
#include <boost/shared_array.hpp>
#include <iostream>

int main()
{
  boost::shared_array<int> p1{new int[1]};
  {
    boost::shared_array<int> p2{p1};
    p2[0] = 1;
  }
  std::cout << p1[0] << '\n';
}
```

In Example 1.6, the smart pointers **p1** and **p2** share ownership of the dynamically allocated int array. When the array in **p2** is accessed with operator[] to store the number 1, the same array is accessed with **p1**. Thus, the example writes 1 to standard output.
Like boost::shared_ptr, boost::shared_array uses a reference counter. The dynamically allocated array is not released when **p2** is destroyed because **p1** still contains a reference to that array. The array is only destroyed at the end of main() when the scope ends for **p1**.
boost::shared_array also provides the member functions get() and reset(). Furthermore, it overloads the operator operator bool.

Example 1.7 boost::shared_ptr with BOOST_SP_USE_QUICK_ALLOCATOR

```
#define BOOST_SP_USE_QUICK_ALLOCATOR
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <ctime>

int main()
{
  boost::shared_ptr<int> p;
  std::time_t then = std::time(nullptr);
  for (int i = 0; i < 1000000; ++i)
    p.reset(new int{i});
  std::time_t now = std::time(nullptr);
  std::cout << now - then << '\n';
}
```

It can make sense to prefer a smart pointer like boost::shared_ptr over the ones from the standard library. Boost.SmartPointers supports macros to optimize the behavior of the smart pointers. Example 1.7 uses the macro BOOST_SP_USE_QUICK_ALLOCATOR to activate an allocator shipped with Boost.SmartPointers. This allocator manages memory chunks to reduce the number of calls to new and delete for reference counters. The example calls std::time() to measure the time before and after the loop. While the time it takes to execute the loop depends on the computer, the example may run faster with BOOST_SP_USE_QUICK_ALLOCATOR than without. The documentation for Boost.SmartPointers doesn't mention BOOST_SP_USE_QUICK_ALLOCATOR. Thus, you should profile your program and compare the results you get with and without BOOST_SP_USE_QUICK_ALLOCATOR.

Tip

In addition to `BOOST_SP_USE_QUICK_ALLOCATOR`, Boost.SmartPointers supports macros like `BOOST_SP_ENABLE_DEBUG_HOOKS`. The names of the macros start with BOOST_SP_ which makes it easy to search for them in the header files to get an overview on the available macros.

1.3 Special Smart Pointers

Every smart pointer introduced so far can be used individually in different scenarios. However, `boost::weak_ptr` only makes sense if used in conjunction with `boost::shared_ptr`. `boost::weak_ptr` is defined in `boost/weak_ptr.hpp`.

Example 1.8 Using `boost::weak_ptr`

```
#include <boost/shared_ptr.hpp>
#include <boost/weak_ptr.hpp>
#include <thread>
#include <functional>
#include <iostream>

void reset(boost::shared_ptr<int> &sh)
{
  sh.reset();
}

void print(boost::weak_ptr<int> &w)
{
  boost::shared_ptr<int> sh = w.lock();
  if (sh)
    std::cout << *sh << '\n';
}

int main()
{
  boost::shared_ptr<int> sh{new int{99}};
  boost::weak_ptr<int> w{sh};
  std::thread t1{reset, std::ref(sh)};
  std::thread t2{print, std::ref(w)};
  t1.join();
  t2.join();
}
```

`boost::weak_ptr` must be initialized with a `boost::shared_ptr`. Its most important member function is `lock()`. `lock()` returns a `boost::shared_ptr` that shares ownership with the shared pointer used to initialize the weak pointer. In case the shared pointer is empty, the returned pointer will be empty as well.
`boost::weak_ptr` makes sense whenever a function is expected to work with an object managed by a shared pointer, but the lifetime of the object does not depend on the function itself. The function can only use the object as long as it is owned by at least one shared pointer somewhere else in the program. In case the shared pointer is reset, the object cannot be kept alive because of an additional shared pointer inside the corresponding function.
Example 1.8 creates two threads in `main()`. The first thread executes the function `reset()`, which receives a reference to a shared pointer. The second thread executes the function `print()`, which receives a reference to a weak pointer. This weak pointer has been previously initialized with the shared pointer.
Once the program is launched, both `reset()` and `print()` are executed at the same time. However, the order of execution cannot be predicted. This leads to the potential problem of `reset()` destroying the object while it is being accessed by `print()`.
The weak pointer solves this issue as follows: invoking `lock()` returns a shared pointer that points to a valid object if one exists at the time of the call. If not, the shared pointer is set to 0 and is equivalent to a null pointer.

boost::weak_ptr itself does not have any impact on the lifetime of an object. To safely access the object within the print() function, lock() returns a boost::shared_ptr. This guarantees that even if a different thread attempts to release the object, it will continue to exist thanks to the returned shared pointer.

Example 1.9 Using boost::intrusive_ptr

```
#include <boost/intrusive_ptr.hpp>
#include <atlbase.h>
#include <iostream>

void intrusive_ptr_add_ref(IDispatch *p) { p->AddRef(); }
void intrusive_ptr_release(IDispatch *p) { p->Release(); }

void check_windows_folder()
{
  CLSID clsid;
  CLSIDFromProgID(CComBSTR{"Scripting.FileSystemObject"}, &clsid);
  void *p;
  CoCreateInstance(clsid, 0, CLSCTX_INPROC_SERVER, __uuidof(IDispatch), &p);
  boost::intrusive_ptr<IDispatch> disp{static_cast<IDispatch*>(p), false};
  CComDispatchDriver dd{disp.get()};
  CComVariant arg{"C:\\Windows"};
  CComVariant ret{false};
  dd.Invoke1(CComBSTR{"FolderExists"}, &arg, &ret);
  std::cout << std::boolalpha << (ret.boolVal != 0) << '\n';
}

int main()
{
  CoInitialize(0);
  check_windows_folder();
  CoUninitialize();
}
```

In general, boost::intrusive_ptr works the same as boost::shared_ptr. However, while boost::shared_ptr keeps track of the number of shared pointers referencing a particular object, the developer has to do this when using boost::intrusive_ptr. This can make sense if other classes already keep track of references.
boost::intrusive_ptr is defined in boost/intrusive_ptr.hpp.
Example 1.9 uses functions provided by COM and, thus, can only be built and run on Windows. COM objects are a good example for boost::intrusive_ptr because they track the number of pointers referencing them. The internal reference counter can be incremented or decremented by 1 with the member functions AddRef() and Release(). Once the counter reaches 0, the COM object is automatically destroyed.
The two member functions AddRef() and Release() are called from intrusive_ptr_add_ref() and intrusive_ptr_release(). Boost.Intrusive expects a developer to define these two functions, which are automatically called whenever a reference counter must be incremented or decremented. The parameter passed to these functions is a pointer to the type that was used to instantiate the class template boost::intrusive_ptr.
The COM object used in this example is called FileSystemObject and is available on Windows by default. It provides access to the underlying file system to, for example, check whether a given directory exists. In Example 1.9, the existence of a directory called C:\Windows is checked. How that works internally depends solely on COM and is irrelevant to the functionality of boost::intrusive_ptr. The crucial point is that once the intrusive pointer **disp** goes out of scope at the end of check_windows_folder(), the function intrusive_ptr_release() is called automatically. This in turn will decrement the internal reference counter of FileSystemObject to 0 and destroy the object.
The parameter false passed to the constructor of boost::intrusive_ptr prevents intrusive_ptr_add_ref() from being called. When a COM object is created with CoCreateInstance(), the counter is already set to 1. Therefore, it must not be incremented with intrusive_ptr_add_ref().

Chapter 2

Boost.PointerContainer

The library Boost.PointerContainer provides containers specialized to manage dynamically allocated objects. For example, with C++11 you can use std::vector<std::unique_ptr<int>> to create such a container. However, the containers from Boost.PointerContainer can provide some extra comfort.

Example 2.1 Using `boost::ptr_vector`

```
#include <boost/ptr_container/ptr_vector.hpp>
#include <iostream>

int main()
{
  boost::ptr_vector<int> v;
  v.push_back(new int{1});
  v.push_back(new int{2});
  std::cout << v.back() << '\n';
}
```

The class `boost::ptr_vector` basically works like std::vector<std::unique_ptr<int>> (see Example 2.1). However, because `boost::ptr_vector` knows that it stores dynamically allocated objects, member functions like `back()` return a reference to a dynamically allocated object and not a pointer. Thus, the example writes `2` to standard output.

Example 2.2 illustrates another reason to use a specialized container. The example stores dynamically allocated variables of type int in a `boost::ptr_set` and a `std::set`. `std::set` is used together with `std::unique_ptr`.

Example 2.2 `boost::ptr_set` with intuitively correct order

```
#include <boost/ptr_container/ptr_set.hpp>
#include <boost/ptr_container/indirect_fun.hpp>
#include <set>
#include <memory>
#include <functional>
#include <iostream>

int main()
{
  boost::ptr_set<int> s;
  s.insert(new int{2});
  s.insert(new int{1});
  std::cout << *s.begin() << '\n';

  std::set<std::unique_ptr<int>, boost::indirect_fun<std::less<int>>> v;
  v.insert(std::unique_ptr<int>(new int{2}));
  v.insert(std::unique_ptr<int>(new int{1}));
  std::cout << **v.begin() << '\n';
}
```

With `boost::ptr_set`, the order of the elements depends on the int values. `std::set` compares pointers of type `std::unique_ptr` and not the variables the pointers refer to. To make `std::set` sort the elements based on int values, the container must be told how to compare elements. In Example 2.2, `boost::indirect_fun` (provided by Boost.PointerContainer) is used. With `boost::indirect_fun`, `std::set` is told that elements shouldn't be sorted based on pointers of type `std::unique_ptr`, but instead based on the int values the pointers refer to. That's why the example displays `1` twice.

Besides `boost::ptr_vector` and `boost::ptr_set`, there are other containers available for managing dynamically allocated objects. Examples of these additional containers include `boost::ptr_deque`, `boost::ptr_list`, `boost::ptr_map`, `boost::ptr_unordered_set`, and `boost::ptr_unordered_map`. These containers correspond to the well-known containers from the standard library.

Example 2.3 Inserters for containers from Boost.PointerContainer

```
#include <boost/ptr_container/ptr_vector.hpp>
#include <boost/ptr_container/ptr_inserter.hpp>
#include <array>
#include <algorithm>
#include <iostream>

int main()
{
  boost::ptr_vector<int> v;
  std::array<int, 3> a{{0, 1, 2}};
  std::copy(a.begin(), a.end(), boost::ptr_container::ptr_back_inserter(v));
  std::cout << v.size() << '\n';
}
```

Boost.PointerContainer provides inserters for its containers. They are defined in the namespace `boost::ptr_container`. To have access to the inserters, you must include the header file `boost/ptr_container/ptr_inserter.hpp`.

Example 2.3 uses the function `boost::ptr_container::ptr_back_inserter()`, which creates an inserter of type `boost::ptr_container::ptr_back_insert_iterator`. This inserter is passed to `std::copy()` to copy all numbers from the array **`a`** to the vector **`v`**. Because **`v`** is a container of type `boost::ptr_vector`, which expects addresses of dynamically allocated int objects, the inserter creates copies with `new` on the heap and adds the addresses to the container.

In addition to `boost::ptr_container::ptr_back_inserter()`, Boost.PointerContainer provides the functions `boost::ptr_container::ptr_front_inserter()` and `boost::ptr_container::ptr_inserter()` to create corresponding inserters.

Chapter 3

Boost.ScopeExit

The library Boost.ScopeExit makes it possible to use RAII without resource-specific classes.
Boost.ScopeExit provides the macro BOOST_SCOPE_EXIT, which can be used to define something that looks like a local function but doesn't have a name. However, it does have a parameter list in parentheses and a block in braces.
The header file boost/scoped_exit.hpp must be included to use BOOST_SCOPE_EXIT.

Example 3.1 Using BOOST_SCOPE_EXIT

```
#include <boost/scope_exit.hpp>
#include <iostream>

int *foo()
{
  int *i = new int{10};
  BOOST_SCOPE_EXIT(&i)
  {
    delete i;
    i = 0;
  } BOOST_SCOPE_EXIT_END
  std::cout << *i << '\n';
  return i;
}

int main()
{
  int *j = foo();
  std::cout << j << '\n';
}
```

The parameter list for the macro contains variables from the outer scope which should be accessible in the block. The variables are passed by copy. To pass a variable by reference, it must be prefixed with an ampersand, as in Example 3.1.
Code in the block can only access variables from the outer scope if the variables are in the parameter list.
BOOST_SCOPE_EXIT is used to define a block that will be executed when the scope the block is defined in ends.
In Example 3.1 the block defined with BOOST_SCOPE_EXIT is executed just before foo() returns.
BOOST_SCOPE_EXIT can be used to benefit from RAII without having to use resource-specific classes. foo() uses new to create an int variable. In order to free the variable, a block that calls delete is defined with BOOST_SCOPE_EXIT. This block is guaranteed to be executed even if, for example, the function returns early because of an exception. In Example 3.1, BOOST_SCOPE_EXIT is as good as a smart pointer.
Please note that the variable **i** is set to 0 at the end of the block defined by BOOST_SCOPE_EXIT. **i** is then returned by foo() and written to the standard output stream in main(). However, the example doesn't display 0. **j** is set to a random value – namely the address where the int variable was before the memory was freed. The block behind BOOST_SCOPE_EXIT got a reference to **i** and freed the memory. But since the block is executed at the end of foo(), the assignment of 0 to **i** is too late. The return value of foo() is a copy of **i** that gets created before **i** is set to 0.
You can ignore Boost.ScopeExit if you use a C++11 development environment. In that case, you can use RAII without resource-specific classes with the help of lambda functions.

Example 3.2 defines the class `scope_exit` whose constructor accepts a function. This function is called by the destructor. Furthermore, a helper function, `make_scope_exit()`, is defined that makes it possible to instantiate `scope_exit` without having to specify a template parameter.
In `foo()` a lambda function is passed to `make_scope_exit()`. The lambda function looks like the block after `BOOST_SCOPE_EXIT` in Example 3.1: The dynamically allocated int variable whose address is stored in **`i`** is freed with `delete`. Then 0 is assigned to **`i`**.
The example does the same thing as the previous one. Not only is the int variable deleted, but **`j`** is not set to 0 either when it is written to the standard output stream.

Example 3.2 Boost.ScopeExit with C++11 lambda functions

```
#include <iostream>
#include <utility>

template <typename T>
struct scope_exit
{
  scope_exit(T &&t) : t_{std::move(t)} {}
  ~scope_exit() { t_(); }
  T t_;
};

template <typename T>
scope_exit<T> make_scope_exit(T &&t) { return scope_exit<T>{
  std::move(t)}; }

int *foo()
{
  int *i = new int{10};
  auto cleanup = make_scope_exit([&i]() mutable { delete i; i = 0; });
  std::cout << *i << '\n';
  return i;
}

int main()
{
  int *j = foo();
  std::cout << j << '\n';
}
```

Example 3.3 introduces some peculiarities of `BOOST_SCOPE_EXIT`:

- When `BOOST_SCOPE_EXIT` is used to define more than one block in a scope, the blocks are executed in reverse order. Example 3.3 displays `first` followed by `last`.
- If no variables will be passed to `BOOST_SCOPE_EXIT`, you need to specify `void`. The parentheses must not be empty.
- If you use `BOOST_SCOPE_EXIT` in a member function and you need to pass a pointer to the current object, you must use **`this_`**, not `this`.

Example 3.3 displays `first`, `last`, and `20` in that order.

Example 3.3 Peculiarities of `BOOST_SCOPE_EXIT`

```
#include <boost/scope_exit.hpp>
#include <iostream>

struct x
{
  int i;

  void foo()
  {
    i = 10;
    BOOST_SCOPE_EXIT(void)
    {
      std::cout << "last\n";
    } BOOST_SCOPE_EXIT_END
    BOOST_SCOPE_EXIT(this_)
    {
      this_->i = 20;
      std::cout << "first\n";
    } BOOST_SCOPE_EXIT_END
  }
};

int main()
{
  x obj;
  obj.foo();
  std::cout << obj.i << '\n';
}
```

Chapter 4

Boost.Pool

Boost.Pool is a library that contains a few classes to manage memory. While C++ programs usually use `new` to allocate memory dynamically, the details of how memory is provided depends on the implementation of the standard library and the operating system. With Boost.Pool you can, for example, accelerate memory management to provide memory to your program faster.

Boost.Pool doesn't change the behavior of `new` or of the operating system. Boost.Pool works because the managed memory is requested from the operating system first – for example using `new`. From the outside, your program has already allocated the memory, but internally, the memory isn't required yet and is handed over to Boost.Pool to manage it.

Boost.Pool partitions memory segments with the same size. Every time you request memory from Boost.Pool, the library accesses the next free segment and assigns memory from that segment to you. The entire segment is then marked as used, no matter how many bytes you actually need from that segment.

This memory management concept is called *simple segregated storage*. This is the only concept supported by Boost.Pool. It is especially useful if many objects of the same size have to be created and destroyed frequently. In this case the required memory can be provided and released quickly.

Boost.Pool provides the class `boost::simple_segregated_storage` to create and manage segregated memory. `boost::simple_segregated_storage` is a low-level class that you usually will not use in your programs directly. It is only used in Example 4.1 to illustrate simple segregated storage. All other classes from Boost.Pool are internally based on `boost::simple_segregated_storage`.

Example 4.1 Using `boost::simple_segregated_storage`

```
#include <boost/pool/simple_segregated_storage.hpp>
#include <vector>
#include <cstddef>

int main()
{
  boost::simple_segregated_storage<std::size_t> storage;
  std::vector<char> v(1024);
  storage.add_block(&v.front(), v.size(), 256);

  int *i = static_cast<int*>(storage.malloc());
  *i = 1;

  int *j = static_cast<int*>(storage.malloc_n(1, 512));
  j[10] = 2;

  storage.free(i);
  storage.free_n(j, 1, 512);
}
```

The header file `boost/pool/simple_segregated_storage.hpp` must be included to use the class template `boost::simple_segregated_storage`. Example 4.1 passes std::size_t as the template parameter. This parameter specifies which type should be used for numbers passed to member functions of `boost::simple_segregated_storage` to refer, for example, to the size of a segment. The practical relevance of this template parameter is rather low.

More interesting are the member functions called on `boost::simple_segregated_storage`. First, `add_block()` is called to pass a memory block with 1024 bytes to **`storage`**. The memory is provided by the vector **`v`**. The third parameter passed to `add_block()` specifies that the memory block should be partitioned in segments with 256 bytes each. Because the total size of the memory block is 1024 bytes, the memory managed by **`storage`** consists of four segments.
The calls to `malloc()` and `malloc_n()` request memory from **`storage`**. While `malloc()` returns a pointer to a free segment, `malloc_n()` returns a pointer to one or more contiguous segments that provide as many bytes in one block as requested. Example 4.1 requests a block with 512 bytes with `malloc_n()`. This call consumes two segments, since each segment is 256 bytes. After the calls to `malloc()` and `malloc_n()`, **`storage`** has only one unused segment left.
At the end of the example, all segments are released with `free()` and `free_n()`. After these two calls, all segments are available and could be requested again with `malloc()` or `malloc_n()`.
You usually don't use `boost::simple_segregated_storage` directly. Boost.Pool provides other classes that allocate memory automatically without requiring you to allocate memory yourself and pass it to `boost::simple_segregated_storage`.

Example 4.2 Using `boost::object_pool`

```
#include <boost/pool/object_pool.hpp>

int main()
{
  boost::object_pool<int> pool;

  int *i = pool.malloc();
  *i = 1;

  int *j = pool.construct(2);

  pool.destroy(i);
  pool.destroy(j);
}
```

Example 4.2 uses the class `boost::object_pool`, which is defined in `boost/pool/object_pool.hpp`. Unlike `boost::simple_segregated_storage`, `boost::object_pool` knows the type of the objects that will be stored in memory. **`pool`** in Example 4.2 is simple segregated storage for int values. The memory managed by **`pool`** consists of segments, each of which is the size of an int – 4 bytes for example.
Another difference is that you don't need to provide memory to `boost::object_pool`. `boost::object_pool` allocates memory automatically. In Example 4.2, the call to `malloc()` makes **`pool`** allocate a memory block with space for 32 int values. `malloc()` returns a pointer to the first of these 32 segments that an int value can fit into exactly.
Please note that `malloc()` returns a pointer of type int*. Unlike `boost::simple_segregated_storage` in Example 4.1, no cast operator is required.
`construct()` is similar to `malloc()` but initializes an object via a call to the constructor. In Example 4.2, **`j`** refers to an int object initialized with the value 2.
Please note that **`pool`** can return a free segment from the pool of 32 segments when `construct()` is called. The call to `construct()` does not make Example 4.2 request memory from the operating system.
The last member function called in Example 4.2 is `destroy()`, which releases an int object.

Example 4.3 Changing the segment size with `boost::object_pool`

```
#include <boost/pool/object_pool.hpp>
#include <iostream>

int main()
{
  boost::object_pool<int> pool{32, 0};
  pool.construct();
  std::cout << pool.get_next_size() << '\n';
  pool.set_next_size(8);
}
```

You can pass two parameters to the constructor of `boost::object_pool`. The first parameter sets the size of

the memory block that `boost::object_pool` will allocate when the first segment is requested with a call to `malloc()` or `construct()`. The second parameter sets the maximum size of the memory block to allocate.
If `malloc()` or `construct()` are called so often that all segments in a memory block are used, the next call to one of these member functions will cause `boost::object_pool` to allocate a new memory block, which will be twice as big as the previous one. The size will double each time a new memory block is allocated by `boost::object_pool`. `boost::object_pool` can manage an arbitrary number of memory blocks, but their sizes will grow exponentially. The second constructor parameter lets you limit the growth.
The default constructor of `boost::object_pool` does the same as what the call to the constructor in Example 4.3 does. The first parameter sets the size of the memory block to 32 int values. The second parameter specifies that there is no maximum size. If 0 is passed, `boost::object_pool` can double the size of the memory block indefinitely.
The call to `construct()` in Example 4.3 makes **`pool`** allocate a memory block of 32 int values. **`pool`** can serve up to 32 calls to `malloc()` or `construct()` without requesting memory from the operating system. If more memory is required, the next memory block to allocate will have space for 64 int values.
`get_next_size()` returns the size of the next memory block to allocate. `set_next_size()` lets you set the size of the next memory block. In Example 4.3 `get_next_size()` returns 64. The call to `set_next_size()` changes the size of the next memory block to allocate from 64 to 8 int values. With `set_next_size()` the size of the next memory block can be changed directly. If you only want to set a maximum size, pass it via the second parameter to the constructor.
With `boost::singleton_pool`, Boost.Pool provides a class between `boost::simple_segregated_storage` and `boost::object_pool` (see Example 4.4).

Example 4.4 Using `boost::singleton_pool`

```
#include <boost/pool/singleton_pool.hpp>

struct int_pool {};
typedef boost::singleton_pool<int_pool, sizeof(int)> singleton_int_pool;

int main()
{
  int *i = static_cast<int*>(singleton_int_pool::malloc());
  *i = 1;

  int *j = static_cast<int*>(singleton_int_pool::ordered_malloc(10));
  j[9] = 2;

  singleton_int_pool::release_memory();
  singleton_int_pool::purge_memory();
}
```

`boost::singleton_pool` is defined in `boost/pool/singleton_pool.hpp`. This class is similar to `boost::simple_segregated_storage` since it also expects the segment size as a template parameter but not the type of the objects to store. That's why member functions such as `ordered_malloc()` and `malloc()`return a pointer of type void*, which must be cast explicitly.
This class is also similar to `boost::object_pool` because it allocates memory automatically. The size of the next memory block and an optional maximum size are passed as template parameters. Here `boost::singleton_pool` differs from `boost::object_pool`: you can't change the size of the next memory block in `boost::singleton_pool` at run time.
You can create multiple objects with `boost::singleton_pool` if you want to manage several memory pools. The first template parameter passed to `boost::singleton_pool` is a *tag*. The tag is an arbitrary type that serves as a name for the memory pool. Example 4.4 uses the structure `int_pool` as a tag to highlight that `singleton_int_pool` is a pool that manages int values. Thanks to tags, multiple singletons can manage different memory pools, even if the second template parameter for the size is the same. The tag has no purpose other than creating separate instances of `boost::singleton_pool`.
`boost::singleton_pool` provides two member functions to release memory: `release_memory()` releases all memory blocks that aren't used at the moment, and `purge_memory()` releases all memory blocks – including those currently being used. The call to `purge_memory()` resets `boost::singleton_pool`.
`release_memory()` and `purge_memory()` return memory to the operating system. To return memory to `boost::singleton_pool` instead of the operating system, call member functions such as `free()` or `ordered_free()`.

boost::object_pool and boost::singleton_pool allow you to request memory explicitly. You do this by calling member functions such as malloc() or construct(). Boost.Pool also provides the class boost::pool_allocator, which you can pass as an allocator to containers (see Example 4.5).

Example 4.5 Using boost::pool_allocator

```
#include <boost/pool/pool_alloc.hpp>
#include <vector>

int main()
{
  std::vector<int, boost::pool_allocator<int>> v;
  for (int i = 0; i < 1000; ++i)
    v.push_back(i);

  v.clear();
  boost::singleton_pool<boost::pool_allocator_tag, sizeof(int)>::
    purge_memory();
}
```

boost::pool_allocator is defined in boost/pool/pool_alloc.hpp. The class is an allocator that is usually passed as a second template parameter to containers from the standard library. The allocator provides memory required by the container.

boost::pool_allocator is based on boost::singleton_pool. To release memory, you have to use a tag to access boost::singleton_pool and call purge_memory() or release_memory(). Example 4.5 uses the tag boost::pool_allocator_tag. This tag is defined by Boost.Pool and is used by boost::pool_allocator for the internal boost::singleton_pool.

When Example 4.5 calls push_back() the first time, **v** accesses the allocator to get the requested memory. Because the allocator boost::pool_allocator is used, a memory block with space for 32 int values is allocated. **v** receives the pointer to the first segment in that memory block that has the size of an int. With every subsequent call to push_back(), another segment is used from the memory block until the allocator detects that a bigger memory block is required.

Please note that you should call clear() on a container before you release memory with purge_memory() (see Example 4.5). A call to purge_memory() releases memory but doesn't notify the container that it doesn't own the memory anymore. A call to release_memory() is less dangerous because it only releases memory blocks that aren't in use.

Boost.Pool also provides an allocator called boost::fast_pool_allocator (see Example 4.6).

Example 4.6 Using boost::fast_pool_allocator

```
#define BOOST_POOL_NO_MT
#include <boost/pool/pool_alloc.hpp>
#include <list>

int main()
{
  typedef boost::fast_pool_allocator<int,
    boost::default_user_allocator_new_delete,
    boost::details::pool::default_mutex,
    64, 128> allocator;

  std::list<int, allocator> l;
  for (int i = 0; i < 1000; ++i)
    l.push_back(i);

  l.clear();
  boost::singleton_pool<boost::fast_pool_allocator_tag, sizeof(int)>::
    purge_memory();
}
```

Both allocators are used in the same way, but boost::pool_allocator should be preferred if you are requesting contiguous segments. boost::fast_pool_allocator can be used if segments are requested one by one. Grossly simplified: You use boost::pool_allocator for std::vector and boost::fast_pool_allocator for std::list.

Example 4.6 illustrates which template parameters can be passed to `boost::fast_pool_allocator`. `boost::pool_allocator` accepts the same parameters.
`boost::default_user_allocator_new_delete` is a class that allocates memory blocks with `new` and releases them with `delete[]`. You can also use `boost::default_user_allocator_malloc_free`, which calls `malloc()` and `free()`.
boost::details::pool::default_mutex is a type definition that is set to `boost::mutex` or `boost::details::pool::null_mutex`. `boost::mutex` is the default type that supports multiple threads requesting memory from the allocator. If the macro `BOOST_POOL_NO_MT` is defined as in Example 4.6, multithreading support for Boost.Pool is disabled. The allocator in Example 4.6 uses a null mutex.
The last two parameters passed to `boost::fast_pool_allocator` in Example 4.6 set the size of the first memory block and the maximum size of memory blocks to allocate.

Part II

String Handling

The following libraries provide tools to simplify working with strings.

- Boost.StringAlgorithms defines many algorithms specifically for strings. For example, you will find algorithms to convert strings to lower or upper case.
- Boost.LexicalCast provides a cast operator to convert a number to a string or vice versa. The library uses stringstreams internally but might be optimized for conversions between certain types.
- Boost.Format provides a type-safe alternative for `std::printf()`. Like Boost.LexicalCast, this library uses stringstreams internally. Boost.Format is extensible and supports user-defined types if output stream operators are defined.
- Boost.Regex and Boost.Xpressive are libraries to search within strings with regular expressions. While Boost.Regex expects regular expressions written as strings, Boost.Xpressive lets you write them as C++ code.
- Boost.Tokenizer makes it possible to iterate over substrings in a string.
- Boost.Spirit can be used to develop parsers based on rules similar to Extended Backus-Naur-Form.

Chapter 5

Boost.StringAlgorithms

The Boost.StringAlgorithms library provides many free-standing functions for string manipulation. Strings can be of type `std::string`, `std::wstring`, or any other instance of the class template `std::basic_string`. This includes the string classes `std::u16string` and `std::u32string` introduced with C++11.
The functions are categorized within different header files. For example, functions converting from uppercase to lowercase are defined in `boost/algorithm/string/case_conv.hpp`. Because Boost.StringAlgorithms consists of more than 20 different categories and as many header files, `boost/algorithm/string.hpp` acts as the common header including all other header files for convenience.

Example 5.1 Converting strings to uppercase

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout << to_upper_copy(s) << '\n';
}
```

The function `boost::algorithm::to_upper_copy()` converts a string to uppercase, and `boost::algorithm::to_lower_copy()` converts a string to lowercase. Both functions return a copy of the input string, converted to the specified case. To convert the string in place, use the functions `boost::algorithm::to_upper()` or `boost::algorithm::to_lower()`.
Example 5.1 converts the string “Boost C++ Libraries” to uppercase using `boost::algorithm::to_upper_copy()`. The example writes `BOOST C++ LIBRARIES` to standard output.
Functions from Boost.StringAlgorithms consider locales. Functions like `boost::algorithm::to_upper_copy()` use the global locale if no locale is passed explicitly as a parameter.

Example 5.2 Converting a string to uppercase with a locale

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <locale>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ k\xfct\xfcphaneleri";
  std::string upper_case1 = to_upper_copy(s);
  std::string upper_case2 = to_upper_copy(s, std::locale{"Turkish"});
  std::locale::global(std::locale{"Turkish"});
  std::cout << upper_case1 << '\n';
  std::cout << upper_case2 << '\n';
}
```

Example 5.2 calls `boost::algorithm::to_upper_copy()` twice to convert the Turkish string “Boost C++ kütüphaneleri” to uppercase. The first call to `boost::algorithm::to_upper_copy()` uses the global locale, which in this case is the C locale. In the C locale, there is no uppercase mapping for characters with umlauts, so the output will look like this: `BOOST C++ KüTüPHANELERI`.
The Turkish locale is passed to the second call to `boost::algorithm::to_upper_copy()`. Since this locale does have uppercase equivalents for umlauts, the entire string can be converted to uppercase. Therefore, the second call to `boost::algorithm::to_upper_copy()` correctly converts the string, which looks like this: `BOOST C++ KÜTÜPHANELERI`.

Note

If you want to run the example on a POSIX operating system, replace “Turkish” with “tr_TR”, and make sure the Turkish locale is installed.

Example 5.3 Algorithms to remove characters from a string

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout << erase_first_copy(s, "s") << '\n';
  std::cout << erase_nth_copy(s, "s", 0) << '\n';
  std::cout << erase_last_copy(s, "s") << '\n';
  std::cout << erase_all_copy(s, "s") << '\n';
  std::cout << erase_head_copy(s, 5) << '\n';
  std::cout << erase_tail_copy(s, 9) << '\n';
}
```

Boost.StringAlgorithms provides several functions you can use to delete individual characters from a string (see Example 5.3). For example, `boost::algorithm::erase_all_copy()` will remove all occurrences of a particular character from a string. To remove only the first occurrence of the character, use `boost::algorithm::erase_first_copy()` instead. To shorten a string by a specific number of characters on either end, use the functions `boost::algorithm::erase_head_copy()` and `boost::algorithm::erase_tail_copy()`.

Example 5.4 Searching for substrings with `boost::algorithm::find_first()`

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  boost::iterator_range<std::string::iterator> r = find_first(s, "C++");
  std::cout << r << '\n';
  r = find_first(s, "xyz");
  std::cout << r << '\n';
}
```

Functions such as `boost::algorithm::find_first()`, `boost::algorithm::find_last()`, `boost::algorithm::find_nth()`, `boost::algorithm::find_head()` and `boost::algorithm::find_tail()` are available to find strings within strings.

All of these functions return a pair of iterators of type `boost::iterator_range`. This class originates from Boost.Range, which implements a range concept based on the iterator concept. Because the operator `operator<<` is overloaded for `boost::iterator_range`, the result of the individual search algorithm can be written directly to standard output. Example 5.4 prints `C++` for the first result and an empty string for the second one.

Example 5.5 Concatenating strings with `boost::algorithm::join()`

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<std::string> v{"Boost", "C++", "Libraries"};
  std::cout << join(v, " ") << '\n';
}
```

A container of strings is passed as the first parameter to the function `boost::algorithm::join()`, which concatenates them separated by the second parameter. Example 5.5 will output `Boost C++ Libraries`.

Example 5.6 Algorithms to replace characters in a string

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout << replace_first_copy(s, "+", "-") << '\n';
  std::cout << replace_nth_copy(s, "+", 0, "-") << '\n';
  std::cout << replace_last_copy(s, "+", "-") << '\n';
  std::cout << replace_all_copy(s, "+", "-") << '\n';
  std::cout << replace_head_copy(s, 5, "BOOST") << '\n';
  std::cout << replace_tail_copy(s, 9, "LIBRARIES") << '\n';
}
```

Like the functions for searching strings or removing characters from strings, Boost.StringAlgorithms also provides functions for replacing substrings within a string. These include the following functions: `boost::algorithm::replace_first_copy()`, `boost::algorithm::replace_nth_copy()`, `boost::algorithm::replace_last_copy()`, `boost::algorithm::replace_all_copy()`, `boost::algorithm::replace_head_copy()` and `boost::algorithm::replace_tail_copy()`. They can be applied in the same way as the functions for searching and removing, except they require an additional parameter – the replacement string (see Example 5.6).

Example 5.7 Algorithms to trim strings

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "\t Boost C++ Libraries \t";
  std::cout << "_" << trim_left_copy(s) << "_\n";
  std::cout << "_" << trim_right_copy(s) << "_\n";
  std::cout << "_" << trim_copy(s) << "_\n";
}
```

To remove spaces on either end of a string, use `boost::algorithm::trim_left_copy()`, `boost::algorithm::trim_right_copy()` and `boost::algorithm::trim_copy()` (see Example 5.7). The global locale determines which characters are considered to be spaces.
Boost.StringAlgorithms lets you provide a predicate as an additional parameter for different functions to determine which characters of the string the function is applied to. The versions with predicates are: `boost::algorithm::trim_right_copy_if()`, `boost::algorithm::trim_left_copy_if()`, and `boost::algorithm::trim_copy_if()`.

Example 5.8 Creating predicates with `boost::algorithm::is_any_of()`

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "--Boost C++ Libraries--";
  std::cout << trim_left_copy_if(s, is_any_of("-")) << '\n';
  std::cout << trim_right_copy_if(s, is_any_of("-")) << '\n';
  std::cout << trim_copy_if(s, is_any_of("-")) << '\n';
}
```

Example 5.8 uses another function called `boost::algorithm::is_any_of()`, which is a helper function to create a predicate that checks whether a certain character – passed as parameter to `is_any_of()` – exists in a string. With `boost::algorithm::is_any_of()`, the characters for trimming a string can be specified. Example 5.8 uses the hyphen character.
Boost.StringAlgorithms provides many helper functions that return commonly used predicates.

Example 5.9 Creating predicates with `boost::algorithm::is_digit()`

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "123456789Boost C++ Libraries123456789";
  std::cout << trim_left_copy_if(s, is_digit()) << '\n';
  std::cout << trim_right_copy_if(s, is_digit()) << '\n';
  std::cout << trim_copy_if(s, is_digit()) << '\n';
}
```

The predicate returned by `boost::algorithm::is_digit()` tests whether a character is numeric. In Example 5.9, `boost::algorithm::is_digit()` is used to remove digits from the string **`s`**.
Boost.StringAlgorithms also provides helper functions to check whether a character is uppercase or lowercase: `boost::algorithm::is_upper()` and `boost::algorithm::is_lower()`. All of these functions use the global locale by default, unless you pass in a different locale as a parameter.
Besides the predicates that verify individual characters of a string, Boost.StringAlgorithms also offers functions that work with strings instead (see Example 5.10).

Example 5.10 Algorithms to compare strings with others

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::cout.setf(std::ios::boolalpha);
  std::cout << starts_with(s, "Boost") << '\n';
  std::cout << ends_with(s, "Libraries") << '\n';
  std::cout << contains(s, "C++") << '\n';
  std::cout << lexicographical_compare(s, "Boost") << '\n';
}
```

The `boost::algorithm::starts_with()`, `boost::algorithm::ends_with()`, `boost::algorithm::contains()`, and `boost::algorithm::lexicographical_compare()` functions compare two individual strings.
Example 5.11 introduces a function that splits a string into smaller parts.

Example 5.11 Splitting strings with `boost::algorithm::split()`

```
#include <boost/algorithm/string.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  std::vector<std::string> v;
  split(v, s, is_space());
  std::cout << v.size() << '\n';
}
```

With `boost::algorithm::split()`, a given string can be split based on a delimiter. The substrings are stored in a container. The function requires as its third parameter a predicate that tests each character and checks whether the string should be split at the given position. Example 5.11 uses the helper function `boost::algorithm::is_space()` to create a predicate that splits the string at every space character.
Many of the functions introduced in this chapter have versions that ignore the case of the string. These versions typically have the same name, except for a leading **`i`**. For example, the equivalent function to `boost::algorithm::erase_all_copy()` is `boost::algorithm::ierase_all_copy()`.
Finally, many functions of Boost.StringAlgorithms also support regular expressions. Example 5.12 uses the function `boost::algorithm::find_regex()` to search for a regular expression.

Example 5.12 Searching strings with `boost::algorithm::find_regex()`

```
#include <boost/algorithm/string.hpp>
#include <boost/algorithm/string/regex.hpp>
#include <string>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::string s = "Boost C++ Libraries";
  boost::iterator_range<std::string::iterator> r =
    find_regex(s, boost::regex{"\\w\\+\\+"});
  std::cout << r << '\n';
}
```

In order to use the regular expression, the program accesses a class called `boost::regex`, which is presented in Chapter 8.
Example 5.12 writes `C++` to standard output.

Chapter 6

Boost.LexicalCast

Boost.LexicalCast provides a cast operator, `boost::lexical_cast`, that can convert numbers from strings to numeric types like int or double and vice versa. `boost::lexical_cast` is an alternative to functions like `std::stoi()`, `std::stod()`, and `std::to_string()`, which were added to the standard library in C++11.

Example 6.1 Using `boost::lexical_cast`

```
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = boost::lexical_cast<std::string>(123);
  std::cout << s << '\n';
  double d = boost::lexical_cast<double>(s);
  std::cout << d << '\n';
}
```

The cast operator `boost::lexical_cast` can convert numbers of different types. Example 6.1 first converts the integer 123 to a string, then converts the string to a floating point number. To use `boost::lexical_cast`, include the header file `boost/lexical_cast.hpp`.

`boost::lexical_cast` uses streams internally to perform the conversion. Therefore, only types with overloaded `operator<<` and `operator>>` can be converted. However, `boost::lexical_cast` can be optimized for certain types to implement a more efficient conversion.

Example 6.2 `boost::bad_lexical_cast` in case of an error

```
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

int main()
{
  try
  {
    int i = boost::lexical_cast<int>("abc");
    std::cout << i << '\n';
  }
  catch (const boost::bad_lexical_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

If a conversion fails, an exception of type `boost::bad_lexical_cast`, which is derived from `std::bad_cast`, is thrown. Example 6.2 throws an exception because the string “abc” cannot be converted to a number of type int.

Chapter 7

Boost.Format

Boost.Format offers a replacement for the function `std::printf()`. `std::printf()` originates from the C standard and allows formatted data output. However, it is neither type safe nor extensible. Boost.Format provides a type-safe and extensible alternative.

Boost.Format provides a class called `boost::format`, which is defined in `boost/format.hpp`. Similar to `std::printf()`, a string containing special characters to control formatting is passed to the constructor of `boost::format`. The data that replaces these special characters in the output is linked via the operator `operator%`.

Example 7.1 Formatted output with `boost::format`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%1%.%2%.%3%"} % 12 % 5 % 2014 << '\n';
}
```

The Boost.Format format string uses numbers placed between two percent signs as placeholders for the actual data, which will be linked in using `operator%`. Example 7.1 creates a date string in the form `12.5.2014` using the numbers 12, 5, and 2014 as the data. To make the month appear in front of the day, which is common in the United States, the placeholders can be swapped. Example 7.2 makes this change, displaying `5/12/2014`

Example 7.2 Numbered placeholders with `boost::format`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%2%/%1%/%3%"} % 12 % 5 % 2014 << '\n';
}
```

To format data with manipulators, Boost.Format provides a function called `boost::io::group()`.

Example 7.3 Using manipulators with `boost::io::group()`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%1% %2% %1%"} %
    boost::io::group(std::showpos, 1) % 2 << '\n';
}
```

Example 7.3 uses the manipulator `std::showpos()` on the value that will be associated with “%1%”. Therefore, this example will display `+1 2 +1` as output. Because the manipulator `std::showpos()` has been linked to the first data value using `boost::io::group()`, the plus sign is automatically added whenever this value is displayed. In this case, the format placeholder “%1%” is used twice.

If the plus sign should only be shown for the first output of 1, the format placeholder needs to be customized.

Example 7.4 Placeholders with special characters

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%|1$+| %2% %1%"} % 1 % 2 << '\n';
}
```

Example 7.4 does this. In this example, the first instance of the placeholder “%1%” is replaced with “%|1$+|”. Customization of a format does not just add two additional pipe signs. The reference to the data is also placed between the pipe signs and uses “1$” instead of “1%”. This is required to modify the output to be `+1 2 1`. You can find details about the format specifications in the Boost documentation.

Placeholder references to data must be specified either for all placeholders or for none. Example 7.5 only provides references for one of three placeholders, which generates an error at run time.

Example 7.5 `boost::io::format_error` in case of an error

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::format{"%|+| %2% %1%"} % 1 % 2 << '\n';
  }
  catch (boost::io::format_error &ex)
  {
    std::cout << ex.what() << '\n';
  }
}
```

Example 7.5 throws an exception of type `boost::io::format_error`. Strictly speaking, Boost.Format throws `boost::io::bad_format_string`. However, because the different exception classes are all derived from `boost::io::format_error`, it is usually easier to catch exceptions of this type.

Example 7.6 shows how to write the program without using references in the format string.

Example 7.6 Placeholders without numbers

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%|+| %|| %||"} % 1 % 2 % 1 << '\n';
}
```

The pipe signs for the second and third placeholder can safely be omitted in this case because they do not specify any format. The resulting syntax then closely resembles `std::printf()` (see Example 7.7).

Example 7.7 `boost::format` with the syntax used from `std::printf()`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%+d %d %d"} % 1 % 2 % 1 << '\n';
}
```

While the format may look like that used by `std::printf()`, Boost.Format provides the advantage of type safety. The letter “d” within the format string does not indicate the output of a number. Instead, it applies the manipulator `std::dec()` to the internal stream object used by `boost::format`. This makes it possible to specify format strings that would make no sense for `std::printf()` and would result in a crash.

Example 7.8 `boost::format` with seemingly invalid placeholders

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%+s %s %s"} % 1 % 2 % 1 << '\n';
}
```

`std::printf()` allows the letter "s" only for strings of type const char*. With `std::printf()`, the combination of "%s" and a numeric value would fail. However, Example 7.8 works perfectly. Boost.Format does not require a string. Instead, it applies the appropriate manipulators to configure the internal stream.

Boost.Format is both type safe and extensible. Objects of any type can be used with Boost.Format as long as the operator `operator<<` is overloaded for `std::ostream`.

Example 7.9 `boost::format` with user-defined type

```
#include <boost/format.hpp>
#include <string>
#include <iostream>

struct animal
{
  std::string name;
  int legs;
};

std::ostream &operator<<(std::ostream &os, const animal &a)
{
  return os << a.name << ',' << a.legs;
}

int main()
{
  animal a{"cat", 4};
  std::cout << boost::format{"%1%"} % a << '\n';
}
```

Example 7.9 uses `boost::format` to write an object of the user-defined type `animal` to standard output. This is possible because the stream operator is overloaded for `animal`.

Chapter 8

Boost.Regex

Boost.Regex allows you to use *regular expressions* in C++. As the library is part of the standard library since C++11, you don't depend on Boost.Regex if your development environment supports C++11. You can use identically named classes and functions in the namespace `std` if you include the header file `regex`.
The two most important classes in Boost.Regex are `boost::regex` and `boost::smatch`, both defined in `boost/regex.hpp`. The former defines a regular expression, and the latter saves the search results.
Boost.Regex provides three different functions to search for regular expressions.

Example 8.1 Comparing strings with `boost::regex_match()`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"\\w+\\s\\w+"};
  std::cout << std::boolalpha << boost::regex_match(s, expr) << '\n';
}
```

`boost::regex_match()` (see Example 8.1) compares a string with a regular expression. It will return `true` only if the expression matches the complete string.
`boost::regex_search()` searches a string for a regular expression.

Example 8.2 Searching strings with `boost::regex_search()`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  boost::smatch what;
  if (boost::regex_search(s, what, expr))
  {
    std::cout << what[0] << '\n';
    std::cout << what[1] << "_" << what[2] << '\n';
  }
}
```

`boost::regex_search()` expects a reference to an object of type `boost::smatch` as an additional parameter, which is used to store the results. `boost::regex_search()` only searches for groups. That's why Example 8.2 returns two strings based on the two groups found in the regular expression.
The result storage class `boost::smatch` is a container holding elements of type `boost::sub_match`, which can be accessed through an interface similar to the one of `std::vector`. For example, elements can be accessed via `operator[]`.

The class `boost::sub_match` stores iterators to the specific positions in a string corresponding to the groups of a regular expression. Because `boost::sub_match` is derived from `std::pair`, the iterators that reference a particular substring can be accessed with **`first`** and **`second`**. However, to write a substring to the standard output stream, you don't have to access these iterators (see Example 8.2). Using the overloaded operator `operator<<`, the substring can be written directly to standard output.
Please note that because iterators are used to point to matched strings, `boost::sub_match` does not copy them. This implies that results are accessible only as long as the corresponding string, which is referenced by the iterators, exists.
Furthermore, please note that the first element of the container `boost::smatch` stores iterators referencing the string that matches the entire regular expression. The first substring that matches the first group is accessible at index 1.
The third function offered by Boost.Regex is `boost::regex_replace()` (see Example 8.3).

Example 8.3 Replacing characters in strings with `boost::regex_replace()`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = " Boost Libraries ";
  boost::regex expr{"\\s"};
  std::string fmt{"_"};
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

In addition to the search string and the regular expression, `boost::regex_replace()` needs a format that defines how substrings that match individual groups of the regular expression should be replaced. In case the regular expression does not contain any groups, the corresponding substrings are replaced one to one using the given format. Thus, Example 8.3 will output `_Boost_Libraries_`.
`boost::regex_replace()` always searches through the entire string for the regular expression. Thus, the program actually replaces all three spaces with underscores.

Example 8.4 Format with references to groups in regular expressions

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  std::string fmt{"\\2 \\1"};
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

The format can access substrings returned by groups of the regular expression. Example 8.4 uses this technique to swap the first and last word, displaying `Libraries Boost` as a result.
There are different standards for regular expressions and formats. Each of the three functions takes an additional parameter that allows you to select a specific standard. You can also specify whether or not special characters should be interpreted in a specific format or whether the format should replace the complete string that matches the regular expression.

Example 8.5 Flags for formats

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w+)\\s(\\w+)"};
  std::string fmt{"\\2 \\1"};
  std::cout << boost::regex_replace(s, expr, fmt,
    boost::regex_constants::format_literal) << '\n';
}
```

Example 8.5 passes the flag `boost::regex_constants::format_literal` as the fourth parameter to `boost::regex_replace()` to suppress handling of special characters in the format. Because the complete string that matches the regular expression is replaced with the format, the output of Example 8.5 is `\2 \1`.

Example 8.6 Iterating over strings with `boost::regex_token_iterator`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"\\w+"};
  boost::regex_token_iterator<std::string::iterator> it{s.begin(), s.end(),
    expr};
  boost::regex_token_iterator<std::string::iterator> end;
  while (it != end)
    std::cout << *it++ << '\n';
}
```

With `boost::regex_token_iterator`, Boost.Regex provides a class to iterate over a string with a regular expression. In Example 8.6 the iteration returns the two words in **`s`**. **`it`** is initialized with iterators to **`s`** and the regular expression "\w+". The default constructor creates an end iterator.
Example 8.6 displays `Boost` and `Libraries`.

Example 8.7 Accessing groups with `boost::regex_token_iterator`

```
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost Libraries";
  boost::regex expr{"(\\w)\\w+"};
  boost::regex_token_iterator<std::string::iterator> it{s.begin(), s.end(),
    expr, 1};
  boost::regex_token_iterator<std::string::iterator> end;
  while (it != end)
    std::cout << *it++ << '\n';
}
```

You can pass a number as an additional parameter to the constructor of `boost::regex_token_iterator`. If 1 is passed, as in Example 8.7, the iterator returns the first group in the regular expression. Because the regular expression "(\w)\w+" is used, Example 8.7 writes the initials `B` and `L` to standard output.
If -1 is passed to `boost::regex_token_iterator`, the regular expression is the delimiter. An iterator initialized with -1 returns substrings that do not match the regular expression.

Example 8.8 Linking a locale to a regular expression

```
#include <boost/regex.hpp>
#include <locale>
#include <string>
#include <iostream>

int main()
{
  std::string s = "Boost k\xfct\xfcphaneleri";
  boost::basic_regex<char, boost::cpp_regex_traits<char>> expr;
  expr.imbue(std::locale{"Turkish"});
  expr = "\\w+\\s\\w+";
  std::cout << std::boolalpha << boost::regex_match(s, expr) << '\n';
}
```

Example 8.8 links a locale with `imbue()` to **`expr`**. This is done to apply the regular expression to the string “Boost kütüphaneleri,” which is the Turkish translation of “Boost Libraries.” If umlauts should be parsed as valid letters, the locale must be set – otherwise `boost::regex_match()` returns `false`.
To use a locale of type `std::locale`, **`expr`** must be based on a class instantiated with the type `boost::cpp_regex_traits`. That’s why Example 8.8 doesn’t use `boost::regex` but instead uses boost::basic_regex<char, boost::cpp_regex_traits<char>>. With the second template parameter of `boost::basic_regex`, the parameter for `imbue()` can be defined indirectly. Only with `boost::cpp_regex_traits` can a locale of type `std::locale` be passed to `imbue()`.

Note

If you want to run the example on a POSIX operating system, replace “Turkish” with “tr_TR”. Also make sure the locale for Turkish is installed.

Note that `boost::regex` is defined with a platform-dependent second template parameter. On Windows this parameter is `boost::w32_regex_traits`, which allows an LCID to be passed to `imbue()`. An LCID is a number that, on Windows, identifies a certain language and culture. If you want to write platform-independent code, you must use `boost::cpp_regex_traits` explicitly, as in Example 8.8. Alternatively, you can define the macro `BOOST_REGEX_USE_CPP_LOCALE`.

Chapter 9

Boost.Xpressive

Like Boost.Regex, Boost.Xpressive provides functions to search strings using *regular expressions*. However, Boost.Xpressive makes it possible to write down regular expressions as C++ code rather than strings. That makes it possible to check at compile time whether a regular expression is valid or not.
Only Boost.Regex was incorporated into C++11. The standard library doesn't provide any support for writing regular expressions as C++ code.
`boost/xpressive/xpressive.hpp` provides access to most library functions in Boost.Xpressive. For some functions, additional header files must be included. All definitions of the library can be found in the namespace `boost::xpressive`.

Example 9.1 Comparing strings with `boost::xpressive::regex_match`

```
#include <boost/xpressive/xpressive.hpp>
#include <string>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  sregex expr = sregex::compile("\\w+\\s\\w+");
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

Boost.Xpressive basically provides the same functions as Boost.Regex, except they are defined in the namespace of Boost.Xpressive. `boost::xpressive::regex_match()` compares strings, `boost::xpressive::regex_search()` searches in strings, and `boost::xpressive::regex_replace()` replaces characters in strings. You can see this in Example 9.1, which uses the function `boost::xpressive::regex_match()`, and which looks similar to Example 8.1.
However, there is a fundamental difference between Boost.Xpressive and Boost.Regex. The type of the regular expression in Boost.Xpressive depends on the type of the string being searched. Because **`s`** is based on `std::string` in Example 9.1, the type of the regular expression must be `boost::xpressive::sregex`. Compare this with Example 9.2, where the regular expression is applied to a string of type const char*.

Example 9.2 `boost::xpressive::cregex` with strings of type const char*

```
#include <boost/xpressive/xpressive.hpp>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  const char *c = "Boost Libraries";
  cregex expr = cregex::compile("\\w+\\s\\w+");
  std::cout << std::boolalpha << regex_match(c, expr) << '\n';
}
```

For strings of type const char*, use the class `boost::xpressive::cregex`. If you use other string types, such as `std::wstring` or const wchar_t*, use `boost::xpressive::wsregex` or `boost::xpressive::wcregex`.
You must call the static member function `compile()` for regular expressions written as strings. The member function must be called on the type used for the regular expression.
Boost.Xpressive supports direct initialization of regular expressions that are written as C++ code. The regular expression has to be expressed in the notation supported by Boost.Xpressive (see Example 9.3).

Example 9.3 A regular expression with C++ code

```
#include <boost/xpressive/xpressive.hpp>
#include <string>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  sregex expr = +_w >> _s >> +_w;
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

The regular expression from Example 9.2, which was written as the string “\w+\s\w+”, is now expressed in Example 9.3 as `+_w >> _s >> +_w`. It is exactly the same regular expression. Both examples search for at least one alphanumeric character followed by one space followed by at least one alphanumeric character.
Boost.Xpressive makes it possible to write regular expressions with C++ code. The library provides objects for character groups. For example, the object **`_w`** is similar to “\w”. **`_s`** has the same meaning as “\s”.
While “\w” and “\s” can be written one after another in a string, objects like **`_w`** and **`_s`** must be concatenated with an operator. Otherwise, the result wouldn’t be valid C++ code. Boost.Xpressive provides the operator `operator>>`, which is used in Example 9.3.
To express that at least one alphanumeric character should be found, **`_w`** is prefixed with a plus sign. While the syntax of regular expressions expects that quantifiers are put behind character groups – like with “\w+” – the plus sign must be put in front of **`_w`**. The plus sign is an unary operator, which in C++ must be put in front of an object.
Boost.Xpressive emulates the rules of regular expressions as much as they can be emulated in C++. However, there are limits. For example, the question mark is a meta character in regular expressions to express that a preceding item is optional. Since the question mark isn’t a valid operator in C++, Boost.Xpressive replaces it with the exclamation mark. A notation like “\w?” becomes `!_w` with Boost.Xpressive because the exclamation mark must be prefixed.
Boost.Xpressive supports actions that can be linked to expressions – something Boost.Regex doesn’t support.

Example 9.4 Linking actions to expressions

```
#include <boost/xpressive/xpressive.hpp>
#include <boost/xpressive/regex_actions.hpp>
#include <string>
#include <iterator>
#include <iostream>

using namespace boost::xpressive;

int main()
{
  std::string s = "Boost Libraries";
  std::ostream_iterator<std::string> it{std::cout, "\n"};
  sregex expr = (+_w)[boost::xpressive::ref(it) = _] >> _s >> +_w;
  std::cout << std::boolalpha << regex_match(s, expr) << '\n';
}
```

Example 9.4 returns `true` for `boost::xpressive::regex_match()` and writes `Boost` to standard output.
You can link actions to expressions. An action is executed when the respective expression is found. In Example 9.4, the expression `+_w` is linked to the action `boost::xpressive::ref(it) =_`. The action is a lambda

function. The object _ refers to characters found by the expression – in this case the first word in **`s`**. The respective characters are assigned to the iterator **`it`**. Because **`it`** is an iterator of type `std::ostream_iterator`, which has been initialized with **`std::cout`**, `Boost` is written to standard output.
Please note that you must use the function `boost::xpressive::ref()` to wrap the iterator **`it`**. Only then it is possible to assign _ to the iterator. _ is an object provided by Boost.Xpressive in the namespace `boost::xpressive`, which normally couldn't be assigned to an iterator of type `std::ostream_iterator`. Because the assignment happens only when the string "Boost" has been found with `+_w`, `boost::xpressive::ref()` turns the assignment into a *lazy* operation. Although the code in square brackets attached to `+_w` is, according to C++ rules, immediately executed, the assignment to the iterator **`it`** can only occur when the regular expression is used. Thus, `boost::xpressive::ref(it) =_` isn't executed immediately.
Example 9.4 includes the header file `boost/xpressive/regex_actions.hpp`. This is required because actions aren't available through `boost/xpressive/xpressive.hpp`.
Like Boost.Regex, Boost.Xpressive supports iterators to split a string with regular expressions. The classes `boost::xpressive::regex_token_iterator` and `boost::xpressive::regex_iterator` do this. It is also possible to link a locale to a regular expression to use a locale other than the global one.

Chapter 10

Boost.Tokenizer

The library Boost.Tokenizer allows you to iterate over partial expressions in a string by interpreting certain characters as separators.

Example 10.1 Iterating over partial expressions in a string with `boost::tokenizer`

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  tokenizer tok{s};
  for (tokenizer::iterator it = tok.begin(); it != tok.end(); ++it)
    std::cout << *it << '\n';
}
```

Boost.Tokenizer defines a class template called `boost::tokenizer` in `boost/tokenizer.hpp`. It expects as a template parameter a class that identifies coherent expressions. Example 10.1 uses the class `boost::char_separator`, which interprets spaces and punctuation marks as separators.
A tokenizer must be initialized with a string of type `std::string`. Using the member functions `begin()` and `end()`, the tokenizer can be accessed like a container. Partial expressions of the string used to initialize the tokenizer are available via iterators. How partial expressions are evaluated depends on the kind of class passed as the template parameter.
Because `boost::char_separator` interprets spaces and punctuation marks as separators by default, Example 10.1 displays `Boost`, `C`, `+`, `+`, and `Libraries`. `boost::char_separator` uses `std::isspace()` and `std::ispunct()` to identify separator characters. Boost.Tokenizer distinguishes between separators that should be displayed and separators that should be suppressed. By default, spaces are suppressed and punctuation marks are displayed.

Example 10.2 Initializing `boost::char_separator` to adapt the iteration

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  boost::char_separator<char> sep{" "};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

To keep punctuation marks from being interpreted as separators, initialize the `boost::char_separator` object before passing it to the tokenizer.

The constructor of `boost::char_separator` accepts a total of three parameters, but only the first one is required. The first parameter describes the individual separators that are suppressed. Example 10.2, like Example 10.1, treats spaces as separators.
The second parameter specifies the separators that should be displayed. If this parameter is omitted, no separators are displayed, and the program will now display `Boost`, `C++` and `Libraries`.

Example 10.3 Simulating the default behavior with `boost::char_separator`

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  boost::char_separator<char> sep{" ", "+"};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

If a plus sign is passed as the second parameter, Example 10.3 behaves like Example 10.1.
The third parameter determines whether or not empty partial expressions are displayed. If two separators are found back-to-back, the corresponding partial expression is empty. By default, these empty expressions are not displayed. Using the third parameter, the default behavior can be changed.

Example 10.4 Initializing `boost::char_separator` to display empty partial expressions

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<char>> tokenizer;
  std::string s = "Boost C++ Libraries";
  boost::char_separator<char> sep{" ", "+", boost::keep_empty_tokens};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

Example 10.4 displays two additional empty partial expressions. The first one is found between the two plus signs, while the second one is found between the second plus sign and the following space.

Example 10.5 Boost.Tokenizer with wide strings

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::char_separator<wchar_t>,
    std::wstring::const_iterator, std::wstring> tokenizer;
  std::wstring s = L"Boost C++ Libraries";
  boost::char_separator<wchar_t> sep{L" "};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::wcout << t << '\n';
}
```

Example 10.5 iterates over a string of type `std::wstring`. In order to support this string type, the tokenizer must be initialized with additional template parameters. The class `boost::char_separator` must also be initialized with wchar_t.

Besides `boost::char_separator`, Boost.Tokenizer provides two additional classes to identify partial expressions.

Example 10.6 Parsing CSV files with `boost::escaped_list_separator`

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::escaped_list_separator<char>> tokenizer;
  std::string s = "Boost,\"C++ Libraries\"";
  tokenizer tok{s};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

`boost::escaped_list_separator` is used to read multiple values separated by commas. This format is commonly known as CSV (Comma Separated Values). `boost::escaped_list_separator` also handles double quotes and escape sequences. Therefore, the output of Example 10.6 is `Boost` and `C++ Libraries`.
The second class provided is `boost::offset_separator`, which must be instantiated. The corresponding object must be passed to the constructor of `boost::tokenizer` as a second parameter.

Example 10.7 Iterating over partial expressions with `boost::offset_separator`

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tokenizer<boost::offset_separator> tokenizer;
  std::string s = "Boost C++ Libraries";
  int offsets[] = {5, 5, 9};
  boost::offset_separator sep{offsets, offsets + 3};
  tokenizer tok{s, sep};
  for (const auto &t : tok)
    std::cout << t << '\n';
}
```

`boost::offset_separator` specifies the locations within the string where individual partial expressions end. Example 10.7 specifies that the first partial expression ends after 5 characters, the second ends after an additional 5 characters, and the third ends after the following 9 characters. The output will be `Boost`, ` C++` and `Libraries`.

Chapter 11

Boost.Spirit

This chapter introduces the library Boost.Spirit. Boost.Spirit is used to develop parsers for text formats. For example, you can use Boost.Spirit to develop a parser to load configuration files. Boost.Spirit can also be used for binary formats, although its usefulness in this respect is limited.
Boost.Spirit simplifies the development of parsers because formats are described with rules. Rules define what a format looks like – Boost.Spirit does the rest. You can compare Boost.Spirit to regular expressions, in the sense that it lets you handle complex processes – pattern searching in the case of regular expressions and parsing for Boost.Spirit – without having to write code to implement that process.
Boost.Spirit expects rules to be described using Parsing Expression Grammar (PEG). PEG is related to Extended Backus-Naur-Form (EBNF). Even if you are not familiar with these languages, the examples in this chapter should be sufficient to get you started.
There are two versions of Boost.Spirit. The first version is known as Spirit.Classic. This version should not be used anymore. The current version is 2.5.2. This is the version introduced in this chapter.
Since version 2.x, Boost.Spirit can be used to generate generators as well as parsers. While parsers read text formats, generators write them. The component of Boost.Spirit that is used to develop parsers is called Spirit.Qi. Spirit.Karma is the component used to develop generators. Namespaces are partitioned accordingly: classes and functions to develop parsers can be found in `boost::spirit::qi` and classes and functions to develop generators can be found in `boost::spirit::karma`.
Besides Spirit.Qi and Spirit.Karma, the library contains a component called Spirit.Lex, which can be used to develop lexers.
This chapter focuses on developing parsers. The examples mainly use classes and functions from `boost::spirit` and `boost::spirit::qi`. For these classes and functions, it is sufficient to include the header file `boost/spirit/include/qi.hpp`.
If you don't want to include a master header file like `boost/spirit/include/qi.hpp`, you can include header files from `boost/spirit/include/` individually. It is important to include header files from this directory only. `boost/spirit/include/` is the interface to the user. Header files in other directories can change in new library versions.

11.1 API

Boost.Spirit provides `boost::spirit::qi::parse()` and `boost::spirit::qi::phrase_parse()` to parse a format.

Example 11.1 Using `boost::spirit::qi::parse()`

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::parse(it, s.end(), ascii::digit);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.1 introduces `boost::spirit::qi::parse()`. This function expects two iterators of the string being parsed and a parser. The example uses the parser **`boost::spirit::ascii::digit`**, which is provided by Boost.Spirit. This is one of several character classification parsers. These parsers test whether characters belong to a certain class. **`boost::spirit::ascii::digit`** tests whether a character is a digit between 0 and 9.
The example passes iterators of a string which is read from **`std::cin`**. Note that the begin iterator isn't passed directly to `boost::spirit::qi::parse()`. It is stored in the variable **`it`**, which is then passed to `boost::spirit::qi::parse()`. This is done because `boost::spirit::qi::parse()` may modify the iterator.
If you type a digit and then **Enter**, the example displays `true`. If you type two digits and then **Enter**, the output will be `true` followed by the second digit. If you enter a letter and then **Enter**, the output will be `false` followed by the letter.
The parser **`boost::spirit::ascii::digit`**, as used in Example 11.1, tests exactly one character to see whether it's a digit. If the first character is a digit, `boost::spirit::qi::parse()` returns `true` – otherwise, it returns `false`. The return value of `boost::spirit::qi::parse()` indicates whether the parser succeeded.
`boost::spirit::qi::parse()` also returns `true` if you enter multiple digits. Because the parser **`boost::spirit::ascii::digit`** only tests the first character, it will succeed on such a string. All digits after the first will be ignored.
To let you determine how much of the string could be parsed successfully, `boost::spirit::qi::parse()` changes the iterator **`it`**. After a call to `boost::spirit::qi::parse()`, **`it`** refers to the character after the last one parsed successfully. If you enter multiple digits, **`it`** refers to the second digit. If you enter exactly one digit, **`it`** equals the end iterator of **`s`**. If you enter a letter, **`it`** refers to that letter.
`boost::spirit::qi::parse()` does not ignore spaces. If you run Example 11.1 and enter a space, `false` is displayed. `boost::spirit::qi::parse()` tests the first entered character, even if that character is a space. If you want to ignore spaces, use `boost::spirit::qi::phrase_parse()` instead of `boost::spirit::qi::parse()`.

Example 11.2 Using `boost::spirit::qi::phrase_parse()`

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit, ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

`boost::spirit::qi::phrase_parse()` works like `boost::spirit::qi::parse()` but expects another parameter called skipper. The skipper is a parser for characters that should be ignored. Example 11.2 uses **`boost::spirit::ascii::space`**, a character classification parser to detect spaces, as the skipper.
The skipper **`boost::spirit::ascii::space`** discards spaces as delimiters. If you start the example and enter a space followed by a digit, it displays `true`. Unlike the previous example, the parser **`boost::spirit::ascii::digit`** is not applied to the space, but to the first character that isn't a space.
Note that this example ignores any number of spaces. Thus, `boost::spirit::qi::phrase_parse()` returns `true` if you enter multiple spaces followed by a digit.
Like `boost::spirit::qi::parse()`, `boost::spirit::qi::phrase_parse()` modifies the iterator passed as the first parameter. That way, you know how far into the string the parser was able to work successfully. Example 11.2 skips spaces that occur after successfully parsed characters. If you enter a digit followed by a space followed by a letter, the iterator will refer to the letter, not the space in front of it. If you want the iterator to refer

to the space, pass `boost::spirit::qi::skip_flag::dont_postskip` as another parameter to `boost::spirit::qi::phrase_parse()`.

Example 11.3 `phrase_parse()` with `boost::spirit::qi::skip_flag::dont_postskip`

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit, ascii::space,
    qi::skip_flag::dont_postskip);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.3 passes `boost::spirit::qi::skip_flag::dont_postskip` to `boost::spirit::qi::phrase_parse()` to tell the parser not to skip spaces that occur after a successfully parsed digit, but before the first unsuccessfully parsed character. If you enter a digit followed by a space followed by a letter, **`it`** refers to the space after the call to `boost::spirit::qi::phrase_parse()`.
The flag `boost::spirit::qi::skip_flag::postskip` is the default value, which is used if neither `boost::spirit::qi::skip_flag::dont_postskip` nor `boost::spirit::qi::skip_flag::postskip` is specified.

Example 11.4 `boost::spirit::qi::phrase_parse()` with wide strings

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::wstring s;
  std::getline(std::wcin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit, ascii::space,
    qi::skip_flag::dont_postskip);
  std::wcout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::wcout << std::wstring{it, s.end()} << '\n';
}
```

`boost::spirit::qi::parse()` and `boost::spirit::qi::phrase_parse()` accept iterators to a wide string. Example 11.4 works like the previous example, except that wide strings are used.
Boost.Spirit also supports the string types `std::u16string` and `std::u32string` from the C++11 standard library.

11.2 Parsers

This section explains how you define parsers. You usually access existing parsers from Boost.Spirit – for example, **`boost::spirit::ascii::digit`** or **`boost::spirit::ascii::space`**. By combining parsers, you can parse more complex formats. The process is similar to defining regular expressions, which are also built from basic building blocks.

Example 11.5 tests whether two digits are entered. `boost::spirit::qi::phrase_parse()` only returns `true` if the two digits are consecutive. Spaces are ignored.
As with the previous examples, **`boost::spirit::ascii::digit`** is used to recognize digits. Because **`boost::spirit::ascii::digit`** tests exactly one character, the parser is used twice to test the input for two digits. To use **`boost::spirit::ascii::digit`** twice in a row, an operator has to be used. Boost.Spirit overloads `operator>>` for parsers. With `ascii::digit >> ascii::digit` a parser is created that tests whether a string contains two digits.
If you run the example and enter two digits, `true` is displayed. If you enter only one digit, the example displays `false`.

Example 11.5 A parser for two consecutive digits

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), ascii::digit >> ascii::digit,
    ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Please note that the example also displays `true` if you enter a space between two digits. Wherever the operator `operator>>` is used in a parser, characters are allowed which are ignored by a skipper. Because Example 11.5 uses **`boost::spirit::ascii::space`** as the skipper, you may enter as many spaces as you like between the two digits.
If you want the parser to accept two digits only if they follow each other with no space in between, use `boost::spirit::qi::parse()` or the directive **`boost::spirit::qi::lexeme`**.

Example 11.6 Parsing character by character with **`boost::spirit::qi::lexeme`**

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(),
    qi::lexeme[ascii::digit >> ascii::digit], ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.6 uses the parser `qi::lexeme[ascii::digit >> ascii::digit]`. Now, `boost::spirit::qi::phrase_parse()` only returns `true` if the digits have no spaces between them.
`boost::spirit::qi::lexeme` is one of several directives that can change the behavior of parsers. You use **`boost::spirit::qi::lexeme`** if you want to disallow characters that would be ignored by a skipper when `operator>>` is used.

Example 11.7 Boost.Spirit rules similar to regular expressions

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

```
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), +ascii::digit, ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.7 defines a parser with `+ascii::digit`, which expects at least one digit. This syntax, in particular the plus sign (+), is similar to that used in regular expressions. The plus sign identifies a character or character group which is expected to occur in a string at least once. If you start the example and enter at least one digit, `true` is displayed. It doesn't matter whether digits are delimited by spaces. If the parser should accept only digits without spaces, use **`boost::spirit::qi::lexeme`** again.

Example 11.8 Numeric parsers

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(), qi::int_, ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.8 expects an integer. **`boost::spirit::qi::int_`** is a numeric parser that can recognize positive and negative integers. Unlike **`boost::spirit::ascii::digit`**, **`boost::spirit::qi::int_`** can recognize several characters, such as +1 or -23, as integers.

Boost.Spirit provides additional logical parsers. **`boost::spirit::qi::float_`**, **`boost::spirit::qi::double_`**, and **`boost::spirit::qi::bool_`** are numeric parsers that can read floating point numbers and boolean values. With **`boost::spirit::qi::eol`**, you can test for an end-of-line character. **`boost::spirit::qi::byte_`** and **`boost::spirit::qi::word`** can be used to read one or two bytes. **`boost::spirit::qi::word`** and other binary parsers recognize the endianness of a platform and parse accordingly. If you want to parse based on a specific endianness, regardless of the platform, you can use parsers like **`boost::spirit::qi::little_word`** and **`boost::spirit::qi::big_word`**.

11.3 Actions

So far, the examples in this chapter only detect two things: whether the parser was successful and where the parse ended. However, parsers normally process data in some way, as you will see in the next examples.

Example 11.9 Linking actions with parsers

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>
```

```
using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  bool match = qi::phrase_parse(it, s.end(),
    qi::int_[([](int i){ std::cout << i << '\n'; })], ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.9 uses **`boost::spirit::qi::int_`** to parse an integer, then writes that integer to standard output. That's why an *action* has been linked with **`boost::spirit::qi::int_`**. Actions are functions or function objects that are called when a parser is applied. Linking is done with the operator `operator[]`, which is overloaded by **`boost::spirit::qi::int_`** and other parsers. Example 11.9 uses a lambda function as an action that expects a sole parameter of type int and writes it to standard output.
If you start Example 11.9 and enter a number, the number is displayed.
The type of the parameter passed to an action depends on the parser. For example, **`boost::spirit::qi::int_`** forwards an int value, while **`boost::spirit::qi::float_`** passes a float value.

Example 11.10 Boost.Spirit with Boost.Phoenix

```
#define BOOST_SPIRIT_USE_PHOENIX_V3
#include <boost/spirit/include/qi.hpp>
#include <boost/phoenix/phoenix.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;
using boost::phoenix::ref;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  int i;
  bool match = qi::phrase_parse(it, s.end(), qi::int_[ref(i) = qi::_1],
    ascii::space);
  std::cout << std::boolalpha << match << '\n';
  if (match)
    std::cout << i << '\n';
  if (it != s.end())
    std::cout << std::string{it, s.end()} << '\n';
}
```

Example 11.10 uses Boost.Phoenix to store the int value parsed with **`boost::spirit::qi::int_`** in **`i`**. If `boost::spirit::qi::phrase_parse()` returns `true`, **`i`** is written to standard output.
This example defines the macro `BOOST_SPIRIT_USE_PHOENIX_V3` before including the header files from Boost.Spirit. This macro selects the third and current version of Boost.Phoenix. This is important because Boost.Phoenix was forked from Boost.Spirit, and Boost.Spirit contains the second version of Boost.Phoenix. If `BOOST_SPIRIT_USE_PHOENIX_V3` isn't defined, the second version of Boost.Phoenix will be included through the Boost.Spirit header files and the third version through `boost/phoenix/phoenix.hpp`. The different versions will lead to a compiler error.
Please note how the lambda function is defined in detail. `boost::phoenix::ref()` creates a reference to the variable **`i`**. However, the placeholder **`_1`** isn't from Boost.Phoenix, it's from Boost.Spirit. This is important because **`boost::spirit::qi::_1`** provides access to the parsed value with the normally expected type – int in Example 11.10. If the lambda function used **`boost::phoenix::placeholders::arg1`**, the compiler would report an error because **`boost::phoenix::placeholders::arg1`** wouldn't represent an int; it would be based on another type from Boost.Spirit, and the int value would need to be extracted.

The Boost.Spirit documentation contains an overview on tools that support a Boost.Phoenix integration.

11.4 Attributes

Actions are one option to process parsed values. Another option is to pass objects to `boost::spirit::qi::parse()` or `boost::spirit::qi::phrase_parse()` that will be used to store parsed values. These objects are called attributes. Their types must match the parsers' types.
You have already used attributes in the previous section. Parameters passed to actions are attributes, too. Every parser has an attribute. For example, the parser **`boost::spirit::qi::int_`** has an attribute of type int. In the following examples, attributes aren't passed as parameters to functions. Instead, parsed values are stored in attributes and can be processed after `boost::spirit::qi::parse()` or `boost::spirit::qi::phrase_parse()` return.

Example 11.11 Storing an int value in an attribute

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  int i;
  if (qi::phrase_parse(it, s.end(), qi::int_, ascii::space, i))
    std::cout << i << '\n';
}
```

Example 11.11 uses the parser **`boost::spirit::qi::int_`**. The parsed int value is stored in the variable **`i`**. **`i`** is passed as another parameter to `boost::spirit::qi::phrase_parse()` and, thus, becomes an attribute of the parser **`boost::spirit::qi::int_`**.
If you start Example 11.11 and enter a digit, the digit will be written to the standard output stream.
Example 11.12 uses a parser that is defined with `qi::int_ % ','`. The parser accepts any number of integers delimited by commas. As usual spaces are ignored.
Because the parser can return multiple int values, the attribute's type must support storing multiple int values. The example passes a vector. If you start the example and enter multiple integers delimited by commas, the integers are written to the standard output stream delimited by semicolons.

Example 11.12 Storing several int values in an attribute

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  std::vector<int> v;
  if (qi::phrase_parse(it, s.end(), qi::int_ % ',', ascii::space, v))
  {
    std::ostream_iterator<int> out{std::cout, ";"};
    std::copy(v.begin(), v.end(), out);
  }
}
```

Instead of a vector, you can also pass containers of other types, such as `std::list`.
The Boost.Spirit documentation describes which attribute types must be used with which operators.

11.5 Rules

In Boost.Spirit, parsers consist of rules. As rules are typically based on parsers provided by Boost.Spirit, there is no clear distinction. For example, **`boost::spirit::ascii::digit`** can be both a parser and a rule. Typically, rules refer to more complicated expressions like `qi::int_ % ','`.
In all of the examples thus far, rules were passed to `boost::spirit::qi::parse()` or `boost::spirit::qi::phrase_parse()` directly. With `boost::spirit::qi::rule`, Boost.Spirit provides a class to define rule variables. For example, `boost::spirit::qi::rule` is required if a rule should be stored in a member variable of a class.

Example 11.13 Defining rules with `boost::spirit::qi::rule`

```
#include <boost/spirit/include/qi.hpp>
#include <string>
#include <vector>
#include <iterator>
#include <algorithm>
#include <iostream>

using namespace boost::spirit;

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  qi::rule<std::string::iterator, std::vector<int>(),
    ascii::space_type> values = qi::int_ % ',';
  std::vector<int> v;
  if (qi::phrase_parse(it, s.end(), values, ascii::space, v))
  {
    std::ostream_iterator<int> out{std::cout, ";"};
    std::copy(v.begin(), v.end(), out);
  }
}
```

Example 11.13 works like Example 11.12. If you enter multiple integers delimited by commas, they are displayed with semicolons. In contrast to the previous example, the parser isn't passed directly to `boost::spirit::qi::phrase_parse()`, but defined in a `boost::spirit::qi::rule` variable.
`boost::spirit::qi::rule` is a class template. The only mandatory parameter is the iterator type of the string being parsed. In the example, two more optional template parameters are also passed.
The second template parameter is std::vector<int>(), which is the signature of a function that returns a vector of type std::vector<int> and expects no parameter. This template parameter indicates that the type of the attribute parsed is an int vector.
The third template parameter is the type of the skipper used by `boost::spirit::qi::phrase_parse()`.
In the example, the skipper **`boost::spirit::ascii::space`** is used. The type of this skipper is available through `boost::spirit::ascii::space_type` and is passed as a template parameter to `boost::spirit::qi::rule`.

Note

If you want your code to be platform independent and work with a C++11 development environment, you should prefer `boost::spirit::qi::rule` over the keyword `auto`. If **`values`** is defined with `auto`, the example works as expected with GCC and Clang. However with Visual C++ 2013, only the first number is parsed and written to standard output.

Example 11.14 Nesting Rules

```
#include <boost/spirit/include/qi.hpp>
#include <boost/variant.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::spirit;

struct print : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const
  {
    std::cout << std::boolalpha << t << ';';
  }
};

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  qi::rule<std::string::iterator, boost::variant<int, bool>(),
    ascii::space_type> value = qi::int_ | qi::bool_;
  qi::rule<std::string::iterator, std::vector<boost::variant<int, bool>>(),
    ascii::space_type> values = value % ',';
  std::vector<boost::variant<int, bool>> v;
  if (qi::phrase_parse(it, s.end(), values, ascii::space, v))
  {
    for (const auto &elem : v)
      boost::apply_visitor(print{}, elem);
  }
}
```

Example 11.14 defines two rules, one of which refers to the other: **`values`** is defined as `value % ','`, and **`value`** is set to `qi::int_ | qi::bool_`. **`values`** says that any number of values delimited by commas can be parsed. **`value`** defines a value as an integer or bool. Together, the rules say that integers and boolean values separated with commas can be entered in any order.

To store any number of values, a container of type `std::vector` is provided. Because the type of the values is either int or bool, a class is required that can store an int or a bool value. According to the overview on attribute types and operators, the class `boost::variant` from Boost.Variant must be used.

If you start the example and enter integers and boolean values delimited by commas, the values are written to the standard output stream delimited by semicolons. This is accomplished with the help of the function `boost::apply_visitor()`, which is provided by Boost.Variant. This function expects a visitor – an object of the class `print` in this example.

Please note that boolean values must be entered as **`true`** and **`false`**.

11.6 Grammar

If you want to parse complex formats and need to define multiple rules that refer to each other, you can group them with `boost::spirit::qi::grammar`.
Example 11.15 works like Example 11.14: you can enter integers and boolean values in any order, delimited by commas. They will be written to the standard output stream in the same order, but delimited by semicolons. The example uses the same rules – **`value`** and **`values`** – as the previous one. However, this time the rules are grouped in a grammar. The grammar is defined in a class called `my_grammar`, which is derived from `boost::spirit::qi::grammar`.
Both `my_grammar` and `boost::spirit::qi::grammar` are class templates. The template parameters expected by `boost::spirit::qi::grammar` are the same as those expected by `boost::spirit::qi::rule`. The iterator type of the string to be parsed has to be passed to `boost::spirit::qi::grammar`. You can also pass the signature of a function that defines the attribute type and the type of the skipper.
In `my_grammar`, `boost::spirit::qi::rule` is used to define the rules **`value`** and **`values`**. The rules are defined as member variables and are initialized in the constructor.
Please note that the outermost rule has to be passed with base_type to the constructor of the base class. This way, Boost.Spirit knows which rule is the entry point of the grammar.
Once a grammar is defined, it can be used like a parser. In Example 11.15, `my_grammar` is instantiated in `main()` to create **`g`**. **`g`** is then passed to `boost::spirit::qi::phrase_parse()`.

Example 11.15 Grouping rules in a grammar

```
#include <boost/spirit/include/qi.hpp>
#include <boost/variant.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::spirit;

template <typename Iterator, typename Skipper>
struct my_grammar : qi::grammar<Iterator,
  std::vector<boost::variant<int, bool>>(), Skipper>
{
  my_grammar() : my_grammar::base_type{values}
  {
    value = qi::int_ | qi::bool_;
    values = value % ',';
  }

  qi::rule<Iterator, boost::variant<int, bool>(), Skipper> value;
  qi::rule<Iterator, std::vector<boost::variant<int, bool>>(), Skipper>
    values;
};

struct print : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const
  {
    std::cout << std::boolalpha << t << ';';
  }
};

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  my_grammar<std::string::iterator, ascii::space_type> g;
  std::vector<boost::variant<int, bool>> v;
  if (qi::phrase_parse(it, s.end(), g, ascii::space, v))
  {
    for (const auto &elem : v)
      boost::apply_visitor(print{}, elem);
  }
}
```

Example 11.16 Storing parsed values in structures

```
#include <boost/spirit/include/qi.hpp>
#include <boost/variant.hpp>
#include <boost/fusion/include/adapt_struct.hpp>
#include <string>
#include <vector>
#include <iostream>

using namespace boost::spirit;

typedef boost::variant<int, bool> int_or_bool;

struct int_or_bool_values
{
  int_or_bool first;
  std::vector<int_or_bool> others;
};

BOOST_FUSION_ADAPT_STRUCT(
  int_or_bool_values,
  (int_or_bool, first)
  (std::vector<int_or_bool>, others)
)

template <typename Iterator, typename Skipper>
struct my_grammar : qi::grammar<Iterator, int_or_bool_values(), Skipper>
{
  my_grammar() : my_grammar::base_type{values}
  {
    value = qi::int_ | qi::bool_;
    values = value >> ',' >> value % ',';
  }

  qi::rule<Iterator, int_or_bool(), Skipper> value;
  qi::rule<Iterator, int_or_bool_values(), Skipper> values;
};

struct print : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const
  {
    std::cout << std::boolalpha << t << ';';
  }
};

int main()
{
  std::string s;
  std::getline(std::cin, s);
  auto it = s.begin();
  my_grammar<std::string::iterator, ascii::space_type> g;
  int_or_bool_values v;
  if (qi::phrase_parse(it, s.end(), g, ascii::space, v))
  {
    print p;
    boost::apply_visitor(p, v.first);
    for (const auto &elem : v.others)
```

```
      boost::apply_visitor(p, elem);
  }
}
```

Example 11.16 is based on the previous example, but expects at least two values. The rule **values** is defined as `value >> ',' >> value % ','`.
The first component in **values** is `value`, and the second one is `value % ','`. The value parsed by the first component has to be stored in an object of type `boost::variant`. The values parsed by the second component have to be stored in a container. With `int_or_bool_values`, the example provides a structure to store values parsed by both components of the rule **values**.
To use `int_or_bool_values` with Boost.Spirit, the macro `BOOST_FUSION_ADAPT_STRUCT` must be used. This macro is provided by Boost.Fusion. This macro makes it possible to treat `int_or_bool_values` like a tuple with two values of type int_or_bool and std::vector<int_or_bool>. Because this tuple has the right number of values with the right types, it is possible to define **values** with the signature `int_or_bool_values()`. **values** will store the first parsed value in **first** and all other parsed values in **others**.
An object of type `int_or_bool_values` is passed to `boost::spirit::qi::phrase_parse()` as an attribute. If you start the example and enter at least two integers or boolean values delimited by commas, they are all stored in the attribute and written to the standard output stream.

Note

The parser has been changed from what was used in the previous example. If **values** was defined with `value % ','`, `int_or_bool_values` would have only one member variable, and all parsed values could be stored in a vector, as in the previous example. Thus, `int_or_bool_values` would be like a tuple with only one value – which Boost.Spirit doesn't support. Structures with only one member variable will cause a compiler error. There are various workarounds for that problem.

Part III

Containers

Containers are one of the most useful data structures in C++. The standard library provides many containers, and the Boost libraries provide even more.

- Boost.MultiIndex goes one step further: the containers from this library can support multiple interfaces from other containers at the same time. Containers from Boost.MultiIndex are like merged containers and provide the advantages of all of the containers they have been merged with.
- Boost.Bimap is based on Boost.MultiIndex. It provides a container similar to `std::unordered_map`, except the elements can be looked up from both sides. Thus, depending on how the container is accessed, either side can be the key. When one side is the key, the other side is the value.
- Boost.Array and Boost.Unordered define the classes `boost::array`, `boost::unordered_set`, and `boost::unordered_map`, which were added to the standard library with C++11.
- Boost.CircularBuffer provides a container whose most important property is that it will overwrite the first element in the buffer when a value is added to a full circular buffer.
- Boost.Heap provides variants of priority queues – classes that resemble `std::priority_queue`.
- Boost.Intrusive lets you create containers that, unlike the containers from the standard library, neither copy nor move objects. However, to add an object to an intrusive list, the object’s type must meet certain requirements.
- Boost.MultiArray tries to simplify the use of multidimensional arrays. For example, it’s possible to treat part of a multidimensional array as a separate array.
- Boost.Container is a library that defines the same containers as the standard library. Using Boost.Container can make sense if, for example, you need to support a program on multiple platforms and you want to avoid problems caused by implementation-specific differences in the standard library.

Chapter 12

Boost.MultiIndex

Boost.MultiIndex makes it possible to define containers that support an arbitrary number of interfaces. While `std::vector` provides an interface that supports direct access to elements with an index and `std::set` provides an interface that sorts elements, Boost.MultiIndex lets you define containers that support both interfaces. Such a container could be used to access elements using an index and in a sorted fashion.

Boost.MultiIndex can be used if elements need to be accessed in different ways and would normally need to be stored in multiple containers. Instead of having to store elements in both a vector and a set and then synchronizing the containers continuously, you can define a container with Boost.MultiIndex that provides a vector interface and a set interface.

Boost.MultiIndex also makes sense if you need to access elements based on multiple different properties. In Example 12.1, animals are looked up by name and by number of legs. Without Boost.MultiIndex, two hash containers would be required – one to look up animals by name and the other to look them up by number of legs.

When you use Boost.MultiIndex, the first step is to define a new container. You have to decide which interfaces your new container should support and which element properties it should access.

The class `boost::multi_index::multi_index_container`, which is defined in `boost/multi_index_container.hpp`, is used for every container definition. This is a class template that requires at least two parameters. The first parameter is the type of elements the container should store – in Example 12.1 this is a user-defined class called `animal`. The second parameter is used to denote different indexes the container should provide.

Example 12.1 Using `boost::multi_index::multi_index_container`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  std::cout << animals.count("cat") << '\n';

  const animal_multi::nth_index<1>::type &legs_index = animals.get<1>();
  std::cout << legs_index.count(8) << '\n';
}
```

The key advantage of containers based on Boost.MultiIndex is that you can access elements via different interfaces. When you define a new container, you can specify the number and type of interfaces. The container in Example 12.1 needs to support searching for animals by name or number of legs, so two interfaces are defined. Boost.MultiIndex calls these interfaces indexes – that's where the library's name comes from.
Interfaces are defined with the help of the class `boost::multi_index::indexed_by`. Each interface is passed as a template parameter. Two interfaces of type `boost::multi_index::hashed_non_unique`, which is defined in `boost/multi_index/hashed_index.hpp`, are used in Example 12.1. Using these interfaces makes the container behave like `std::unordered_set` and look up values using a hash value.
The class `boost::multi_index::hashed_non_unique` is a template as well and expects as its sole parameter a class that calculates hash values. Because both interfaces of the container need to look up animals, one interface calculates hash values for the name, while the other interface does so for the number of legs.
Boost.MultiIndex offers the helper class template `boost::multi_index::member`, which is defined in `boost/multi_index/member.hpp`, to access a member variable. As seen in Example 12.1, several parameters have been specified to let `boost::multi_index::member` know which member variable of `animal` should be accessed and which type the member variable has.
Even though the definition of `animal_multi` looks complicated at first, the class works like a map. The name and number of legs of an animal can be regarded as a key/value pair. The advantage of the container `animal_multi` over a map like `std::unordered_map` is that animals can be looked up by name or by number of legs. `animal_multi` supports two interfaces, one based on the name and one based on the number of legs. The interface determines which member variable is the key and which member variable is the value.
To access a MultiIndex container, you need to select an interface. If you directly access the object **`animals`** using `insert()` or `count()`, the first interface is used. In Example 12.1, this is the hash container for the member variable **`name`**. If you need a different interface, you must explicitly select it.
Interfaces are numbered consecutively, starting at index 0 for the first interface. To access the second interface – as shown in Example 12.1 – call the member function `get()` and pass in the index of the desired interface as the template parameter.
The return value of `get()` looks complicated. It accesses a class of the MultiIndex container called `nth_index` which, again, is a template. The index of the interface to be used must be specified as a template parameter. This index must be the same as the one passed to `get()`. The final step is to access the type definition named type of `nth_index`. The value of type represents the type of the corresponding interface. The following examples use the keyword `auto` to simplify the code.
Although you do not need to know the specifics of an interface, since they are automatically derived from `nth_index` and type, you should still understand what kind of interface is accessed. Since interfaces are numbered consecutively in the container definition, this can be answered easily, since the index is passed to both `get()` and `nth_index`. Thus, **`legs_index`** is a hash interface that looks up animals by legs.
Because data such as names and legs can be keys of the MultiIndex container, they cannot be arbitrarily changed. If the number of legs is changed after an animal has been looked up by name, an interface using legs as a key would be unaware of the change and would not know that a new hash value needs to be calculated.

Example 12.2 Changing elements in a MultiIndex container with `modify()`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
```

```
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  auto &legs_index = animals.get<1>();
  auto it = legs_index.find(4);
  legs_index.modify(it, [](animal &a){ a.name = "dog"; });

  std::cout << animals.count("dog") << '\n';
}
```

Just as the keys in a container of type `std::unordered_map` cannot be modified, neither can data stored within a MultiIndex container. Strictly speaking, all data stored in a MultiIndex container is constant. This includes member variables that aren't used by any interface. Even if no interface accesses **`legs`**, **`legs`** cannot be changed. To avoid having to remove elements from a MultiIndex container and insert new ones, Boost.MultiIndex provides member functions to change values directly. Because these member functions operate on the MultiIndex container itself, and because no element in a container is modified directly, all interfaces will be notified and can calculate new hash values.

Every interface offered by Boost.MultiIndex provides the member function `modify()`, which operates directly on the container. The object to be modified is identified through an iterator passed as the first parameter to `modify()`. The second parameter is a function or function object that expects as its sole parameter an object of the type stored in the container. The function or function object can change the element as much as it wants. Example 12.2 illustrates how to use the member function `modify()` to change an element.

So far, only one interface has been introduced: `boost::multi_index::hashed_non_unique`, which calculates a hash value that does not have to be unique. In order to guarantee that no value is stored twice, use `boost::multi_index::hashed_unique`. Please note that values cannot be stored if they don't satisfy the requirements of all interfaces of a particular container. If one interface does not allow you to store values multiple times, it does not matter whether another interface does allow it.

Example 12.3 A MultiIndex container with `boost::multi_index::hashed_unique`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/member.hpp>
```

```
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
    >,
    hashed_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"dog", 4});

  auto &legs_index = animals.get<1>();
  std::cout << legs_index.count(4) << '\n';
}
```

The container in Example 12.3 uses `boost::multi_index::hashed_unique` as the second interface. That means no two animals with the same number of legs can be stored in the container because the hash values would be the same.
The example tries to store a dog, which has the same number of legs as the already stored cat. Because this violates the requirement of having unique hash values for the second interface, the dog will not be stored in the container. Therefore, when searching for animals with four legs, the program displays `1`, because only the cat was stored and counted.
Example 12.4 introduces the last three interfaces of Boost.MultiIndex: `boost::multi_index::sequenced`, `boost::multi_index::ordered_non_unique` , and `boost::multi_index::random_access`.
The interface `boost::multi_index::sequenced` allows you to treat a MultiIndex container like a list of type `std::list`. Elements are stored in the given order.
With the interface `boost::multi_index::ordered_non_unique`, objects are automatically sorted. This interface requires that you specify a sorting criterion when defining the container. Example 12.4 sorts objects of type `animal` by the number of legs using the helper class `boost::multi_index::member`.
`boost::multi_index::ordered_non_unique` provides special member functions to find specific ranges within the sorted values. Using `lower_bound()` and `upper_bound()`, the program searches for animals that have at least four and no more than eight legs. Because they require elements to be sorted, these member functions are not provided by other interfaces.
The final interface introduced is `boost::multi_index::random_access`, which allows you to treat the MultiIndex container like a vector of type `std::vector`. The two most prominent member functions are `operator[]` and `at()`.
`boost::multi_index::random_access` includes `boost::multi_index::sequenced`. With `boost::multi_index::random_access`, all member functions of `boost::multi_index::sequenced` are available

as well.

Example 12.4 The interfaces `sequenced`, `ordered_non_unique` and `random_access`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/sequenced_index.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/random_access_index.hpp>
#include <boost/multi_index/member.hpp>
#include <string>
#include <iostream>

using namespace boost::multi_index;

struct animal
{
  std::string name;
  int legs;
};

typedef multi_index_container<
  animal,
  indexed_by<
    sequenced<>,
    ordered_non_unique<
      member<
        animal, int, &animal::legs
      >
    >,
    random_access<>
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.push_back({"cat", 4});
  animals.push_back({"shark", 0});
  animals.push_back({"spider", 8});

  auto &legs_index = animals.get<1>();
  auto it = legs_index.lower_bound(4);
  auto end = legs_index.upper_bound(8);
  for (; it != end; ++it)
    std::cout << it->name << '\n';

  const auto &rand_index = animals.get<2>();
  std::cout << rand_index[0].name << '\n';
}
```

Example 12.5 The key extractors `identity` and `const_mem_fun`

```
#include <boost/multi_index_container.hpp>
#include <boost/multi_index/ordered_index.hpp>
#include <boost/multi_index/hashed_index.hpp>
#include <boost/multi_index/identity.hpp>
#include <boost/multi_index/mem_fun.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::multi_index;

class animal
{
```

```
public:
  animal(std::string name, int legs) : name_{std::move(name)},
    legs_(legs) {}
  bool operator<(const animal &a) const { return legs_ < a.legs_; }
  const std::string &name() const { return name_; }
private:
  std::string name_;
  int legs_;
};

typedef multi_index_container<
  animal,
  indexed_by<
    ordered_unique<
      identity<animal>
    >,
    hashed_unique<
      const_mem_fun<
        animal, const std::string&, &animal::name
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

  animals.emplace("cat", 4);
  animals.emplace("shark", 0);
  animals.emplace("spider", 8);

  std::cout << animals.begin()->name() << '\n';

  const auto &name_index = animals.get<1>();
  std::cout << name_index.count("shark") << '\n';
}
```

Now that we've covered the four interfaces of Boost.MultiIndex, the remainder of this chapter focuses on *key extractors*. One of the key extractors has already been introduced: `boost::multi_index::member`, which is defined in `boost/multi_index/member.hpp`. This helper class is called a key extractor because it allows you to specify which member variable of a class should be used as the key of an interface.

Example 12.5 introduces two more key extractors.

The key extractor `boost::multi_index::identity`, defined in `boost/multi_index/identity.hpp`, uses elements stored in the container as keys. This requires the class `animal` to be sortable because objects of type `animal` will be used as the key for the interface `boost::multi_index::ordered_unique`. In Example 12.5, this is achieved through the overloaded `operator<`.

The header file `boost/multi_index/mem_fun.hpp` defines two key extractors – `boost::multi_index::const_mem_fun` and `boost::multi_index::mem_fun` – that use the return value of a member function as a key. In Example 12.5, the return value of `name()` is used that way. `boost::multi_index::const_mem_fun` is used for constant member functions, while `boost::multi_index::mem_fun` is used for non-constant member functions.

Boost.MultiIndex offers two more key extractors: `boost::multi_index::global_fun` and `boost::multi_index::composite_key`. The former can be used for free-standing or static member functions, and the latter allows you to design a key extractor made up of several other key extractors.

Chapter 13

Boost.Bimap

The library Boost.Bimap is based on Boost.MultiIndex and provides a container that can be used immediately without being defined first. The container is similar to `std::map`, but supports looking up values from either side. Boost.Bimap allows you to create maps where either side can be the key, depending on how you access the map. When you access the left side as the key, the right side is the value, and vice versa.

Example 13.1 Using `boost::bimap`

```
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  std::cout << animals.left.count("cat") << '\n';
  std::cout << animals.right.count(8) << '\n';
}
```

`boost::bimap` is defined in `boost/bimap.hpp` and provides two member variables, **`left`** and **`right`**, which can be used to access the two containers of type `std::map` that are unified by `boost::bimap`. In Example 13.1, **`left`** uses keys of type `std::string` to access the container, and **`right`** uses keys of type int.

Besides supporting access to individual records using a left or right container, `boost::bimap` allows you to view records as relations (see Example 13.2).

Example 13.2 Accessing relations

```
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string, int> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  for (auto it = animals.begin(); it != animals.end(); ++it)
    std::cout << it->left << " has " << it->right << " legs\n";
}
```

It is not necessary to access records using **`left`** or **`right`**. By iterating over records, the left and right parts of an individual record are made available through the iterator.

Example 13.3 Using `boost::bimaps::set_of` explicitly

```
#include <boost/bimap.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>,
    boost::bimaps::set_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  std::cout << animals.left.count("spider") << '\n';
  std::cout << animals.right.count(8) << '\n';
}
```

While `std::map` is accompanied by a container called `std::multimap`, which can store multiple records using the same key, there is no such equivalent for `boost::bimap`. However, this does not mean that storing multiple records with the same key inside a container of type `boost::bimap` is impossible. Strictly speaking, the two required template parameters specify container types for **`left`** and **`right`**, not the types of the elements to store. If no container type is specified, the container type `boost::bimaps::set_of` is used by default. This container, like `std::map`, only accepts records with unique keys.

Example 13.3 specifies `boost::bimaps::set_of`.

Other container types besides `boost::bimaps::set_of` can be used to customize `boost::bimap`.

Example 13.4 Allowing duplicates with `boost::bimaps::multiset_of`

```
#include <boost/bimap.hpp>
#include <boost/bimap/multiset_of.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<boost::bimaps::set_of<std::string>,
    boost::bimaps::multiset_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"dog", 4});

  std::cout << animals.left.count("dog") << '\n';
  std::cout << animals.right.count(4) << '\n';
}
```

Example 13.4 uses the container type `boost::bimaps::multiset_of`, which is defined in `boost/bimap/multiset_of.hpp`. It works like `boost::bimaps::set_of`, except that keys don't need to be unique. Example 13.4 will successfully display `2` when searching for animals with four legs.

Because `boost::bimaps::set_of` is used by default for containers of type `boost::bimap`, the header file `boost/bimap/set_of.hpp` does not need to be included explicitly. However, when using other container types, the corresponding header files must be included.

In addition to the classes shown above, Boost.Bimap provides the following: `boost::bimaps::unordered_set_of`, `boost::bimaps::unordered_multiset_of`, `boost::bimaps::list_of`, `boost::bimaps::vector_of`, and `boost::bimaps::unconstrained_set_of`. Except for `boost::bimaps::unconstrained_set_of`, all of the other container types operate just like their counterparts from the standard library.

Example 13.5 Disabling one side with `boost::bimaps::unconstrained_set_of`

```
#include <boost/bimap.hpp>
#include <boost/bimap/unconstrained_set_of.hpp>
#include <boost/bimap/support/lambda.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::bimap<std::string,
    boost::bimaps::unconstrained_set_of<int>> bimap;
  bimap animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});

  auto it = animals.left.find("cat");
  animals.left.modify_key(it, boost::bimaps::_key = "dog");

  std::cout << it->first << '\n';
}
```

`boost::bimaps::unconstrained_set_of` can be used to disable one side of `boost::bimap`. In Example 13.5, `boost::bimap` behaves like `std::map`. You can't access **`right`** to search for animals by legs.
Example 13.5 illustrates another reason why it can make sense to prefer `boost::bimap` over `std::map`. Since Boost.Bimap is based on Boost.MultiIndex, member functions from Boost.MultiIndex are available. Example 13.5 modifies a key using `modify_key()` – something that is not possible with `std::map`.
Note how the key is modified. A new value is assigned to the current key using **`boost::bimaps::_key`**, which is a placeholder that is defined in `boost/bimap/support/lambda.hpp`.
`boost/bimap/support/lambda.hpp` also defines **`boost::bimaps::_data`**. When calling the member function `modify_data()`, **`boost::bimaps::_data`** can be used to modify a value in a container of type `boost::bimap`.

Chapter 14

Boost.Array

The library Boost.Array defines the class template `boost::array` in `boost/array.hpp`. `boost::array` is similar to `std::array`, which was added to the standard library with C++11. You can ignore `boost::array` if you work with a C++11 development environment.

Example 14.1 Various member functions of `boost::array`

```
#include <boost/array.hpp>
#include <string>
#include <algorithm>
#include <iostream>

int main()
{
  typedef boost::array<std::string, 3> array;
  array a;

  a[0] = "cat";
  a.at(1) = "shark";
  *a.rbegin() = "spider";

  std::sort(a.begin(), a.end());

  for (const std::string &s : a)
    std::cout << s << '\n';

  std::cout << a.size() << '\n';
  std::cout << a.max_size() << '\n';
}
```

With `boost::array`, an array can be created that exhibits the same properties as a C array. In addition, `boost::array` conforms to the requirements of C++ containers, which makes handling such an array as easy as handling any other container. In principle, one can think of `boost::array` as the container `std::vector`, except the number of elements in `boost::array` is constant.
As seen in Example 14.1, using `boost::array` is fairly simple and needs no additional explanation since the member functions called have the same meaning as their counterparts from `std::vector`.

Chapter 15

Boost.Unordered

Boost.Unordered provides the classes `boost::unordered_set`, `boost::unordered_multiset`, `boost::unordered_map`, and `boost::unordered_multimap`. These classes are identical to the hash containers that were added to the standard library with C++11. Thus, you can ignore the containers from Boost.Unordered if you work with a development environment supporting C++11.

Example 15.1 Using `boost::unordered_set`

```
#include <boost/unordered_set.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::unordered_set<std::string> unordered_set;
  unordered_set set;

  set.emplace("cat");
  set.emplace("shark");
  set.emplace("spider");

  for (const std::string &s : set)
    std::cout << s << '\n';

  std::cout << set.size() << '\n';
  std::cout << set.max_size() << '\n';

  std::cout << std::boolalpha << (set.find("cat") != set.end()) << '\n';
  std::cout << set.count("shark") << '\n';
}
```

`boost::unordered_set` can be replaced with `std::unordered_set` in Example 15.1. `boost::unordered_set` doesn't differ from `std::unordered_set`.

Example 15.2 Using `boost::unordered_map`

```
#include <boost/unordered_map.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::unordered_map<std::string, int> unordered_map;
  unordered_map map;

  map.emplace("cat", 4);
  map.emplace("shark", 0);
  map.emplace("spider", 8);

  for (const auto &p : map)
    std::cout << p.first << ";" << p.second << '\n';
  std::cout << map.size() << '\n';
  std::cout << map.max_size() << '\n';

  std::cout << std::boolalpha << (map.find("cat") != map.end()) << '\n';
  std::cout << map.count("shark") << '\n';
}
```

Example 15.2 uses `boost::unordered_map` to store the names and the number of legs for several animals. Once again, `boost::unordered_map` could be replaced with `std::unordered_map`.
In Example 15.3 elements of type `animal` are stored in a container of type `boost::unordered_set`. Because the hash function of `boost::unordered_set` doesn't know the class `animal`, hash values can't be automatically calculated for elements of this type. That's why a hash function must be defined – otherwise the example can't be compiled.
The name of the hash function to define is `hash_value()`. It must expect as its sole parameter an object of the type the hash value should be calculated for. The type of the return value of `hash_value()` must be std::size_t. The function `hash_value()` is automatically called when the hash value has to be calculated for an object. This function is defined for various types in the Boost libraries, including `std::string`. For user-defined types like `animal`, it must be defined by the developer.
Usually, the definition of `hash_value()` is rather simple: Hash values are created by accessing the member variables of an object one after another. This is done with the function `boost::hash_combine()`, which is provided by Boost.Hash and defined in `boost/functional/hash.hpp`. You don't have to include this header file if you use Boost.Unordered because all containers from this library access Boost.Hash to calculate hash values.

Example 15.3 User-defined type with Boost.Unordered

```
#include <boost/unordered_set.hpp>
#include <string>
#include <cstddef>

struct animal
{
  std::string name;
  int legs;
};

bool operator==(const animal &lhs, const animal &rhs)
{
  return lhs.name == rhs.name && lhs.legs == rhs.legs;
}

std::size_t hash_value(const animal &a)
{
  std::size_t seed = 0;
  boost::hash_combine(seed, a.name);
  boost::hash_combine(seed, a.legs);
  return seed;
}

int main()
{
  typedef boost::unordered_set<animal> unordered_set;
  unordered_set animals;

  animals.insert({"cat", 4});
  animals.insert({"shark", 0});
  animals.insert({"spider", 8});
}
```

In addition to defining `hash_value()`, you need to make sure two objects can be compared using `==`. That's why the operator `operator==` is overloaded for `animal` in Example 15.3.
The hash containers from the C++11 standard library use a hash function from the header file `functional`.

The hash containers from Boost.Unordered expect the hash function `hash_value()`. Whether you use Boost.Hash within `hash_value()` doesn't matter. Boost.Hash makes sense because functions like `boost::hash_combine()` make it easier to calculate hash values from multiple member variables step by step. However, this is only an implementation detail of `hash_value()`. Apart from the different hash functions used, the hash containers from Boost.Unordered and the standard library are basically equivalent.

Chapter 16

Boost.CircularBuffer

The library Boost.CircularBuffer provides a *circular buffer*, which is a container with the following two fundamental properties:

- The capacity of the circular buffer is constant and set by you. The capacity doesn't change automatically when you call a member function such as `push_back()`. Only you can change the capacity of the circular buffer. The size of the circular buffer can not exceed the capacity you set.
- Despite constant capacity, you can call `push_back()` as often as you like to insert elements into the circular buffer. If the maximum size has been reached and the circular buffer is full, elements are overwritten.

A circular buffer makes sense when the amount of available memory is limited, and you need to prevent a container from growing arbitrarily big. Another example is continuous data flow where old data becomes irrelevant as new data becomes available. Memory is automatically reused by overwriting old data.
To use the circular buffer from Boost.CircularBuffer, include the header file `boost/circular_buffer.hpp`. This header file defines the class `boost::circular_buffer`.
`boost::circular_buffer` is a template and must be instantiated with a type. For instance, the circular buffer **`cb`** in Example 16.1 stores numbers of type int.
The capacity of the circular buffer is specified when instantiating the class, not through a template parameter. The default constructor of `boost::circular_buffer` creates a buffer with a capacity of zero elements. Another constructor is available to set the capacity. In Example 16.1, the buffer **`cb`** has a capacity of three elements.

Example 16.1 Using `boost::circular_buffer`

```
#include <boost/circular_buffer.hpp>
#include <iostream>

int main()
{
  typedef boost::circular_buffer<int> circular_buffer;
  circular_buffer cb{3};

  std::cout << cb.capacity() << '\n';
  std::cout << cb.size() << '\n';

  cb.push_back(0);
  cb.push_back(1);
  cb.push_back(2);

  std::cout << cb.size() << '\n';

  cb.push_back(3);
  cb.push_back(4);
  cb.push_back(5);

  std::cout << cb.size() << '\n';

  for (int i : cb)
    std::cout << i << '\n';
}
```

The capacity of a circular buffer can be queried by calling `capacity()`. In Example 16.1, `capacity()` will return 3.
The capacity is not equivalent to the number of stored elements. While the return value of `capacity()` is constant, `size()` returns the number of elements in the buffer, which may be different. The return value of `size()` will always be between 0 and the capacity of the circular buffer.
Example 16.1 returns 0 the first time `size()` is called since the buffer does not contain any data. After calling `push_back()` three times, the buffer contains three elements, and the second call to `size()` will return 3. Calling `push_back()` again does not cause the buffer to grow. The three new numbers overwrite the previous three. Therefore, `size()` will return 3 when called for the third time.
As a verification, the stored numbers are written to standard output at the end of Example 16.1. The output contains the numbers 3, 4, and 5 since the previously stored numbers have been overwritten.

Example 16.2 Various member functions of `boost::circular_buffer`

```
#include <boost/circular_buffer.hpp>
#include <iostream>

int main()
{
  typedef boost::circular_buffer<int> circular_buffer;
  circular_buffer cb{3};

  cb.push_back(0);
  cb.push_back(1);
  cb.push_back(2);
  cb.push_back(3);

  std::cout << std::boolalpha << cb.is_linearized() << '\n';

  circular_buffer::array_range ar1, ar2;

  ar1 = cb.array_one();
  ar2 = cb.array_two();
  std::cout << ar1.second << ";" << ar2.second << '\n';

  for (int i : cb)
    std::cout << i << '\n';

  cb.linearize();

  ar1 = cb.array_one();
  ar2 = cb.array_two();
  std::cout << ar1.second << ";" << ar2.second << '\n';
}
```

Example 16.2 uses the member functions `is_linearized()`, `array_one()`, `array_two()` and `linearize()`, which do not exist in other containers. These member functions clarify the internals of the circular buffer.
A circular buffer is essentially comparable to `std::vector`. Because the beginning and end are well defined, a vector can be treated as a conventional C array. That is, memory is contiguous, and the first and last elements are always at the lowest and highest memory address. However, a circular buffer does not offer such a guarantee.
Even though it may sound strange to talk about the beginning and end of a circular buffer, they do exist. Elements can be accessed via iterators, and `boost::circular_buffer` provides member functions such as `begin()` and `end()`. While you don't need to be concerned about the position of the beginning and end when using iterators, the situation becomes a bit more complicated when accessing elements using regular pointers, unless you use `is_linearized()`, `array_one()`, `array_two()`, and `linearize()`.
The member function `is_linearized()` returns `true` if the beginning of the circular buffer is at the lowest memory address. In this case, all the elements in the buffer are stored consecutively from beginning to the end at increasing memory addresses, and elements can be accessed like a conventional C array.
If `is_linearized()` returns `false`, the beginning of the circular buffer is not at the lowest memory address, which is the case in Example 16.2. While the first three elements 0, 1, and 2 are stored in exactly this order, calling `push_back()` for the fourth time will overwrite the number 0 with the number 3. Because 3 is the last element added by a call to `push_back()`, it is now the new end of the circular buffer. The beginning is now the el-

ement with the number 1, which is stored at the next higher memory address. This means elements are no longer stored consecutively at increasing memory addresses.

If the end of the circular buffer is at a lower memory address than the beginning, the elements can be accessed via two conventional C arrays. To avoid the need to calculate the position and size of each array, `boost::circular_buffer` provides the member functions `array_one()` and `array_two()`.

Both `array_one()` and `array_two()` return a `std::pair` whose first element is a pointer to the corresponding array and whose second element is the size. `array_one()` accesses the array at the beginning of the circular buffer, and `array_two()` accesses the array at the end of the buffer.

If the circular buffer is linearized and `is_linearized()` returns `true`, `array_two()` can be called, too. However, since there is only one array in the buffer, the second array contains no elements.

To simplify matters and treat the circular buffer as a conventional C array, you can force a rearrangement of the elements by calling `linearize()`. Once complete, you can access all stored elements using `array_one()`, and you don't need to use `array_two()`.

Boost.CircularBuffer offers an additional class called `boost::circular_buffer_space_optimized`. This class is also defined in `boost/circular_buffer.hpp`. Although this class is used in the same way as `boost::circular_buffer`, it does not reserve any memory at instantiation. Rather, memory is allocated dynamically when elements are added until the capacity is reached. Removing elements releases memory accordingly. `boost::circular_buffer_space_optimized` manages memory more efficiently and, therefore, can be a better choice in certain scenarios. For example, it may be a good choice if you need a circular buffer with a large capacity, but your program doesn't always use the full buffer.

Chapter 17

Boost.Heap

Boost.Heap could have also been called Boost.PriorityQueue since the library provides several priority queues. However, the priority queues in Boost.Heap differ from `std::priority_queue` by supporting more functions.

Example 17.1 Using `boost::heap::priority_queue`

```
#include <boost/heap/priority_queue.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  priority_queue<int> pq;
  pq.push(2);
  pq.push(3);
  pq.push(1);

  for (int i : pq)
    std::cout << i << '\n';

  priority_queue<int> pq2;
  pq2.push(4);
  std::cout << std::boolalpha << (pq > pq2) << '\n';
}
```

Example 17.1 uses the class `boost::heap::priority_queue`, which is defined in `boost/heap/priority_queue.hpp`. In general this class behaves like `std::priority_queue`, except it allows you to iterate over elements. The order of elements returned in the iteration is random.

Objects of type `boost::heap::priority_queue` can be compared with each other. The comparison in Example 17.1 returns `true` because **pq** has more elements than **pq2**. If both queues had the same number of elements, the elements would be compared in pairs.

Example 17.2 Using `boost::heap::binomial_heap`

```
#include <boost/heap/binomial_heap.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  binomial_heap<int> bh;
  bh.push(2);
  bh.push(3);
  bh.push(1);

  binomial_heap<int> bh2;
  bh2.push(4);
  bh.merge(bh2);
  for (auto it = bh.ordered_begin(); it != bh.ordered_end(); ++it)
    std::cout << *it << '\n';
  std::cout << std::boolalpha << bh2.empty() << '\n';
}
```

Example 17.2 introduces the class `boost::heap::binomial_heap`. In addition to allowing you to iterate over elements in priority order, it also lets you merge priority queues. Elements from one queue can be added to another queue.
The example calls `merge()` on the queue **bh**. The queue **bh2** is passed as a parameter. The call to `merge()` moves the number 4 from **bh2** to **bh**. After the call, **bh** contains four numbers, and **bh2** is empty.
The `for` loop calls `ordered_begin()` and `ordered_end()` on **bh**. `ordered_begin()` returns an iterator that iterates from high priority elements to low priority elements. Thus, Example 17.2 writes the numbers 4, 3, 2, and 1 in order to standard output.
`boost::heap::binomial_heap` lets you change elements after they have been added to the queue. Example 17.3 saves a handle returned by `push()`, making it possible to access the number 2 stored in **bh**.
`update()` is a member function of `boost::heap::binomial_heap` that can be called to change an element. Example 17.3 calls the member function to replace 2 with 4. Afterwards, the element with the highest priority, now 4, is fetched with `top()`.

Example 17.3 Changing elements in `boost::heap::binomial_heap`

```
#include <boost/heap/binomial_heap.hpp>
#include <iostream>

using namespace boost::heap;

int main()
{
  binomial_heap<int> bh;
  auto handle = bh.push(2);
  bh.push(3);
  bh.push(1);

  bh.update(handle, 4);

  std::cout << bh.top() << '\n';
}
```

In addition to `update()`, `boost::heap::binomial_heap` provides other member functions to change elements. The member functions `increase()` or `decrease()` can be called if you know in advance whether a change will result in a higher or lower priority. In Example 17.3, the call to `update()` could be replaced with a call to `increase()` since the number is increased from 2 to 4.
Boost.Heap provides additional priority queues whose member functions mainly differ in their runtime complexity. For example, you can use the class `boost::heap::fibonacci_heap` if you want the member function `push()` to have a constant runtime complexity. The documentation on Boost.Heap provides a table with an overview of the runtime complexities of the various classes and functions.

Chapter 18

Boost.Intrusive

Boost.Intrusive is a library especially suited for use in high performance programs. The library provides tools to create *intrusive containers*. These containers replace the known containers from the standard library. Their disadvantage is that they can't be used as easily as, for example, `std::list` or `std::set`. But they have these advantages:

- Intrusive containers don't allocate memory dynamically. A call to `push_back()` doesn't lead to a dynamic allocation with `new`. This is a one reason why intrusive containers can improve performance.
- Intrusive containers store the original objects, not copies. After all, they don't allocate memory dynamically. This leads to another advantage: Member functions such as `push_back()` don't throw exceptions because they neither allocate memory nor copy objects.

The advantages are paid for with more complicated code because preconditions must be met to store objects in intrusive containers. You cannot store objects of arbitrary types in intrusive containers. For example, you cannot put strings of type `std::string` in an intrusive container; instead you must use containers from the standard library.
Example 18.1 prepares a class `animal` to allow objects of this type to be stored in an intrusive list.
In a list, an element is always accessed from another element, usually using a pointer. If an intrusive list is to store objects of type `animal` without dynamic memory allocation, pointers must exist somewhere to concatenate elements.
To store objects of type `animal` in an intrusive list, the class must provide the variables required by the intrusive list to concatenate elements. Boost.Intrusive provides *hooks* – classes from which the required variables are inherited. To allow objects of the type `animal` to be stored in an intrusive list, `animal` must be derived from the class `boost::intrusive::list_base_hook`.

Example 18.1 Using `boost::intrusive::list`

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal a3{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(a3);

  a1.name = "dog";

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Hooks make it possible to ignore the implementation details. However, it's safe to assume that `boost::intrusive::list_base_hook` provides at least two pointers because `boost::intrusive::list` is a doubly linked list. Thanks to the base class `boost::intrusive::list_base_hook`, `animal` defines these two pointers to allow objects of this type to be concatenated.

Please note that `boost::intrusive::list_base_hook` is a template that comes with default template parameters. Thus, no types need to be passed explicitly.

Boost.Intrusive provides the class `boost::intrusive::list` to create an intrusive list. This class is defined in `boost/intrusive/list.hpp` and is used like `std::list`. Elements can be added using `push_back()`, and it's also possible to iterate over elements.

It is important to understand that intrusive containers do not store copies; they store the original objects. Example 18.1 writes `dog`, `shark`, and `spider` to standard output – not `cat`. The object **`a1`** is linked into the list. That's why the change of the name is visible when the program iterates over the elements in the list and displays the names.

Because intrusive containers don't store copies, you must remove objects from intrusive containers before you destroy them.

Example 18.2 Removing and destroying dynamically allocated objects

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  animals.pop_back();
  delete a3;

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Example 18.2 creates an object of type `animal` with `new` and inserts it to the list **`animals`**. If you want to destroy the object with `delete` when you don't need it anymore, you must remove it from the list. Make sure that you remove the object from the list before you destroy it – the order is important. Otherwise, the pointers in the elements of the intrusive container might refer to a memory location that no longer contains an object of type `animal`.
Because intrusive containers neither allocate nor free memory, objects stored in an intrusive container continue to exist when the intrusive container is destroyed.
Since removing elements from intrusive containers doesn't automatically destroy them, the containers provide non-standard extensions. `pop_back_and_dispose()` is one such member function.

Example 18.3 Removing and destroying with `pop_back_and_dispose()`

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal : public list_base_hook<>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef list<animal> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  animals.pop_back_and_dispose([](animal *a){ delete a; });

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

`pop_back_and_dispose()` removes an element from a list and destroys it. Because intrusive containers don't know how an element should be destroyed, you need to pass to `pop_back_and_dispose()` a function or function object that does know how to destroy the element. `pop_back_and_dispose()` will remove the object from the list, then call the function or function object and pass it a pointer to the object to be destroyed. Example 18.3 passes a lambda function that calls `delete`.
In Example 18.3, only the third element in **`animals`** can be removed with `pop_back_and_dispose()`. The other elements in the list haven't been created with `new` and, thus, must not be destroyed with `delete`.
Boost.Intrusive supports another mechanism to link removing and destroying of elements.

Example 18.4 Removing and destroying with auto unlink mode

```
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

typedef link_mode<auto_unlink> mode;
```

```
struct animal : public list_base_hook<mode>
{
  std::string name;
  int legs;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal *a3 = new animal{"spider", 8};

  typedef constant_time_size<false> constant_time_size;
  typedef list<animal, constant_time_size> animal_list;
  animal_list animals;

  animals.push_back(a1);
  animals.push_back(a2);
  animals.push_back(*a3);

  delete a3;

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

Hooks support a parameter to set a link mode. The link mode is set with the class template `boost::intrusive::link_mode`. If `boost::intrusive::auto_unlink` is passed as a template parameter, the auto unlink mode is selected.
The auto unlink mode automatically removes an element from an intrusive container when it is destroyed. Example 18.4 writes only `cat` and `shark` to standard output.
The auto unlink mode can only be used if the member function `size()`, which is provided by all intrusive containers, has no *constant complexity*. By default, it has constant complexity, which means: the time it takes for `size()` to return the number of elements doesn't depend on how many elements are stored in a container. Switching constant complexity on or off is another option to optimize performance.
To change the complexity of `size()`, use the class template `boost::intrusive::constant_time_size`, which expects either `true` or `false` as a template parameter. `boost::intrusive::constant_time_size` can be passed as a second template parameter to intrusive containers, such as `boost::intrusive::list`, to set the complexity for `size()`.
Now that we've seen that intrusive containers support link mode and that there is an option to set the complexity for `size()`, it might seem as though there is still much more to discover, but there actually isn't. There are, for example, only three link modes supported, and auto unlink mode is the only one you need to know. The default mode used if you don't pick a link mode is good enough for all other use cases.
Furthermore, there are no options for other member functions. There are no other classes, other than `boost::intrusive::constant_time_size`, that you need to learn about.
Example 18.5 introduces a hook mechanism using another intrusive container: `boost::intrusive::set`.

Example 18.5 Defining a hook for `boost::intrusive::set` as a member variable

```
#include <boost/intrusive/set.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::intrusive;

struct animal
{
  std::string name;
  int legs;
  set_member_hook<> set_hook;
  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"shark", 0};
  animal a3{"spider", 8};

  typedef member_hook<animal, set_member_hook<>, &animal::set_hook> hook;
  typedef set<animal, hook> animal_set;
  animal_set animals;

  animals.insert(a1);
  animals.insert(a2);
  animals.insert(a3);

  for (const animal &a : animals)
    std::cout << a.name << '\n';
}
```

There are two ways to add a hook to a class: either derive the class from a hook or define the hook as a member variable. While the previous examples derived a class from `boost::intrusive::list_base_hook`, Example 18.5 uses the class `boost::intrusive::set_member_hook` to define a member variable.
Please note that the name of the member variable doesn't matter. However, the hook class you use depends on the intrusive container. For example, to define a hook as a member variable for an intrusive list, use `boost::intrusive::list_member_hook` instead of `boost::intrusive::set_member_hook`.
Intrusive containers have different hooks because they have different requirements for elements. However, you can use different several hooks to allow objects to be stored in multiple intrusive containers. `boost::intrusive::any_base_hook` and `boost::intrusive::any_member_hook` let you store objects in any intrusive container. Thanks to these classes, you don't need to derive from multiple hooks or define multiple member variables as hooks.
Intrusive containers expect hooks to be defined in base classes by default. If a member variable is used as a hook, as in Example 18.5, the intrusive container has to be told which member variable to use. That's why both `animal` and the type hook are passed to `boost::intrusive::set`. hook is defined with `boost::intrusive::member_hook`, which is used whenever a member variable serves as a hook. `boost::intrusive::member_hook` expects the element type, the type of the hook, and a pointer to the member variable as template parameters.
Example 18.5 writes `shark`, `cat`, and `spider`, in that order, to standard output.
In addition to the classes `boost::intrusive::list` and `boost::intrusive::set` introduced in this chapter, Boost.Intrusive also provides, for example, `boost::intrusive::slist` for singly linked lists and `boost::intrusive::unordered_set` for hash containers.

Chapter 19

Boost.MultiArray

Boost.MultiArray is a library that simplifies using arrays with multiple dimensions. The most important advantage is that multidimensional arrays can be used like containers from the standard library. For example, there are member functions, such as `begin()` and `end()`, that let you access elements in multidimensional arrays through iterators. Iterators are easier to use than the pointers normally used with C arrays, especially with arrays that have many dimensions.

Example 19.1 One-dimensional array with `boost::multi_array`

```
#include <boost/multi_array.hpp>
#include <iostream>

int main()
{
  boost::multi_array<char, 1> a{boost::extents[6]};

  a[0] = 'B';
  a[1] = 'o';
  a[2] = 'o';
  a[3] = 's';
  a[4] = 't';
  a[5] = '\0';

  std::cout << a.origin() << '\n';
}
```

Boost.MultiArray provides the class `boost::multi_array` to create arrays. This is the most important class provided. It is defined in `boost/multi_array.hpp`.

`boost::multi_array` is a template expecting two parameters: The first parameter is the type of the elements to store in the array. The second parameter determines how many dimensions the array should have.

The second parameter only sets the number of dimensions, not the number of elements in each dimension. Thus, in Example 19.1, **a** is a one-dimensional array.

The number of elements in a dimension is set at runtime. Example 19.1 uses the global object **`boost::extents`** to set dimension sizes. This object is passed to the constructor of **a**.

An object of type `boost::multi_array` can be used like a normal C array. Elements are accessed by passing an index to `operator[]`. Example 19.1 stores five letters and a null character in **a** – a one-dimensional array with six elements. `origin()` returns a pointer to the first element. The example uses this pointer to write the word stored in the array – `Boost` – to standard output.

Unlike containers from the standard library, `operator[]` checks whether an index is valid. If an index is not valid, the program exits with `std::abort()`. If you don't want the validity of indexes to be checked, define the macro `BOOST_DISABLE_ASSERTS` before you include `boost/multi_array.hpp`.

Example 19.2 Views and subarrays of a two-dimensional array

```
#include <boost/multi_array.hpp>
#include <algorithm>
#include <iostream>
#include <cstring>
```

```
int main()
{
  boost::multi_array<char, 2> a{boost::extents[2][6]};

  typedef boost::multi_array<char, 2>::array_view<1>::type array_view;
  typedef boost::multi_array_types::index_range range;
  array_view view = a[boost::indices[0][range{0, 5}]];

  std::memcpy(view.origin(), "tsooB", 6);
  std::reverse(view.begin(), view.end());

  std::cout << view.origin() << '\n';

  boost::multi_array<char, 2>::reference subarray = a[1];
  std::memcpy(subarray.origin(), "C++", 4);

  std::cout << subarray.origin() << '\n';
}
```

Example 19.2 creates a two-dimensional array. The number of elements in the first dimension is set to 2 and for the second dimension set to 6. Think of the array as a table with two rows and six columns.
The first row of the table will contain the word Boost. Since only five letters need to be stored for this word, a *view* is created which spans exactly five elements of the array.
A view, which is based on the class `boost::multi_array::array_view`, lets you access a part of an array and treat that part as though it were a separate array.
`boost::multi_array::array_view` is a template that expects the number of dimensions in the view as a template parameter. In Example 19.2 the number of dimensions for the view is 1. Because the array **`a`** has two dimensions, one dimension is ignored. To save the word Boost, a one-dimensional array is sufficient; more dimensions would be confusing.
As with `boost::multi_array`, the number of dimensions is passed in as a template parameter, and the size of each dimension is set at runtime. However, with `boost::multi_array::array_view` this isn't done with **`boost::extents`**. Instead it's done with **`boost::indices`**, which is another global object provided by Boost.MultiArray.
As with **`boost::extents`**, indexes must be passed to **`boost::indices`**. While only numbers may be passed to **`boost::extents`**, **`boost::indices`** accepts also ranges. These are defined using `boost::multi_array_types::index_range`.
In Example 19.2, the first parameter passed to **`boost::indices`** isn't a range, it's the number 0. When a number is passed, you cannot use `boost::multi_array_types::index_range`. In the example, the view will take the first dimension of **`a`** – the one with index 0.
For the second parameter, `boost::multi_array_types::index_range` is used to define a range. By passing 0 and 5 to the constructor, the first five elements of the first dimension of **`a`** are made available. The range starts at index 0 and ends at index 5 – excluding the element at index 5. The sixth element in the first dimension is ignored.
Thus, **`view`** is a one-dimensional array consisting of five elements – the first five elements in the first row of **`a`**. When **`view`** is accessed to copy a string with `std::memcpy()` and reverse the elements with `std::reverse()`, this relation doesn't matter. Once the view is created, it acts like an independent array with five elements.
When `operator[]` is called on an array of type `boost::multi_array`, the return value depends on the number of dimensions. In Example 19.1, the operator returns char elements because the array accessed is one dimensional.
In Example 19.2, **`a`** is a two-dimensional array. Thus, `operator[]` returns a subarray rather than a char element. Because the type of the subarray isn't public, `boost::multi_array::reference` must be used. This type isn't identical to `boost::multi_array::array_view`, even if the subarray behaves like a view. A view must be defined explicitly and can span arbitrary parts of an array, whereas a subarray is automatically returned by `operator[]` and spans all elements in every dimension.

Example 19.3 Wrapping a C array with `boost::multi_array_ref`

```
#include <boost/multi_array.hpp>
#include <algorithm>
#include <iostream>
#include <cstring>

int main()
```

```
{
  char c[12] =
  {
    't', 's', 'o', 'o', 'B', '\0',
    'C', '+', '+', '\0', '\0', '\0'
  };

  boost::multi_array_ref<char, 2> a{c, boost::extents[2][6]};

  typedef boost::multi_array<char, 2>::array_view<1>::type array_view;
  typedef boost::multi_array_types::index_range range;
  array_view view = a[boost::indices[0][range{0, 5}]];

  std::reverse(view.begin(), view.end());
  std::cout << view.origin() << '\n';

  boost::multi_array<char, 2>::reference subarray = a[1];
  std::cout << subarray.origin() << '\n';
}
```

The class `boost::multi_array_ref` wraps an existing C array. In Example 19.3, **a** provides the same interface as `boost::multi_array`, but without allocating memory. With `boost::multi_array_ref`, a C array – no matter how many dimensions it has – can be treated like a multidimensional array of type `boost::multi_array`. The C array just needs to be added as an additional parameter to the constructor.

Boost.MultiArray also provides the class `boost::const_multi_array_ref`, which treats a C array as a constant multidimensional array.

Chapter 20

Boost.Container

Boost.Container is a Boost library that provides the same containers as the standard library. Boost.Container focuses on additional flexibility. For example, all containers from this library can be used with Boost.Interprocess in shared memory – something that is not always possible with containers from the standard library.
Boost.Container provides additional advantages:

- The interfaces of the containers resemble those of the containers in the C++11 standard library. For example, they provide member functions such as `emplace_back()`, which you can use in a C++98 program even though it wasn't added to the standard library until C++11.
- With `boost::container::slist` or `boost::container::stable_vector`, Boost.Container offers containers the standard library doesn't provide.
- The implementation is platform independent. The containers behave the same everywhere. You don't need to worry about possible differences between implementations of the standard library.
- The containers from Boost.Container support *incomplete types* and can be used to define recursive containers.

Example 20.1 illustrates incomplete types.

Note

The examples in this chapters cannot be compiled with Visual C++ 2013 and Boost 1.55.0. This bug is described in ticket 9332. It was fixed in Boost 1.56.0.

Example 20.1 Recursive containers with Boost.Container

```
#include <boost/container/vector.hpp>

using namespace boost::container;

struct animal
{
  vector<animal> children;
};

int main()
{
  animal parent, child1, child2;
  parent.children.push_back(child1);
  parent.children.push_back(child2);
}
```

The class `animal` has a member variable **`children`** of type boost::container::vector<animal>. `boost::container::vector` is defined in the header file `boost/container/vector.hpp`. Thus, the type of the member variable **children** is based on the class animal, which defines the variable **children**. At this point, animal hasn't been defined completely. While the standard doesn't require containers from the standard library to support incomplete types, recursive containers are explicitly supported by Boost.Container. Whether containers defined by the standard library can be used recursively is implementation dependent.

Example 20.2 Using boost::container::stable_vector

```
#include <boost/container/stable_vector.hpp>
#include <iostream>

using namespace boost::container;

int main()
{
  stable_vector<int> v(2, 1);
  int &i = v[1];
  v.erase(v.begin());
  std::cout << i << '\n';
}
```

Boost.Container provides containers in addition to the well-known containers from the standard library. Example 20.2 introduces the container boost::container::stable_vector, which behaves similarly to std::vector, except that if boost::container::stable_vector is changed, all iterators and references to existing elements remain valid. This is possible because elements aren't stored contiguously in boost::container::stable_vector. It is still possible to access elements with an index even though elements are not stored next to each other in memory.
Boost.Container guarantees that the reference **i** in Example 20.2 remains valid when the first element in the vector is erased. The example displays 1.
Please note that neither boost::container::stable_vector nor other containers from this library support C++11 initializer lists. In Example 20.2 **v** is initialized with two elements both set to 1.
boost::container::stable_vector is defined in boost/container/stable_vector.hpp.
Additional containers provided by Boost.Container are boost::container::flat_set, boost::container::flat_map, boost::container::slist, and boost::container::static_vector:

- boost::container::flat_set and boost::container::flat_map resemble std::set and std::map. However they are implemented as sorted vectors, not as a tree. This allows faster lookups and iterations, but inserting and removing elements is more expensive.

  These two containers are defined in the header files boost/container/flat_set.hpp and boost/container/flat_map.hpp.

- boost::container::slist is a singly linked list. It is similar to std::forward_list, which was added to the standard library with C++11. boost::container::slist provides a member function size(), which is missing in std::forward_list.

  boost::container::slist is defined in boost/container/slist.hpp.

- boost::container::static_vector stores elements like std::array directly in the container. Like std::array, the container has a constant capacity, though the capacity doesn't say anything about the number of elements. The member functions push_back(), pop_back(), insert(), and erase() are available to insert or remove elements. In this regard, boost::container::static_vector is similar to std::vector. The member function size() returns the number of currently stored elements in the container.

  The capacity is constant, but can be changed with resize(). push_back() doesn't change the capacity. You may add an element with push_back() only if the capacity is greater than the number of currently stored elements. Otherwise, push_back() throws an exception of type std::bad_alloc.

  boost::container::static_vector is defined in boost/container/static_vector.hpp.

Part IV

Data Structures

Data structures are similar to containers since they can store one or multiple elements. However, they differ from containers because they don't support operations containers usually support. For example, it isn't possible, with the data structures introduced in this part, to access all elements in a single iteration.

- Boost.Optional makes it easy to mark optional return values. Objects created with Boost.Optional are either empty or contain a single element. With Boost.Optional, you don't need to use special values like a null pointer or -1 to indicate that a function might not have a return value.
- Boost.Tuple provides `boost::tuple`, a class that has been part of the standard library since C++11.
- Boost.Any and Boost.Variant let you create variables that can store values of different types. Boost.Any supports any arbitrary type, and Boost.Variant lets you pass the types that need to be supported as template parameters.
- Boost.PropertyTree provides a tree-like data structure. This library is typically used to help manage configuration data. The data can also be written to and loaded from a file in formats such as JSON.
- Boost.DynamicBitset provides a class that resembles `std::bitset` but is configured at runtime.
- Boost.Tribool provides a data type similar to bool that supports three states.
- Boost.CompressedPair defines the class `boost::compressed_pair`, which can replace `std::pair`. This class supports the so-called empty base class optimization.

Chapter 21

Boost.Optional

The library Boost.Optional provides the class `boost::optional`, which can be used for optional return values. These are return values from functions that may not always return a result. Example 21.1 illustrates how optional return values are usually implemented without Boost.Optional.

Example 21.1 Special values to denote optional return values

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

int get_even_random_number()
{
  int i = std::rand();
  return (i % 2 == 0) ? i : -1;
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  int i = get_even_random_number();
  if (i != -1)
    std::cout << std::sqrt(static_cast<float>(i)) << '\n';
}
```

Example 21.1 uses the function `get_even_random_number()`, which should return an even random number. It does this in a rather naive fashion by calling the function `std::rand()` from the standard library. If `std::rand()` generates an even random number, that number is returned by `get_even_random_number()`. If the generated random number is odd, -1 is returned.
In this example, -1 means that no even random number could be generated. Thus, `get_even_random_number()` can't guarantee that an even random number is returned. The return value is optional.
Many functions use special values like -1 to denote that no result can be returned. For example, the member function `find()` of the class `std::string` returns the special value **`std::string::npos`** if a substring can't be found. Functions whose return value is a pointer often return 0 to indicate that no result exists.
Boost.Optional provides `boost::optional`, which makes it possible to clearly mark optional return values.

Example 21.2 Optional return values with `boost::optional`

```
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using boost::optional;

optional<int> get_even_random_number()
{
  int i = std::rand();
  return (i % 2 == 0) ? i : optional<int>{};
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  optional<int> i = get_even_random_number();
  if (i)
    std::cout << std::sqrt(static_cast<float>(*i)) << '\n';
}
```

In Example 21.2 the return value of `get_even_random_number()` has a new type, boost::optional<int>. `boost::optional` is a template that must be instantiated with the actual type of the return value. `boost/optional.hpp` must be included for `boost::optional`.
If `get_even_random_number()` generates an even random number, the value is returned directly, automatically wrapped in an object of type boost::optional<int>, because `boost::optional` provides a non-exclusive constructor. If `get_even_random_number()` does not generate an even random number, an empty object of type boost::optional<int> is returned. The return value is created with a call to the default constructor.
`main()` checks whether **`i`** is empty. If it isn't empty, the number stored in **`i`** is accessed with `operator*`. `boost::optional` appears to work like a pointer. However, you should not think of `boost::optional` as a pointer because, for example, values in `boost::optional` are copied by the copy constructor while a pointer does not copy the value it points to.

Example 21.3 Other useful member functions of `boost::optional`

```
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using boost::optional;

optional<int> get_even_random_number()
{
  int i = std::rand();
  return optional<int>{i % 2 == 0, i};
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  optional<int> i = get_even_random_number();
  if (i.is_initialized())
    std::cout << std::sqrt(static_cast<float>(i.get())) << '\n';
}
```

Example 21.3 introduces other useful member functions of `boost::optional`. This class provides a special constructor that takes a condition as the first parameter. If the condition is true, an object of type `boost::optional` is initialized with the second parameter. If the condition is false, an empty object of type `boost::optional` is created. Example 21.3 uses this constructor in the function `get_even_random_number()`.
With `is_initialized()` you can check whether an object of type `boost::optional` is not empty. Boost.Optional speaks about initialized and uninitialized objects – hence, the name of the member function `is_initialized()`. The member function `get()` is equivalent to `operator*`.
Boost.Optional provides free-standing helper functions such as `boost::make_optional()` and `boost::get_optional_value_or()` (see Example 21.4). `boost::make_optional()` can be called to create an object of type `boost::optional`. If you want a default value to be returned when `boost::optional` is empty, you can call `boost::get_optional_value_or()`.
The function `boost::get_optional_value_or()` is also provided as a member function of `boost::optional`. It is called `get_value_or()`.
Along with `boost/optional/optional_io.hpp`, Boost.Optional provides a header file with overloaded stream operators, which let you write objects of type `boost::optional` to, for example, standard output.

Example 21.4 Various helper functions of Boost.Optional

```
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

using namespace boost;

optional<int> get_even_random_number()
{
  int i = std::rand();
  return make_optional(i % 2 == 0, i);
}

int main()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  optional<int> i = get_even_random_number();
  double d = get_optional_value_or(i, 0);
  std::cout << std::sqrt(d) << '\n';
}
```

Chapter 22

Boost.Tuple

The library Boost.Tuple provides a class called `boost::tuple`, which is a generalized version of `std::pair`. While `std::pair` can only store exactly two values, `boost::tuple` lets you choose how many values to store. The standard library has provided the class `std::tuple` since C++11. If you work with a development environment supporting C++11, you can ignore Boost.Tuple because `boost::tuple` and `std::tuple` are identical.

Example 22.1 `boost::tuple` replacing `std::pair`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int> animal;
  animal a{"cat", 4};
  std::cout << a << '\n';
}
```

To use `boost::tuple`, include the header file `boost/tuple/tuple.hpp`. To use tuples with streams, include the header file `boost/tuple/tuple_io.hpp`. Boost.Tuple doesn't provide a master header file that automatically includes all others.

`boost::tuple` is used in the same way `std::pair` is. In Example 22.1, a tuple containing one value of type `std::string` and one value of type int is created. This type is called animal, and it stores the name and the number of legs of an animal.

While the definition of type animal could have used `std::pair`, objects of type `boost::tuple` can be written to a stream. To do this you must include the header file `boost/tuple/tuple_io.hpp`, which provides the required operators. Example 22.1 displays `(cat 4)`.

Example 22.2 `boost::tuple` as the better `std::pair`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a{"cat", 4, true};
  std::cout << std::boolalpha << a << '\n';
}
```

Example 22.2 stores a name, the number of legs, and a flag that indicates whether the animal has a tail. All three values are placed in a tuple. When executed, this program displays `(cat 4 true)`.

You can create a tuple using the helper function `boost::make_tuple()`, which works like the helper function `std::make_pair()` for `std::pair` (see Example 22.3).

Example 22.3 Creating tuples with `boost::make_tuple()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <iostream>

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << boost::make_tuple("cat", 4, true) << '\n';
}
```

A tuple can also contain references, as shown in Example 22.4.
The values 4 and `true` are passed by value and, thus, are stored directly inside the tuple, However, the first element is a reference to the string **`s`**. The function `boost::ref()` from Boost.Ref is used to create the reference. To create a constant reference, use `boost::cref()`.

Example 22.4 Tuples with references

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <boost/ref.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "cat";
  std::cout.setf(std::ios::boolalpha);
  std::cout << boost::make_tuple(boost::ref(s), 4, true) << '\n';
}
```

Usually, you can use `std::ref()` from the C++11 standard library instead of `boost::ref()`. However, Example 22.4 uses `boost::ref()` because only Boost.Ref provides an operator to write to standard output.
`std::pair` uses the member variables **`first`** and **`second`** to provide access. Because a tuple does not have a fixed number of elements, access must be handled differently.

Example 22.5 Reading elements of a tuple

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a = boost::make_tuple("cat", 4, true);
  std::cout << a.get<0>() << '\n';
  std::cout << boost::get<0>(a) << '\n';
}
```

There are two ways to access values in a tuple. You can call the member function `get()`, or you can pass the tuple to the free-standing function `boost::get()`. In both cases, the index of the corresponding element in the tuple must be provided as a template parameter. Example 22.5 accesses the first element of the tuple **`a`** in both cases and, thus, displays `cat` twice.
Specifying an invalid index results in a compiler error because index validity is checked at compile time.
The member function `get()` and the free-standing function `boost::get()` both return a reference that allows you to change a value inside a tuple.

Example 22.6 Writing elements of a tuple

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>
```

```
int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a = boost::make_tuple("cat", 4, true);
  a.get<0>() = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

Example 22.6 modifies the animal's name and, thus, displays `(dog 4 true)`.

Boost.Tuple also defines comparison operators. To compare tuples, include the header file `boost/tuple/tuple_comparison.hpp`.

Example 22.7 Comparing tuples

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string, int, bool> animal;
  animal a1 = boost::make_tuple("cat", 4, true);
  animal a2 = boost::make_tuple("shark", 0, true);
  std::cout << std::boolalpha << (a1 != a2) << '\n';
}
```

Example 22.7 displays `true` because the tuples **`a1`** and **`a2`** are different.

The header file `boost/tuple/tuple_comparison.hpp` also contains definitions for other comparison operators such as greater-than, which performs a lexicographical comparison.

Boost.Tuple supports a specific form of tuples called *tier*. Tiers are tuples whose elements are all reference types. They can be constructed with the function `boost::tie()`.

Example 22.8 Creating a tier with `boost::tie()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, int&, bool&> animal;
  std::string name = "cat";
  int legs = 4;
  bool tail = true;
  animal a = boost::tie(name, legs, tail);
  name = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

Example 22.8 creates a tier **`a`**, which consists of references to the variables **`name`**, **`legs`**, and **`tail`**. When the variable **`name`** is modified, the tier is modified at the same time.

Example 22.8 could have also been written using `boost::make_tuple()` and `boost::ref()` (see Example 22.9).

Example 22.9 Creating a tier without `boost::tie()`

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_io.hpp>
#include <string>
#include <iostream>

int main()
{
  typedef boost::tuple<std::string&, int&, bool&> animal;
  std::string name = "cat";
  int legs = 4;
  bool tail = true;
  animal a = boost::make_tuple(boost::ref(name), boost::ref(legs),
    boost::ref(tail));
  name = "dog";
  std::cout << std::boolalpha << a << '\n';
}
```

`boost::tie()` shortens the syntax. This function can also be used to unpack tuples. In Example 22.10, the individual values of the tuple, returned by a function, are instantly stored in variables.

Example 22.10 Unpacking return values of a function from a tuple

```
#include <boost/tuple/tuple.hpp>
#include <string>
#include <iostream>

boost::tuple<std::string, int> new_cat()
{
  return boost::make_tuple("cat", 4);
}

int main()
{
  std::string name;
  int legs;
  boost::tie(name, legs) = new_cat();
  std::cout << name << ", " << legs << '\n';
}
```

`boost::tie()` stores the string “cat” and the number 4, both of which are returned as a tuple from `new_cat()`, in the variables **`name`** and **`legs`**.

Chapter 23

Boost.Any

Strongly typed languages, such as C++, require that each variable have a specific type that defines what kind of information it can store. Other languages, such as JavaScript, allow developers to store any kind of information in a variable. For example, in JavaScript a single variable can contain a string, then a number, and afterwards a boolean value.

Boost.Any provides the class `boost::any` which, like JavaScript variables, can store arbitrary types of information.

Example 23.1 Using `boost::any`

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

int main()
{
  boost::any a = 1;
  a = 3.14;
  a = true;
}
```

To use `boost::any`, include the header file `boost/any.hpp`. Objects of type `boost::any` can then be created to store arbitrary information. In Example 23.1, **a** stores an int, then a double, then a bool.

Variables of type `boost::any` are not completely unlimited in what they can store; there are some preconditions, albeit minimal ones. Any value stored in a variable of type `boost::any` must be copy-constructible. Thus, it is not possible to store a C array, since C arrays aren't copy-constructible.

To store a string, and not just a pointer to a C string, use `std::string` (see Example 23.2).

Example 23.2 Storing a string in `boost::any`

```
#include <boost/any.hpp>
#include <string>

int main()
{
  boost::any a = std::string{"Boost"};
}
```

To access the value of `boost::any` variables, use the cast operator `boost::any_cast` (see Example 23.3).

Example 23.3 Accessing values with `boost::any_cast`

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  boost::any a = 1;
  std::cout << boost::any_cast<int>(a) << '\n';
  a = 3.14;
  std::cout << boost::any_cast<double>(a) << '\n';
  a = true;
  std::cout << std::boolalpha << boost::any_cast<bool>(a) << '\n';
}
```

By passing the appropriate type as a template parameter to `boost::any_cast`, the value of the variable is converted. If an invalid type is specified, an exception of type `boost::bad_any_cast` will be thrown.

Example 23.4 `boost::bad_any_cast` in case of an error

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  try
  {
    boost::any a = 1;
    std::cout << boost::any_cast<float>(a) << '\n';
  }
  catch (boost::bad_any_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Example 23.4 throws an exception because the template parameter of type float does not match the type int stored in **a**. The program would also throw an exception if short or long were used as the template parameter.
Because `boost::bad_any_cast` is derived from `std::bad_cast`, `catch` handlers can catch exceptions of this type, too.
To check whether or not a variable of type `boost::any` contains information, use the member function `empty()`. To check the type of the stored information, use the member function `type()`.

Example 23.5 Checking type of currently stored value

```
#include <boost/any.hpp>
#include <typeinfo>
#include <iostream>

int main()
{
  boost::any a = 1;
  if (!a.empty())
  {
    const std::type_info &ti = a.type();
    std::cout << ti.name() << '\n';
  }
}
```

Example 23.5 uses both `empty()` and `type()`. While `empty()` returns a boolean value, the return value of `type()` is of type `std::type_info`, which is defined in the header file `typeinfo`.
Example 23.6 shows how to obtain a pointer to the value stored in a `boost::any` variable using `boost::any_cast`.

Example 23.6 Accessing values through a pointer

```
#include <boost/any.hpp>
#include <iostream>

int main()
{
  boost::any a = 1;
  int *i = boost::any_cast<int>(&a);
  std::cout << *i << '\n';
}
```

You simply pass a pointer to a `boost::any` variable to `boost::any_cast`; the template parameter remains unchanged.

Chapter 24

Boost.Variant

Boost.Variant provides a class called boost::variant that resembles union. You can store values of different types in a boost::variant variable. At any point only one value can be stored. When a new value is assigned, the old value is overwritten. However, the new value may have a different type from the old value. The only requirement is that the types must have been passed as template parameters to boost::variant so they are known to the boost::variant variable.

boost::variant supports any type. For example, it is possible to store a std::string in a boost::variant variable – something that wasn't possible with union before C++11. With C++11, the requirements for union were relaxed. Now a union can contain a std::string. Because a std::string must be initialized with placement new and has to be destroyed by an explicit call to the destructor, it can still make sense to use boost::variant, even in a C++11 development environment.

Example 24.1 Using boost::variant

```
#include <boost/variant.hpp>
#include <string>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  v = 'A';
  v = "Boost";
}
```

boost::variant is defined in boost/variant.hpp. Because boost::variant is a template, at least one parameter must be specified. One or more template parameters specify the supported types. In Example 24.1, **v** can store values of type double, char, or std::string. However, if you tried to assign a value of type int to **v**, the resulting code would not compile.

Example 24.2 Accessing values in boost::variant with boost::get()

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << boost::get<double>(v) << '\n';
  v = 'A';
  std::cout << boost::get<char>(v) << '\n';
  v = "Boost";
  std::cout << boost::get<std::string>(v) << '\n';
}
```

To display the stored values of **v**, use the free-standing function boost::get() (see Example 24.2).

boost::get() expects one of the valid types for the corresponding variable as a template parameter. Specifying an invalid type will result in a run-time error because validation of types does not take place at compile time.

Variables of type `boost::variant` can be written to streams such as the standard output stream, bypassing the hazard of run-time errors (see Example 24.3).

Example 24.3 Direct output of `boost::variant` on a stream

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  std::cout << v << '\n';
  v = 'A';
  std::cout << v << '\n';
  v = "Boost";
  std::cout << v << '\n';
}
```

Example 24.4 Using a visitor for `boost::variant`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

struct output : public boost::static_visitor<>
{
  void operator()(double d) const { std::cout << d << '\n'; }
  void operator()(char c) const { std::cout << c << '\n'; }
  void operator()(std::string s) const { std::cout << s << '\n'; }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output{}, v);
  v = 'A';
  boost::apply_visitor(output{}, v);
  v = "Boost";
  boost::apply_visitor(output{}, v);
}
```

For type-safe access, Boost.Variant provides a function called `boost::apply_visitor()`.
As its first parameter, `boost::apply_visitor()` expects an object of a class derived from `boost::static_visitor`. This class must overload `operator()` for every type used by the `boost::variant` variable it acts on. Consequently, the operator is overloaded three times in Example 24.4 because **v** supports the types double, char, and `std::string`.
`boost::static_visitor` is a template. The type of the return value of `operator()` must be specified as a template parameter. If the operator does not have a return value, a template parameter is not required, as seen in the example.
The second parameter passed to `boost::apply_visitor()` is a `boost::variant` variable.
`boost::apply_visitor()` automatically calls the `operator()` for the first parameter that matches the type of the value currently stored in the second parameter. This means that the sample program uses different overloaded operators every time `boost::apply_visitor()` is invoked – first the one for double, followed by the one for char, and finally the one for `std::string`.
The advantage of `boost::apply_visitor()` is not only that the correct operator is called automatically. In addition, `boost::apply_visitor()` ensures that overloaded operators have been provided for every type supported by `boost::variant` variables. If one of the three overloaded operators had not been defined, the code could not be compiled.

Example 24.5 Using a visitor with a function template for `boost::variant`

```
#include <boost/variant.hpp>
#include <string>
#include <iostream>

struct output : public boost::static_visitor<>
{
  template <typename T>
  void operator()(T t) const { std::cout << t << '\n'; }
};

int main()
{
  boost::variant<double, char, std::string> v;
  v = 3.14;
  boost::apply_visitor(output{}, v);
  v = 'A';
  boost::apply_visitor(output{}, v);
  v = "Boost";
  boost::apply_visitor(output{}, v);
}
```

If overloaded operators are equivalent in functionality, the code can be simplified by using a template (see Example 24.5).
Because `boost::apply_visitor()` ensures code correctness at compile time, it should be preferred over `boost::get()`.

Chapter 25

Boost.PropertyTree

With the class boost::property_tree::ptree, Boost.PropertyTree provides a tree structure to store key/-value pairs. Tree structure means that a trunk exists with numerous branches that have numerous twigs. A file system is a good example of a tree structure. File systems have a root directory with subdirectories that themselves can have subdirectories and so on.
To use boost::property_tree::ptree, include the header file boost/property_tree/ptree.hpp. This is a master header file, so no other header files need to be included for Boost.PropertyTree.

Example 25.1 Accessing data in boost::property_tree::ptree

```
#include <boost/property_tree/ptree.hpp>
#include <iostream>

using boost::property_tree::ptree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");

  ptree &c = pt.get_child("C:");
  ptree &windows = c.get_child("Windows");
  ptree &system = windows.get_child("System");
  std::cout << system.get_value<std::string>() << '\n';
}
```

Example 25.1 uses boost::property_tree::ptree to store a path to a directory. This is done with a call to put(). This member function expects two parameters because boost::property_tree::ptree is a tree structure that saves key/value pairs. The tree doesn't just consist of branches and twigs, a value must be assigned to each branch and twig. In Example 25.1 the value is “20 files”.
The first parameter passed to put() is more interesting. It is a path to a directory. However, it doesn't use the backlash, which is the common path separator on Windows. It uses the dot.
You need to use the dot because it's the separator Boost.PropertyTree expects for keys. The parameter “C:.Windows.System” tells **pt** to create a branch called C: with a branch called Windows that has another branch called System. The dot creates the nested structure of branches. If “C:\Windows\System” had been passed as the parameter, **pt** would only have one branch called C:\Windows\System.
After the call to put(), **pt** is accessed to read the stored value “20 files” and write it to standard output. This is done by jumping from branch to branch – or directory to directory.
To access a subbranch, you call get_child(), which returns a reference to an object of the same type get_child() was called on. In Example 25.1, this is a reference to boost::property_tree::ptree. Because every branch can have subbranches, and because there is no structural difference between higher and lower branches, the same type is used.
The third call to get_child() retrieves the boost::property_tree::ptree, which represents the directory System. get_value() is called to read the value that was stored at the beginning of the example with put(). Please note that get_value() is a function template. You pass the type of the return value as a template parameter. That way get_value() can do an automatic type conversion.

Example 25.2 Accessing data in `basic_ptree<std::string, int>`

```
#include <boost/property_tree/ptree.hpp>
#include <utility>
#include <iostream>

int main()
{
  typedef boost::property_tree::basic_ptree<std::string, int> ptree;
  ptree pt;
  pt.put(ptree::path_type{"C:\\Windows\\System", '\\'}, 20);
  pt.put(ptree::path_type{"C:\\Windows\\Cursors", '\\'}, 50);

  ptree &windows = pt.get_child(ptree::path_type{"C:\\Windows", '\\'});
  int files = 0;
  for (const std::pair<std::string, ptree> &p : windows)
    files += p.second.get_value<int>();
  std::cout << files << '\n';
}
```

There are two changes in Example 25.2 compared with Example 25.1. These changes are to save paths to directories and the number of files in directories more easily. First, paths use a backslash as the separator when passed to `put()`. Secondly, the number of files is stored as an int.

By default, Boost.PropertyTree uses a dot as the separator for keys. If you need to use another character, such as the backslash, as the separator, you don't pass the key as a string to `put()`. Instead you wrap it in an object of type `boost::property_tree::ptree::path_type`. The constructor of this class, which depends on `boost::property_tree::ptree`, takes the key as its first parameter and the separator character as its second parameter. That way, you can use a path such as C:\Windows\System, as shown in Example 25.2, without having to replace backslashes with dots.

`boost::property_tree::ptree` is based on the class template `boost::property_tree::basic_ptree`. Because keys and values are often strings, `boost::property_tree::ptree` is predefined. However, you can use `boost::property_tree::basic_ptree` with different types for keys and values. The tree in Example 25.2 uses an int to store the number of files in a directory rather than a string.

`boost::property_tree::ptree` provides the member functions `begin()` and `end()`. However, `boost::property_tree::ptree` only lets you iterate over the branches in one level. Example 25.2 iterates over the subdirectories of C:\Windows. You can't get an iterator to iterate over all branches in all levels.

The `for` loop in Example 25.2 reads the number of files in all subdirectories of C:\Windows to calculate a total. As a result, the example displays `70`. The example doesn't access objects of type `ptree` directly. Instead it iterates over elements of type `std::pair<std::string, ptree>`. **`first`** contains the key of the current branch. That is System and Cursors in Example 25.2. **`second`** provides access to an object of type `ptree`, which represents the possible subdirectories. In the example, only the values assigned to System and Cursors are read. As in Example 25.1, the member function `get_value()` is called.

`boost::property_tree::ptree` only stores the value of the current branch, not its key. You can get the value with `get_value()`, but there is no member function to get the key. The key is stored in `boost::property_tree::ptree` one level up. This also explains why the `for` loop iterates over elements of type `std::pair<std::string, ptree>`.

Example 25.3 uses with `boost::property_tree::iptree` another predefined tree from Boost.PropertyTree. In general, this type behaves like `boost::property_tree::ptree`. The only difference is that `boost::property_tree::iptree` doesn't distinguish between lower and upper case. For example, a value stored with the key C:\Windows\System can be read with c:\windows\system.

Example 25.3 Accessing data with a translator

```
#include <boost/property_tree/ptree.hpp>
#include <boost/optional.hpp>
#include <iostream>
#include <cstdlib>

struct string_to_int_translator
{
  typedef std::string internal_type;
  typedef int external_type;
  boost::optional<int> get_value(const std::string &s)
  {
    char *c;
    long l = std::strtol(s.c_str(), &c, 10);
    return boost::make_optional(c != s.c_str(), static_cast<int>(l));
  }
};

int main()
{
  typedef boost::property_tree::iptree ptree;
  ptree pt;
  pt.put(ptree::path_type{"C:\\Windows\\System", '\\'}, "20 files");
  pt.put(ptree::path_type{"C:\\Windows\\Cursors", '\\'}, "50 files");

  string_to_int_translator tr;
  int files =
    pt.get<int>(ptree::path_type{"c:\\windows\\system", '\\'}, tr) +
    pt.get<int>(ptree::path_type{"c:\\windows\\cursors", '\\'}, tr);
  std::cout << files << '\n';
}
```

Unlike Example 25.1, `get_child()` isn't called multiple times to access subbranches. Just as `put()` can be used to store a value in a subbranch directly, a value from a subbranch can be read with `get()`. The key is defined the same way – for example using `boost::property_tree::iptree::path_type`.
Like `get_value()`, `get()` is a function template. You have to pass the type of the return value as a template parameter. Boost.PropertyTree does an automatic type conversion.
To convert types, Boost.PropertyTree uses *translators*. The library provides a few translators out of the box that are based on streams and can convert types automatically.
Example 25.3 defines the translator `string_to_int_translator`, which converts a value of type `std::string` to int. The translator is passed as an additional parameter to `get()`. Because the translator is just used to read, it only defines one member function, `get_value()`. If you want to use the translator for writing, too, then you would need to define a member function `put_value()` and then pass the translator as an additional parameter to `put()`.
`get_value()` returns a value of the type that is used in **`pt`**. However, because a type conversion doesn't always succeed, `boost::optional` is used. If a value is stored in Example 25.3 that can't be converted to an int with `std::strtol()`, an empty object of type `boost::optional` will be returned.
Please note that a translator must also define the two types internal_type and external_type. If you need to convert types when storing data, define `put_value()` similar to `get_value()`.
If you modify Example 25.3 to store the value “20” instead of value “20 files,” `get_value()` can be called without passing a translator. The translators provided by Boost.PropertyTree can convert from `std::string` to int. However, the type conversion only succeeds when the entire string can be converted. The string must not contain any letters. Because `std::strtol()` can do a type conversion as long as the string starts with digits, the more liberal translator `string_to_int_translator` is used in Example 25.3.

Example 25.4 Various member functions of `boost::property_tree::ptree`

```
#include <boost/property_tree/ptree.hpp>
#include <utility>
#include <iostream>

using boost::property_tree::ptree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");

  boost::optional<std::string> c = pt.get_optional<std::string>("C:");
  std::cout << std::boolalpha << c.is_initialized() << '\n';

  pt.put_child("D:.Program Files", ptree{"50 files"});
  pt.add_child("D:.Program Files", ptree{"60 files"});

  ptree d = pt.get_child("D:");
  for (const std::pair<std::string, ptree> &p : d)
    std::cout << p.second.get_value<std::string>() << '\n';

  boost::optional<ptree&> e = pt.get_child_optional("E:");
  std::cout << e.is_initialized() << '\n';
}
```

You can call the member function `get_optional()` if you want to read the value of a key, but you aren't sure if the key exists. `get_optional()` returns the value in an object of type `boost::optional`. The object is empty if the key wasn't found. Otherwise, `get_optional()` works the same as `get()`.
It might seem like `put_child()` and `add_child()` are the same as `put()`. The difference is that `put()` creates only a key/value pair while `put_child()` and `add_child()` insert an entire subtree. Note that an object of type `boost::property_tree::ptree` is passed as the second parameter to `put_child()` and `add_child()`.
The difference between `put_child()` and `add_child()` is that `put_child()` accesses a key if that key already exists, while `add_child()` always inserts a new key into the tree. That's why the tree in Example 25.4 has two keys called "D:.Program Files". Depending on the use case, this can be confusing. If a tree represents a file system, there shouldn't be two identical paths. You have to avoid inserting identical keys if you don't want duplicates in a tree.
Example 25.4 displays the value of the keys below "D:" in the `for` loop. The example writes `50 files` and `60 files` to standard output, which proves there are two identical keys called "D:.Program Files".
The last member function introduced in Example 25.4 is `get_child_optional()`. This function is used like `get_child()`. `get_child_optional()` returns an object of type `boost::optional`. You call `boost::optional` if you aren't sure whether a key exists.

Example 25.5 Serializing a `boost::property_tree::ptree` in the JSON format

```
#include <boost/property_tree/ptree.hpp>
#include <boost/property_tree/json_parser.hpp>
#include <iostream>

using namespace boost::property_tree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");
  pt.put("C:.Windows.Cursors", "50 files");

  json_parser::write_json("file.json", pt);

  ptree pt2;
  json_parser::read_json("file.json", pt2);

  std::cout << std::boolalpha << (pt == pt2) << '\n';
}
```

Boost.PropertyTree does more than just provide structures to manage data in memory. As can be seen in Example 25.5, the library also provides functions to save a `boost::property_tree::ptree` in a file and load it from a file.
The header file `boost/property_tree/json_parser.hpp` provides access to the functions `boost::property_tree::json_parser::write_json()` and `boost::property_tree::json_parser::read_json()`. These functions make it possible to save and load a `boost::property_tree::ptree` serialized in the JSON format. That way you can support configuration files in the JSON format.
If you want to call functions that store a `boost::property_tree::ptree` in a file or load it from a file, you must include header files such as `boost/property_tree/json_parser.hpp`. It isn't sufficient to only include `boost/property_tree/ptree.hpp`.
In addition to the functions `boost::property_tree::json_parser::write_json()` and `boost::property_tree::json_parser::read_json()`, Boost.PropertyTree provides functions for additional data formats.

You use `boost::property_tree::ini_parser::write_ini()` and `boost::property_tree::ini_parser::read_ini()` from `boost/property_tree/ini_parser.hpp` to support INI-files. With `boost::property_tree::xml_parser::write_xml()` and `boost::property_tree::xml_parser::read_xml()` from `boost/property_tree/xml_parser.hpp`, data can be loaded and stored in XML format. With `boost::property_tree::info_parser::write_info()` and `boost::property_tree::info_parser::read_info()` from `boost/property_tree/info_parser.hpp`, you can access another format that was developed and optimized to serialize trees from Boost.PropertyTree.
None of the supported formats guarantees that a `boost::property_tree::ptree` will look the same after it has been saved and reloaded. For example, the JSON format can lose type information because `boost::property_tree::ptree` can't distinguish between `true` and "true". The type is always the same. Even if the various functions make it easy to save and load a `boost::property_tree::ptree`, don't forget that Boost.PropertyTree doesn't support the formats completely. The main focus of the library is on the structure `boost::property_tree::ptree` and not on supporting various data formats.

Chapter 26

Boost.DynamicBitset

The library Boost.DynamicBitset provides the class `boost::dynamic_bitset`, which is used like `std::bitset`. The difference is that the number of bits for `std::bitset` must be specified at compile time, whereas the number of bits for `boost::dynamic_bitset` is specified at run time.
To use `boost::dynamic_bitset`, include the header file `boost/dynamic_bitset.hpp`.

Example 26.1 Using `boost::dynamic_bitset`

```
#include <boost/dynamic_bitset.hpp>
#include <iostream>

int main()
{
  boost::dynamic_bitset<> db{3, 4};

  db.push_back(true);

  std::cout.setf(std::ios::boolalpha);
  std::cout << db.size() << '\n';
  std::cout << db.count() << '\n';
  std::cout << db.any() << '\n';
  std::cout << db.none() << '\n';

  std::cout << db[0].flip() << '\n';
  std::cout << ~db[3] << '\n';
  std::cout << db << '\n';
}
```

`boost::dynamic_bitset` is a template that requires no template parameters when instantiated; default types are used in that case. More important are the parameters passed to the constructor. In Example 26.1, the constructor creates **db** with 3 bits. The second parameter initializes the bits; in this case, the number 4 initializes the most significant bit – the bit on the very left.
The number of bits inside an object of type `boost::dynamic_bitset` can be changed at any time. The member function `push_back()` adds another bit, which will become the most significant bit. Calling `push_back()` in Example 26.1 causes **db** to contain 4 bits, of which the two most significant bits are set. Therefore, **db** stores the number 12.
You can decrease the number of bits by calling the member function `resize()`. Depending on the parameter passed to `resize()`, bits will either be added or removed.
`boost::dynamic_bitset` provides member functions to query data and access individual bits. The member functions `size()` and `count()` return the number of bits and the number of bits currently set, respectively. `any()` returns `true` if at least one bit is set, and `none()` returns `true` if no bit is set.
To access individual bits, use array syntax. A reference to an internal class is returned that represents the corresponding bit and provides member functions to manipulate it. For example, the member function `flip()` toggles the bit. Bitwise operators such as `operator~` are available as well. Overall, the class `boost::dynamic_bitset` offers the same bit manipulation functionality as `std::bitset`.
Like `std::bitset`, `boost::dynamic_bitset` does not support iterators.

Chapter 27

Boost.Tribool

The library Boost.Tribool provides the class `boost::logic::tribool`, which is similar to bool. However, while bool can distinguish two states, `boost::logic::tribool` handles three.
To use `boost::logic::tribool`, include the header file `boost/logic/tribool.hpp`.
A variable of type `boost::logic::tribool` can be set to `true`, `false`, or `indeterminate`. The default constructor initializes the variable to `false`. That's why Example 27.1 writes `false` first.

Example 27.1 Three states of `boost::logic::tribool`

```
#include <boost/logic/tribool.hpp>
#include <iostream>

using namespace boost::logic;

int main()
{
  tribool b;
  std::cout << std::boolalpha << b << '\n';

  b = true;
  b = false;
  b = indeterminate;
  if (b)
    ;
  else if (!b)
    ;
  else
    std::cout << "indeterminate\n";
}
```

The `if` statement in Example 27.1 illustrates how to evaluate **b** correctly. You have to check for `true` and `false` explicitly. If the variable is set to `indeterminate`, as in the example, the `else` block will be executed.
Boost.Tribool also provides the function `boost::logic::indeterminate()`. If you pass a variable of type `boost::logic::tribool` that is set to `indeterminate`, this function will return `true`. If the variable is set to `true` or `false`, it will return `false`.

Example 27.2 Logical operators with `boost::logic::tribool`

```
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>
#include <iostream>

using namespace boost::logic;

int main()
{
  std::cout.setf(std::ios::boolalpha);

  tribool b1 = true;
  std::cout << (b1 || indeterminate) << '\n';
  std::cout << (b1 && indeterminate) << '\n';

  tribool b2 = false;
  std::cout << (b2 || indeterminate) << '\n';
  std::cout << (b2 && indeterminate) << '\n';

  tribool b3 = indeterminate;
  std::cout << (b3 || b3) << '\n';
  std::cout << (b3 && b3) << '\n';
}
```

You can use logical operators with variables of type `boost::logic::tribool`, just as you can with variables of type bool. In fact, this is the only way to process variables of type `boost::logic::tribool` because the class doesn't provide any member functions.

Example 27.2 returns `true` for `b1 || indeterminate`, `false` for `b2 && indeterminate`, and `indeterminate` in all other cases. If you look at the operations and their results, you will notice that `boost::logic::tribool` behaves as one would expect intuitively. The documentation on Boost.Tribool also contains tables that show which operations lead to which results.

Example 27.2 also illustrates how the values `true`, `false`, and `indeterminate` are written to standard output with variables of type `boost::logic::tribool`. The header file `boost/logic/tribool_io.hpp` must be included and the flag **`std::ios::boolalpha`** must be set for standard output.

Boost.Tribool also provides the macro `BOOST_TRIBOOL_THIRD_STATE`, which lets you substitute another value for `indeterminate`. For example, you could use `dontknow` instead of `indeterminate`.

Chapter 28

Boost.CompressedPair

Boost.CompressedPair provides `boost::compressed_pair`, a class that behaves like `std::pair`. However, if one or both template parameters are empty classes, `boost::compressed_pair` consumes less memory. `boost::compressed_pair` uses a technique known as empty base class optimization.
To use `boost::compressed_pair`, include the header file `boost/compressed_pair.hpp`.

Example 28.1 Reduced memory requirements with `boost::compressed_pair`

```
#include <boost/compressed_pair.hpp>
#include <utility>
#include <iostream>

struct empty {};

int main()
{
  std::pair<int, empty> p;
  std::cout << sizeof(p) << '\n';

  boost::compressed_pair<int, empty> cp;
  std::cout << sizeof(cp) << '\n';
}
```

Example 28.1 illustrates this by using `boost::compressed_pair` for **cp** and `std::pair` for **p**. When compiled using Visual C++ 2013 and run on a 64-bit Windows 7 system, the example returns 4 for `sizeof(cp)` and 8 for `sizeof(p)`.
Please note that there is another difference between `boost::compressed_pair` and `std::pair`: the values stored in `boost::compressed_pair` are accessed through the member functions `first()` and `second()`. `std::pair` uses two identically named member variables instead.

Part V

Algorithms

The following libraries provide algorithms that complement the algorithms from the standard library.

- Boost.Algorithm collects and provides useful algorithms.
- Boost.Range also provides algorithms, but more important, it defines a new concept called *range*, which should make using algorithms easier.
- Boost.Graph is specialized for graphs and provides algorithms such as finding the shortest path between two points.

A few libraries that contain algorithms are introduced in other parts of the book. For example, you will find algorithms for strings in the library Boost.StringAlgorithms, which is introduced in Part II

Chapter 29

Boost.Algorithm

Boost.Algorithm provides algorithms that complement the algorithms from the standard library. Unlike Boost.Range, Boost.Algorithm doesn't introduce new concepts. The algorithms defined by Boost.Algorithm resemble the algorithms from the standard library.

Please note that there are numerous algorithms provided by other Boost libraries. For example, you will find algorithms to process strings in Boost.StringAlgorithms. The algorithms provided by Boost.Algorithm are not bound to particular classes, such as `std::string`. Like the algorithms from the standard library, they can be used with any container.

Example 29.1 Testing for exactly one value with `boost::algorithm::one_of_equal()`

```
#include <boost/algorithm/cxx11/one_of.hpp>
#include <array>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::array<int, 6> a{{0, 5, 2, 1, 4, 3}};
  auto predicate = [](int i){ return i == 4; };
  std::cout.setf(std::ios::boolalpha);
  std::cout << one_of(a.begin(), a.end(), predicate) << '\n';
  std::cout << one_of_equal(a.begin(), a.end(), 4) << '\n';
}
```

`boost::algorithm::one_of()` tests whether a condition is met exactly once. The condition to test is passed as a predicate. In Example 29.1 the call to `boost::algorithm::one_of()` returns `true` since the number 4 is stored exactly once in **a**.

To test elements in a container for equality, call `boost::algorithm::one_of_equal()`. You don't pass a predicate. Instead, you pass a value to compare to `boost::algorithm::one_of_equal()`. In Example 29.1 the call to `boost::algorithm::one_of_equal()` also returns `true`.

`boost::algorithm::one_of()` complements the algorithms `std::all_of()`, `std::any_of()`, and `std::none_of()`, which were added to the standard library with C++11. However, Boost.Algorithm provides the functions `boost::algorithm::all_of()`, `boost::algorithm::any_of()`, and `boost::algorithm::none_of()` for developers whose development environment doesn't support C++11. You will find these algorithms in the header files `boost/algorithm/cxx11/all_of.hpp`, `boost/algorithm/cxx11/any_of.hpp`, and `boost/algorithm/cxx11/none_of.hpp`.

Boost.Algorithm also defines the following functions: `boost::algorithm::all_of_equal()`, `boost::algorithm::any_of_equal()`, and `boost::algorithm::none_of_equal()`.

Boost.Algorithm provides more algorithms from the C++11 standard library. For example, you have access to `boost::algorithm::is_partitioned()`, `boost::algorithm::is_permutation()`, `boost::algorithm::copy_n()`, `boost::algorithm::find_if_not()` and `boost::algorithm::iota()`. These functions work like the identically named functions from the C++11 standard library and are provided for developers who don't use C++11. However, Boost.Algorithm provides a few function variants that could be useful for C++11 developers, too.

Example 29.2 More variants of C++11 algorithms

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/algorithm/cxx11/is_sorted.hpp>
#include <boost/algorithm/cxx11/copy_if.hpp>
#include <vector>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v;
  iota_n(std::back_inserter(v), 10, 5);
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_increasing(v) << '\n';
  std::ostream_iterator<int> out{std::cout, ","};
  copy_until(v, out, [](int i){ return i > 12; });
}
```

Boost.Algorithm provides the C++11 algorithm `boost::algorithm::iota()` in the header file `boost/algorithm/cxx11/iota.hpp`. This function generates sequentially increasing numbers. It expects two iterators for the beginning and end of a container. The elements in the container are then overwritten with sequentially increasing numbers.
Instead of `boost::algorithm::iota()`, Example 29.2 uses `boost::algorithm::iota_n()`. This function expects one iterator to write the numbers to. The number of numbers to generate is passed as a third parameter to `boost::algorithm::iota_n()`.
`boost::algorithm::is_increasing()` and `boost::algorithm::is_sorted()` are defined in the header file `boost/algorithm/cxx11/is_sorted.hpp`. `boost::algorithm::is_increasing()` has the same function as `boost::algorithm::is_sorted()`, but the function name expresses more clearly that the function checks that values are in increasing order. The header file also defines the related function `boost::algorithm::is_decreasing()`.
In Example 29.2, **v** is passed directly to `boost::algorithm::is_increasing()`. All functions provided by Boost.Algorithm have a variant that operates based on ranges. Containers can be passed directly to these functions.
`boost::algorithm::copy_until()` is defined in `boost/algorithm/cxx11/copy_if.hpp`. This is another variant of `std::copy()`. Boost.Algorithm also provides `boost::algorithm::copy_while()`.
Example 29.2 displays `true` as a result from `boost::algorithm::is_increasing()`, and `boost::algorithm::copy_until()` writes the numbers `10`, `11`, and `12` to standard output.

Example 29.3 C++14 algorithms from Boost.Algorithm

```
#include <boost/algorithm/cxx14/equal.hpp>
#include <boost/algorithm/cxx14/mismatch.hpp>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v{1, 2};
  std::vector<int> w{1, 2, 3};
  std::cout.setf(std::ios::boolalpha);
  std::cout << equal(v.begin(), v.end(), w.begin(), w.end()) << '\n';
  auto pair = mismatch(v.begin(), v.end(), w.begin(), w.end());
  if (pair.first != v.end())
    std::cout << *pair.first << '\n';
  if (pair.second != w.end())
    std::cout << *pair.second << '\n';
}
```

Besides the algorithms from the C++11 standard library, Boost.Algorithm also defines algorithms that will very likely be added to the standard library with C++14. Example 29.3 uses new variants of two of these functions, `boost::algorithm::equal()` and `boost::algorithm::mismatch()`. In contrast to the identically named functions that have been part of the standard library since C++98, four iterators, rather than three, are passed to these new functions. The algorithms in Example 29.3 don't expect the second sequence to contain as many elements as the first sequence.
While `boost::algorithm::equal()` returns a bool, `boost::algorithm::mismatch()` returns two iterators in a `std::pair`. **`first`** and **`second`** refer to the elements in the first and second sequence that are the first ones mismatching. These iterators may also refer to the end of a sequence.
Example 29.3 writes `false` and `3` to standard output. `false` is the return value of `boost::algorithm::equal()`, `3` the third element in **`w`**. Because the first two elements in **`v`** and **`w`** are equal, `boost::algorithm::mismatch()` returns, in **`first`**, an iterator to the end of **`v`** and, in **`second`**, an iterator to the third element of **`w`**. Because **`first`** refers to the end of **`v`**, the iterator isn't de-referenced, and there is no output.

Example 29.4 Using `boost::algorithm::hex()` and `boost::algorithm::unhex()`

```
#include <boost/algorithm/hex.hpp>
#include <vector>
#include <string>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<char> v{'C', '+', '+'};
  hex(v, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string s = "C++";
  std::cout << hex(s) << '\n';

  std::vector<char> w{'4', '3', '2', 'b', '2', 'b'};
  unhex(w, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string t = "432b2b";
  std::cout << unhex(t) << '\n';
}
```

Example 29.4 uses the two functions `boost::algorithm::hex()` and `boost::algorithm::unhex()`. These functions are designed after the identically named functions from the database system MySQL. They convert characters to hexadecimal values or hexadecimal values to characters.
Example 29.4 passes the vector **`v`** with the characters “C”, “+”, and “+” to `boost::algorithm::hex()`. This function expects an iterator as the second parameter to write the hexadecimal values to. The example writes `43` for “C” and `2B` (twice) for the two instances of “+” to standard output. The second call to `boost::algorithm::hex()` does the same thing except that “C++” is passed as a string and “432B2B” is returned as a string.
`boost::algorithm::unhex()` is the opposite of `boost::algorithm::hex()`. If the array **`w`** from Example 29.4 is passed with six hexadecimal values, each of the three pairs of values is interpreted as ASCII-Code. The same happens with the second call to `boost::algorithm::unhex()` when six hexadecimal values are passed as a string. In both cases `C++` is written to standard output.
Boost.Algorithm provides even more algorithms. For example, there are several string matching algorithms that search text efficiently. The documentation contains an overview of all available algorithms.

Chapter 30

Boost.Range

Boost.Range is a library that, on the first sight, provides algorithms similar to those provided by the standard library. For example, you will find the function `boost::copy()`, which does the same thing as `std::copy()`. However, `std::copy()` expects two parameters while `boost::copy()` expects a range.

30.1 Algorithms

You can think of a *range* as two iterators that refer to the beginning and end of a group of elements that you can iterate over. Because all containers support iterators, every container can be thought of as a range. Since all algorithms from Boost.Range expect a range as a first parameter, a container like `std::vector` can be passed directly. You don't have to call `begin()` and `end()` and then pass two iterators separately. This protects you from mistakes such as passing the begin and end iterator in the wrong order or passing iterators that belong to two different containers.

Example 30.1 Counting with `boost::count()`

```
#include <boost/range/algorithm.hpp>
#include <array>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 0, 1, 0, 1}};
  std::cout << boost::count(a, 0) << '\n';
}
```

Example 30.1 uses the algorithm `boost::count()`, which is defined in `boost/range/algorithm.hpp`. This header file provides access to all of the algorithms for which counterparts exist in the standard library header file `algorithm`.

All algorithms from Boost.Range require the first parameter to be a range. An object of type `std::array` can be passed to `boost::count()` directly since containers are ranges. Because `boost::count()` is equivalent to `std::count()`, you must pass in the value that the elements in the range will be compared with.

In Example 30.1, **a** contains three zeros, so `3` is written to standard output.

Example 30.2 introduces more algorithms which, like `boost::count()`, are similar to algorithms from the standard library.

Example 30.2 Range algorithms related to algorithms from the standard library

```
#include <boost/range/algorithm.hpp>
#include <boost/range/numeric.hpp>
#include <array>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 2, 3, 4, 5}};
  boost::random_shuffle(a);
  boost::copy(a, std::ostream_iterator<int>{std::cout, ","});
  std::cout << "\n" << *boost::max_element(a) << '\n';
  std::cout << boost::accumulate(a, 0) << '\n';
}
```

`boost::random_shuffle()` works like `std::random_shuffle()`, changing the order of elements in a range randomly. Example 30.2 uses `boost::random_shuffle()` with a default random number generator. However, you can pass a random number generator as a second parameter. That can be a random number generator either from the C++11 header file `random` or from Boost.Random.

`boost::copy()` works like `std::copy()`. `boost::max_element()` and `boost::accumulate()` work like the identically named algorithms from the standard library. Like `std::max_element()`, `boost::max_element()` returns an iterator to the element with the greatest number.

The header file `boost/range/numeric.hpp` must be included for `boost::accumulate()`. Just as `std::accumulate()` is defined in `numeric`, `boost::accumulate()` is defined in `boost/range/numeric.hpp` and not in `boost/range/algorithm.hpp`.

Boost.Range also provides a few algorithms without counterparts in the standard library.

Example 30.3 Range algorithms without counterparts in the standard library

```
#include <boost/range/algorithm_ext.hpp>
#include <array>
#include <deque>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 2, 3, 4, 5}};
  std::cout << std::boolalpha << boost::is_sorted(a) << '\n';
  std::deque<int> d;
  boost::push_back(d, a);
  boost::remove_erase(d, 2);
  boost::copy_n(d, 3, std::ostream_iterator<int>{std::cout, ","});
}
```

The algorithms used in Example 30.3 require the header file `boost/range/algorithm_ext.hpp`. This header file provides access to algorithms that have no counterpart in the standard library.

`boost::is_sorted()` tests whether elements in a range are sorted. In Example 30.3, `boost::is_sorted()` returns `true` because **a** is sorted. A predicate can be passed as the second parameter to `boost::is_sorted()` to check, for example, whether a range is sorted in descending order.

`boost::push_back()` expects as its first parameter a container and as its second parameter a range. The container must define the member function `push_back()`. All elements from the range are added to the container using this member function, in the order specified by the range. Because **d** starts out empty, it will contain the same numbers as **a**, in the same order, after the call to `boost::push_back()`.

`boost::remove_erase()` removes the number 2 from **d**. This algorithm combines a call to the function `std::remove()` and a call to the member function `erase()` of the respective container. Thanks to `boost::remove_erase()`, you don't need to find the iterator to the element you need to remove and then pass it to `erase()` in a second step.

`boost::copy_n()` is similar to `boost::copy()`, but copies only as many elements as the number passed as its second parameter. Example 30.3 only writes the first three numbers from **d** to standard output. Because 2 was removed from **d** in the previous line, `0,1,3,` is displayed.

30.2 Adaptors

The standard library provides several algorithms you can pass a predicate to. For example, the predicate passed to `std::count_if()` determines which elements are counted. Boost.Range provides the similar function `boost::count_if()`. However, this algorithm is only provided for completeness because Boost.Range provides adaptors that make algorithms with predicates superfluous.

You can think of *adaptors* as filters. They return a new range based on another range. Data isn't necessarily copied. Since a range is just a pair of iterators, an adaptor returns a new pair. The pair can still be used to iter-

ate over the original range but may, for example, skip certain elements. If `boost::count()` is used with such an adaptor, `boost::count_if()` is no longer required. Algorithms don't have to be defined multiple times just so they can be called with and without predicates.
The difference between algorithms and adaptors is that algorithms iterate over a range and process data, while adaptors return a new range – new iterators – that determines what elements the iteration returns. However, no iteration is executed. An algorithm must be called first.

Example 30.4 Filtering a range with `boost::adaptors::filter()`

```
#include <boost/range/algorithm.hpp>
#include <boost/range/adaptors.hpp>
#include <array>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 5, 2, 1, 3, 4}};
  boost::copy(boost::adaptors::filter(a, [](int i){ return i > 2; }),
    std::ostream_iterator<int>{std::cout, ","});
}
```

Example 30.4 uses an adaptor that can filter ranges. As you can see, the adaptor is just a function. `boost::adaptors::filter()` expects as its first parameter a range to filter and as its second parameter a predicate. The predicate in Example 30.4 removes all numbers from the range that aren't greater than 2.
`boost::adaptors::filter()` does not change the range **a**, it returns a new range. Since a range isn't much different from a pair of iterators, the new range also refers to **a**. However, the iterators for the new range skip all numbers that are less than or equal to 2.
Example 30.4 writes `5,3,4` to standard output.
Example 30.5 uses two adaptors, `boost::adaptors::keys()` and `boost::adaptors::values()`, to access keys and values in a container of type `std::map`. It also shows how adaptors can be nested. Because **m** stores pointers to the values to be printed, rather than the values themselves, the range returned by `boost::adaptors::values()` is passed to `boost::adaptors::indirect()`. This adaptor can always be used when a range consists of pointers, but an iteration should return the values the pointers refer to. That's why Example 30.5 writes `a,b,c,0,1,2,` to standard output.

Example 30.5 Using `keys()`, `values()` and `indirect()`

```
#include <boost/range/algorithm.hpp>
#include <boost/range/adaptors.hpp>
#include <array>
#include <map>
#include <string>
#include <utility>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 3> a{{0, 1, 2}};
  std::map<std::string, int*> m;
  m.insert(std::make_pair("a", &a[0]));
  m.insert(std::make_pair("b", &a[1]));
  m.insert(std::make_pair("c", &a[2]));

  boost::copy(boost::adaptors::keys(m),
    std::ostream_iterator<std::string>{std::cout, ","});
  boost::copy(boost::adaptors::indirect(boost::adaptors::values(m)),
    std::ostream_iterator<int>{std::cout, ","});
}
```

Example 30.6 introduces an adaptor for strings. You can use `boost::adaptors::tokenize()` to get a range from a string with the help of a regular expression. You pass a string and a regular expression of the type `boost::regex` to `boost::adaptors::tokenize()`. In addition, you need to pass a number that refers to a group in the regular expression and a flag. If no group is used, you can pass 0. The flag **`boost::regex_constants::match_default`** selects the default settings for regular expressions. You can also pass other flags. For example, you can use **`boost::regex_constants::match_perl`** if you want the regular expression to be applied according to the rules for the programming language Perl.

Example 30.6 `boost::adaptors::tokenize()` – an adaptor for strings

```
#include <boost/range/algorithm.hpp>
#include <boost/range/adaptors.hpp>
#include <boost/regex.hpp>
#include <string>
#include <iostream>

int main()
{
  std::string s = "The Boost C++ Libraries";
  boost::regex expr{"[\\w+]+"};
  boost::copy(boost::adaptors::tokenize(s, expr, 0,
    boost::regex_constants::match_default),
    std::ostream_iterator<std::string>{std::cout, ","});
}
```

30.3 Helper Classes and Functions

The algorithms and adaptors provided by Boost.Range are based on templates. You don't have to transform a container to a range to pass it to an algorithm or adaptor. However, Boost.Range defines a few range classes, with `boost::iterator_range` being the most important. `boost::iterator_range` is required because adaptors and a few algorithms return ranges that must have a type. In addition, helper functions exist that create ranges whose iterators contain all the data required for an iteration. The iterators from these ranges don't refer to a container or to another data structure. A class like `boost::iterator_range` is used here, too.

Example 30.7 Creating a range for integers with `boost::irange()`

```
#include <boost/range/algorithm.hpp>
#include <boost/range/irange.hpp>
#include <iostream>

int main()
{
  boost::integer_range<int> ir = boost::irange(0, 3);
  boost::copy(ir, std::ostream_iterator<int>{std::cout, ","});
}
```

Example 30.7 uses the function `boost::irange()`. This function creates a range for integers without having to use a container or another data structure. You only pass a lower and upper bound to `boost::irange()` with the upper bound being exclusive.

`boost::irange()` returns a range of type `boost::integer_range`. This class is derived from `boost::iterator_range`. `boost::iterator_range` is a template that expects an iterator type as its sole template parameter. The iterator used by `boost::irange()` is tightly coupled to that function and is an implementation detail. Thus, the iterator type isn't known and can't be passed as a template parameter to `boost::iterator_range`. However, `boost::integer_range` only expects an integer type, which makes it easier to use than if you had to pass an iterator type.

Example 30.7 writes `0,1,2` to standard output.

Example 30.8 introduces the function `boost::istream_range()`, which creates a range for an input stream. The function returns the range as a `boost::iterator_range`. This means that an iterator type has to be passed as a template parameter.

When you start Example 30.8, type a number, and press **Enter**, the number is printed in the next line. If you type another number and press **Enter**, that number is printed. The range returned by `boost::istream_range()` makes it possible for `boost::copy()` to iterate over all entered numbers and write them to **`std::cout`**.

You can terminate the program any time by typing Ctrl+C.

Example 30.8 Creating a range for an input stream with `boost::istream_range()`

```
#include <boost/range/algorithm.hpp>
#include <boost/range/istream_range.hpp>
#include <iterator>
#include <iostream>

int main()
{
  boost::iterator_range<std::istream_iterator<int>> ir =
    boost::istream_range<int>(std::cin);
  boost::copy(ir, std::ostream_iterator<int>{std::cout, "\n"});
}
```

Besides `boost::iterator_range`, Boost.Range provides the class `boost::sub_range`, which is derived from `boost::iterator_range`. `boost::sub_range` is a template like `boost::iterator_range`. However, `boost::sub_range` expects the type of the range as a template parameter, not an iterator type. This can simplify usage.

Example 30.9 Creating ranges more easily with `boost::sub_range()`

```
#include <boost/range/algorithm.hpp>
#include <boost/range/iterator_range.hpp>
#include <boost/range/sub_range.hpp>
#include <array>
#include <iterator>
#include <iostream>

int main()
{
  std::array<int, 6> a{{0, 1, 2, 3, 4, 5}};
  boost::iterator_range<std::array<int, 6>::iterator> r1 =
    boost::random_shuffle(a);
  boost::sub_range<std::array<int, 6>> r2 =
    boost::random_shuffle(r1);
  boost::copy(r2, std::ostream_iterator<int>{std::cout, ","});
}
```

A few algorithms from Boost.Range return a range – for example, `boost::random_shuffle()`. This algorithm directly modifies the range passed to it by reference and returns the modified range. In Example 30.9, `boost::random_shuffle()` is called twice, so array **a** is randomly shuffled twice.
For the first return value, the example uses `boost::iterator_range`; for the second return value, it uses `boost::sub_range`. The usage of both classes only differs in the template parameter. Not only can `boost::sub_range` be instantiated more easily, but it also provides the type definition const_iterator.

Chapter 31

Boost.Graph

Boost.Graph provides tools to work with graphs. Graphs are two-dimensional point clouds with any number of lines between points. A subway map is a good example of a graph. Subway stations are points, which are connected by subway lines.
The graph theory is the field of mathematics that researches graphs. Graph theory tries to answer questions such as how to determine the shortest path between two points. Auto navigation systems have to solve that problem to guide drivers to their desired location using the shortest path. Graphs are very important in practice because many problems can be modelled with them.
Boost.Graph provides containers to define graphs. However, even more important are the algorithms Boost.Graph offers to operate on graphs, for example, to find the shortest path. This chapter introduces you to the containers and algorithms in Boost.Graph.

31.1 Vertices and Edges

Graphs consist of points and lines. To create a graph, you have to define a set of points and any lines between them. Example 31.1 contains a first simple graph consisting of four points and no lines.
Boost.Graph provides three containers to define graphs. The most important container is `boost::adjacency_list` which is used in nearly all of the examples in this chapter. To use this class, include the header file `boost/graph/adjacency_list.hpp`. If you want to use another container, you must include another header file. There is no master header file to get access to all classes and functions from Boost.Graph.

Example 31.1 A graph of type `boost::adjacency_list` with four vertices

```
#include <boost/graph/adjacency_list.hpp>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v3 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v4 = boost::add_vertex(g);

  std::cout << v1 << ", " << v2 << ", " << v3 << ", " << v4 << '\n';
}
```

`boost::adjacency_list` is a template that is instantiated with default parameters in Example 31.1. Later, you will see what parameters you can pass. This class is defined in `boost`. All classes and functions from Boost.Graph are defined in this namespace.
To add four points to the graph, the function `boost::add_vertex()` has to be called four times.
`boost::add_vertex()` is a free-standing function and not a member function of `boost::adjacency_list`. You will find there are many free-standing functions in Boost.Graph that could have been implemented as member functions. Boost.Graph is designed to be more of a generic library than an object-oriented library.

`boost::add_vertex()` adds a point to a graph. In graph theory, a point is called vertex, which explains the function name.
`boost::add_vertex()` returns an object of type `boost::adjacency_list::vertex_descriptor`. This object represents a newly added point in the graph. You can write the objects to standard output as shown in Example 31.1. The example displays `0, 1, 2, 3`.
Example 31.1 identifies points through positive integers. These numbers are indexes to a vector that is used internally in `boost::adjacency_list`. It's no surprise that `boost::add_vertex()` returns 0, 1, 2, and 3 since every call adds another point to the vector.
`std::vector` is the container `boost::adjacency_list` uses by default to store points. In this case, `boost::adjacency_list::vertex_descriptor` is a type definition for std::size_t. Because other containers can be used to store points, `boost::adjacency_list::vertex_descriptor` isn't necessarily always std::size_t.

Example 31.2 Accessing vertices with `boost::vertices()`

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;

  boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<boost::adjacency_list<>::vertex_iterator,
    boost::adjacency_list<>::vertex_iterator> vs = boost::vertices(g);

  std::copy(vs.first, vs.second,
    std::ostream_iterator<boost::adjacency_list<>::vertex_descriptor>{
      std::cout, "\n"});
}
```

To get all points from a graph, call `boost::vertices()`. This function returns two iterators of type `boost::adjacency_list::vertex_iterator`, which refer to the beginning and ending points. The iterators are returned in a `std::pair`. Example 31.2 uses the iterators to write all points to standard output. This example displays the number 0, 1, 2, and 3, just like the previous example.
Example 31.3 explains how points are connected with lines.
You call `boost::add_edge()` to connect two points in a graph. You have to pass the points and the graph as parameters. In graph theory, lines between points are called edges – that's why the function is called `boost::add_edge()`.
`boost::add_edge()` returns a `std::pair`. **`first`** provides access to the line. **`second`** is a bool variable that indicates whether the line was successfully added. If you run Example 31.3, you'll see that `p.second` is set to `true` for each call to `boost::add_edge()`, and a new line is added to the graph with each call.
`boost::edges()` provides access to all lines in a graph. Like `boost::vertices()`, `boost::edges()` returns two iterators that refer to the beginning and ending lines. Example 31.3 writes all lines to standard output. The example displays `(0,1)`, `(0,1)` and `(1,0)`.

Example 31.3 Accessing edges with `boost::edges()`

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;
  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<boost::adjacency_list<>::edge_descriptor, bool> p =
    boost::add_edge(v1, v2, g);
  std::cout.setf(std::ios::boolalpha);
  std::cout << p.second << '\n';

  p = boost::add_edge(v1, v2, g);
  std::cout << p.second << '\n';

  p = boost::add_edge(v2, v1, g);
  std::cout << p.second << '\n';

  std::pair<boost::adjacency_list<>::edge_iterator,
    boost::adjacency_list<>::edge_iterator> es = boost::edges(g);

  std::copy(es.first, es.second,
    std::ostream_iterator<boost::adjacency_list<>::edge_descriptor>{
      std::cout, "\n"});
}
```

The output shows that the graph has three lines. All three connect the first two points – those with the indexes 0 and 1. The output also shows where the lines start and end. Two lines start at the first point, one at the second. The direction of the lines depends on the order of the parameters passed to `boost::add_edge()`.

As you see, you can have multiple lines between the same two points. However, this feature can be deactivated. Example 31.4 doesn't instantiate `boost::adjacency_list` with default template parameters. Three parameters, called *selectors*, are passed in. By convention, the names of selectors end in S. These selectors determine what types will be used in `boost::adjacency_list` to store points and lines.

Example 31.4 `boost::adjacency_list` with selectors

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<graph::edge_descriptor, bool> p =
    boost::add_edge(v1, v2, g);
  std::cout.setf(std::ios::boolalpha);
  std::cout << p.second << '\n';

  p = boost::add_edge(v1, v2, g);
  std::cout << p.second << '\n';

  p = boost::add_edge(v2, v1, g);
  std::cout << p.second << '\n';

  std::pair<graph::edge_iterator,
    graph::edge_iterator> es = boost::edges(g);
  std::copy(es.first, es.second,
    std::ostream_iterator<graph::edge_descriptor>{std::cout, "\n"});
}
```

By default, `boost::adjacency_list` uses `std::vector` for points and lines. By passing `boost::setS` as the first template parameter in Example 31.4, `std::set` is selected as the container for lines. Because `std::set` doesn't support duplicates, it is not possible to add the same line using `boost::add_edge()` multiple times. Thus, the example only displays `(0,1)` once.
The second template parameter tells `boost::adjacency_list` which class should be used for points. In Example 31.4, `boost::vecS` is passed. This is the default value for the second template parameter. It is only set so that you can pass a third template parameter.
The third template parameter determines whether lines are directed or undirected. The default is `boost::directedS`, which means all lines are directed and can be drawn as arrows. Lines can only be crossed in one direction. `boost::undirectedS` is used in Example 31.4. This selector makes all lines undirected, which means it is possible to cross a line in any direction. It doesn't matter which point is the start and which is the end. This is another reason why the graph in Example 31.4 contains only one line. The third call to the function `boost::add_edge()` swaps the start and end points, but because lines in this example are undirected, this line is the same as the previous lines and, therefore, isn't added.
Boost.Graph offers more selectors, including `boost::listS`, `boost::mapS`, and `boost::hash_setS`. `boost::bidirectionalS` can be used to make lines bidirectional. This selector is similar to `boost::undirectedS`, but in this case, start and end points matter. If you use `boost::bidirectionalS` in Example 31.4, the third call to `boost::add_edge()` will add a line to the graph.
Example 31.5 shows a simpler method for adding points and lines to a graph.

Example 31.5 Creating indexes automatically with `boost::add_edge()`

```
#include <boost/graph/adjacency_list.hpp>
#include <tuple>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  enum { topLeft, topRight, bottomRight, bottomLeft };

  boost::add_edge(topLeft, topRight, g);
  boost::add_edge(topRight, bottomRight, g);
  boost::add_edge(bottomRight, bottomLeft, g);
  boost::add_edge(bottomLeft, topLeft, g);

  graph::edge_iterator it, end;
  std::tie(it, end) = boost::edges(g);
  std::copy(it, end,
    std::ostream_iterator<graph::edge_descriptor>{std::cout, "\n"});
}
```

Example 31.5 defines a graph consisting of four points. You can visualize the graph as a map with four fields, each represented by a point. The points are given the names `topLeft`, `topRight`, `bottomRight`, and `bottomLeft`. Because the names are assigned in an enumeration, each will have a numeric value that is used as an index.
It is possible to define a graph without calling `boost::add_vertex()`. Boost.Graph adds missing points to a graph automatically if the points passed to `boost::add_edge()` don't exist. The multiple calls to `boost::add_edge()` in Example 31.5 define not only lines but also add the four points required for the lines to the graph.
Please note how `std::tie()` is used to store the iterators returned in a `std::pair` from `boost::edges()` in **`it`** and **`end`**. `std::tie()` has been part of the standard library since C++11.
The graph in Example 31.5 is a map with four fields. To get from the top left to the bottom right, one can either cross the field in the top right or the one in the bottom left. There is no line between opposite fields. Thus it's not possible to go directly from the top left to the bottom right. All examples in this chapter use this graph.

Example 31.6 `boost::adjacent_vertices()` and `boost::out_edges()`

```
#include <boost/graph/adjacency_list.hpp>
#include <tuple>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  enum { topLeft, topRight, bottomRight, bottomLeft };

  boost::add_edge(topLeft, topRight, g);
  boost::add_edge(topRight, bottomRight, g);
  boost::add_edge(bottomRight, bottomLeft, g);
  boost::add_edge(bottomLeft, topLeft, g);

  graph::adjacency_iterator vit, vend;
  std::tie(vit, vend) = boost::adjacent_vertices(topLeft, g);
  std::copy(vit, vend,
    std::ostream_iterator<graph::vertex_descriptor>{std::cout, "\n"});

  graph::out_edge_iterator eit, eend;
  std::tie(eit, eend) = boost::out_edges(topLeft, g);
  std::for_each(eit, eend,
    [&g](graph::edge_descriptor it)
      { std::cout << boost::target(it, g) << '\n'; });
}
```

Example 31.6 introduces functions to gain additional information on points. `boost::adjacent_vertices()` returns a pair of iterators that refer to points a point connects to. You call `boost::out_edges()` if you want to access all outgoing lines from a point. `boost::in_edges()` accesses all ingoing lines. With undirected lines, it doesn't matter which of the two functions is called.
`boost::target()` returns the end point of a line. The start point is returned with `boost::source()`.
Example 31.6 writes 1 and 3, the indexes of the top right and bottom left fields, to standard output twice. `boost::adjacent_vertices()`, is called with **`topLeft`** and returns and displays the indexes of the top right and bottom left fields. **`topLeft`** is also passed to `boost::out_edges()` to retrieve the outgoing lines. Because `boost::target()` is called on every outgoing line with `std::for_each()`, the indexes of the top right and bottom left fields are displayed twice.
Example 31.7 illustrates how to define a graph with `boost::adjacency_list` without having to call `boost::add_edge()` for every line.

Example 31.7 Initializing `boost::adjacency_list` with lines

```
#include <boost/graph/adjacency_list.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  std::cout << boost::num_vertices(g) << '\n';
  std::cout << boost::num_edges(g) << '\n';

  g.clear();
}
```

You can pass iterators to the constructor of `boost::adjacency_list` that refer to objects of type std::pair<int, int>, which define lines. If you pass iterators, you also have to supply a third parameter that determines the total number of points in the graph. The graph will contain at least the points required for the lines. The third parameter let's you add points to the graph that aren't connected to other points.
Example 31.7 uses the functions `boost::num_vertices()` and `boost::num_edges()`, which return the number of points and lines, respectively. The example displays 4 twice.
Example 31.7 calls `boost::adjacency_list::clear()`. This member function removes all points and lines. It is a member function of `boost::adjacency_list` and not a free-standing function.

31.2 Algorithms

Algorithms from Boost.Graph resemble those from the standard library – they are generic and very flexible. However, it's not always immediately clear how they should be used.

Example 31.8 Visiting points from inside to outside with `breadth_first_search()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iterator>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> distances{{0}};

  boost::breadth_first_search(g, topLeft,
    boost::visitor(
      boost::make_bfs_visitor(
        boost::record_distances(distances.begin(),
          boost::on_tree_edge{}))));

  std::copy(distances.begin(), distances.end(),
    std::ostream_iterator<int>{std::cout, "\n"});
}
```

Example 31.8 uses the algorithm `boost::breadth_first_search()` to visit points from inside to outside. The algorithm starts at the point passed as the second parameter. It first visits all points that can be reached directly from that point, working like a wave.

`boost::breadth_first_search()` doesn't return a specific result. The algorithm just visits points. Whether data is collected and stored depends on the *visitors* passed to `boost::breadth_first_search()`.

Visitors are objects whose member functions are called when a point is visited. By passing visitors to an algorithm like `boost::breadth_first_search()`, you decide what should happen when a point is visited. Visitors are like function objects that can be passed to algorithms of the standard library.

Example 31.8 uses a visitor that records distances. A distance is the number of lines that have to be crossed to get from one point to another, starting at the point passed to `boost::breadth_first_search()` as the second parameter. Boost.Graph provides the helper function `boost::record_distances()` to create the visitor. A *property map* and a *tag* also have to be passed.

Property maps store properties for points or lines. Boost.Graph describes the concept of property maps. Since a pointer or iterator is taken as the beginning of a property map, it isn't important to understand property maps in detail. In Example 31.8 the beginning of the array **`distances`** is passed with `distances.begin()` to `boost::record_distances()`. This is sufficient for the array **`distances`** to be used as a property map. However, it is important that the size of the array isn't smaller than the number of points in the graph. After all, the distance to each and every point in the graph needs to be stored.

Please note that **`distances`** is based on `boost::array` and not on `std::array`. Using `std::array` would lead to a compiler error.

Depending on the algorithm, there are different events. The second parameter passed to `boost::record_distances()` specifies which events the visitor should be notified about. Boost.Graph defines tags that are empty classes to give events names. The tag `boost::on_tree_edge` in Example 31.8 specifies that a distance should be recorded when a new point has been found.

Events depend on the algorithm. You have to check the documentation on algorithms to find out which events are supported and which tags you can use.

A visitor created by `boost::record_distances()` is algorithm independent, so you can use `boost::record_distances()` with other algorithms. An adapter is used to bind an algorithm and a visitor. Example 31.8 calls `boost::make_bfs_visitor()` to create this adapter. This helper function returns a visitor as expected by the algorithm `boost::breadth_first_search()`. This visitor defines member functions that fit the events the algorithm supports. For example, the visitor returned by `boost::make_bfs_visitor()` defines the member function `tree_edge()`. If a visitor that is defined with the tag `boost::on_tree_edge` is passed to `boost::make_bfs_visitor()` (as in Example 31.8), the visitor is notified when `tree_edge()` is called. This lets you use visitors with different algorithms without those visitors having to define all of the member functions expected by all algorithms.

The adapter returned by `boost::make_bfs_visitor()` can't be passed directly to the algorithm `boost::breadth_first_search()`. It has to be wrapped with `boost::visitor()` and then passed as a third parameter.

There are two variants of algorithms like `boost::breadth_first_search()`. One variant expects that every parameter the algorithm supports will be passed. Another variant supports something similar to named parameters. It's typically easier to use this second variant because only the parameters you're interested in have to be passed. Many parameters don't have to be passed because algorithms use default values.

Example 31.8 uses the variant of `boost::breadth_first_search()` that expects named parameters. The first two parameters are the graph and the start point, which are required. However, the third parameter can be nearly everything. In Example 31.8 a visitor needs to be passed. For that to work, the adapter returned by `boost::make_bfs_visitor()` is named using `boost::visitor()`. Now, it's clear that the third parameter is a visitor. You'll see in the following examples how other parameters are passed by name to `boost::breadth_first_search()`.

Example 31.8 displays the numbers 0, 1, 2, and 1. These are the distances to all points from the top left. The top right field – the one with the index 1 – is only one step away. The bottom right field – the one with the index 2 – is two steps away. The bottom left field – the one with the index 3 – is again only one step away. The number 0 which is printed first refers to the top left field. Since it's the start point that was passed to `boost::breadth_first_search()`, zero steps are required to reach it.

`boost::breadth_first_search()` doesn't set the elements in the array, it just increases the stored values. Therefore, you must initialize all elements in the array **`distances`** before you start.

Example 31.9 illustrates how to find the shortest path.

Example 31.9 displays 0, 1, and 2. This is the shortest path from top left to bottom right. It leads over the top right field although the path over the bottom left field would be equally short.
`boost::breadth_first_search()` is used again – this time to find the shortest path. As you already know, this algorithm just visits points. To get a description of the shortest path, an appropriate visitor must be used. Example 31.9 calls `boost::record_predecessors()` to get one.
`boost::record_predecessors()` returns a visitor to store the predecessor of every point. Whenever `boost::breadth_first_search()` visits a new point, the previous point is stored in the property map passed to `boost::record_predecessors()`. As `boost::breadth_first_search()` visits points from the inside to the outside, the shortest path is found – starting at the point passed as a second parameter to `boost::breadth_first_search()`. Example 31.9 finds the shortest paths from all points in the graph to the bottom right.

Example 31.9 Finding paths with `breadth_first_search()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> predecessors;
  predecessors[bottomRight] = bottomRight;

  boost::breadth_first_search(g, bottomRight,
    boost::visitor(
      boost::make_bfs_visitor(
        boost::record_predecessors(predecessors.begin(),
          boost::on_tree_edge{}))));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
  std::cout << p << '\n';
}
```

After `boost::breadth_first_search()` returns, the property map **`predecessors`** contains the predecessor of every point. To find the first field when travelling from the top left to the bottom right, the element with the index 0 – the index of the top left field – is accessed in **`predecessors`**. The value found in **`predecessors`** is 1, which means the next field is at the top right. Accessing **`predecessors`** with the index 1 returns the next field. In Example 31.9 that's the bottom right field – the one with the index 2. That way it's possible to find the points iteratively in huge graphs to get from a start to an end point.

Example 31.10 Finding distances and paths with `breadth_first_search()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> distances{{0}};
  boost::array<int, 4> predecessors;
  predecessors[bottomRight] = bottomRight;

  boost::breadth_first_search(g, bottomRight,
    boost::visitor(
      boost::make_bfs_visitor(
        std::make_pair(
          boost::record_distances(distances.begin(),
            boost::on_tree_edge()),
          boost::record_predecessors(predecessors.begin(),
            boost::on_tree_edge{})))));

  std::for_each(distances.begin(), distances.end(),
    [](int d){ std::cout << d << '\n'; });

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
  std::cout << p << '\n';
}
```

Example 31.10 shows how `boost::breadth_first_search()` is used with two visitors. To use two visitors, you need to put them in a pair with `std::make_pair()`. If more than two visitors are needed, the pairs have to be nested. Example 31.10 does the same thing as Example 31.8 and Example 31.9 together.
`boost::breadth_first_search()` can only be used if every line has the same weight. This means the time taken to cross any line between points is always the same. If lines are weighted, meaning that each line may require a different amount of time to traverse, then you need to use a different algorithm to find the shortest path.

Example 31.11 Finding paths with `dijkstra_shortest_paths()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
```

```
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    boost::property<boost::edge_weight_t, int>> graph;

  std::array<int, 4> weights{{2, 1, 1, 1}};

  graph g{edges.begin(), edges.end(), weights.begin(), 4};

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Example 31.11 uses `boost::dijkstra_shortest_paths()` to find the shortest paths to the bottom right. This algorithm is used if lines are weighted. Example 31.11 assumes that it takes twice as long to cross the line from the top left to the top right as it takes to cross any other line.
Before `boost::dijkstra_shortest_paths()` can be used, weights have to be assigned to lines. This is done with the array **`weights`**. The elements in the array correspond to the lines in the graph. Because the line from the top left to the top right is first, the first element in **`weights`** is set to a value twice as big as all others.
To assign weights to lines, the iterator to the beginning of the array **`weights`** is passed as the third parameter to the constructor of the graph. This third parameter can be used to initialize properties of lines. This only works if properties have been defined for lines.
Example 31.11 passes additional template parameters to `boost::adjacency_list`. The fourth and fifth template parameter specify if points and lines have properties and what those properties are. You can assign properties to both lines and points.
By default, `boost::adjacency_list` uses `boost::no_property`, which means that neither points nor lines have properties. In Example 31.11, `boost::no_property` is passed as a fourth parameter to specify no properties for points. The fifth parameter uses `boost::property` to define a *bundled property*.
Bundled properties are properties that are stored internally in a graph. Because it's possible to define multiple bundled properties, `boost::property` expects a tag to define each property. Boost.Graph provides some tags, such as `boost::edge_weight_t`, to define frequently used properties that are automatically recognized and used by algorithms. The second template parameter passed to `boost::property` is the type of the property. In Example 31.11 weights are int values.
Example 31.11 works because `boost::dijkstra_shortest_paths()` automatically uses the bundled property of type `boost::edge_weight_t`.
Note that no visitor is passed to `boost::dijkstra_shortest_paths()`. This algorithm doesn't just visit points. It looks for shortest paths – that's why it's called `boost::dijkstra_shortest_paths()`. You don't need to think about events or visitors. You only need to pass a container to store the predecessor of every point.
If you use the variant of `boost::dijkstra_shortest_paths()` that expects named parameters, as in Ex-

ample 31.11, pass the container with `boost::predecessor_map()`. This is a helper function which expects a pointer or an iterator to the beginning of an array.
Example 31.11 displays 0, 3, and 2: The shortest path from top left to bottom right leads over the bottom left field. The path over the top right field has a greater weight than the other possibilities.

Example 31.12 User-defined properties with `dijkstra_shortest_paths()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    edge_properties> graph;

  graph g{edges.begin(), edges.end(), 4};

  graph::edge_iterator it, end;
  boost::tie(it, end) = boost::edges(g);
  g[*it].weight = 2;
  g[*++it].weight = 1;
  g[*++it].weight = 1;
  g[*++it].weight = 1;

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()).
    weight_map(boost::get(&edge_properties::weight, g)));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Example 31.12 works like the previous one and displays the same numbers, but it uses a user-defined class, `edge_properties`, rather than a predefined property.
`edge_properties` defines the member variable **`weight`** to store the weight of a line. It is possible to add more member variables if other properties are required.
You can access user-defined properties if you use the descriptor of lines as an index for the graph. Thus, the graph behaves like an array. You get the descriptors from the line iterators that are returned from `boost::edges()`.

That way a weight can be assigned to every line.
To make `boost::dijkstra_shortest_paths()` understand that weights are stored in **`weight`** in `edge_properties`, another named parameter has to be passed. This is done with `weight_map()`. Note that `weight_map()` is a member function of the object returned from `boost::predecessor_map()`. There is also a free-standing function called `boost::weight_map()`. If you need to pass multiple named parameters, you have to call a member function on the first named parameter (the one that was returned by the free-standing function). That way all parameters are packed into one object that is then passed to the algorithm.
To tell `boost::dijkstra_shortest_paths()` that **`weight`** in `edge_properties` contains the weights, a pointer to that property is passed. It isn't passed to `weight_map()` directly. Instead it is passed in an object created with `boost::get()`. Now the call is complete, and `boost::dijkstra_shortest_paths()` knows which property to access to get the weights.

Example 31.13 Initializing user-defined properties at graph definition

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    edge_properties> graph;

  boost::array<edge_properties, 4> props{{2, 1, 1, 1}};

  graph g{edges.begin(), edges.end(), props.begin(), 4};

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()).
    weight_map(boost::get(&edge_properties::weight, g)));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

It's possible to initialize user-defined properties when a graph is defined. You only have to pass an iterator as the third parameter to the constructor of `boost::adjacency_list`, which refers to objects of the type of the user-defined property. Thus, you don't need to access properties of lines through descriptors. Example 31.13 works like the previous one and displays the same result.

Example 31.14 Random paths with `random_spanning_tree()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/random_spanning_tree.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <random>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS> graph;

  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> predecessors;

  std::mt19937 gen{static_cast<uint32_t>(std::time(0))};
  boost::random_spanning_tree(g, gen,
    boost::predecessor_map(predecessors.begin()).
    root_vertex(bottomLeft));

  int p = topRight;
  while (p != -1)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
}
```

The algorithm introduced in Example 31.14 finds random paths. `boost::random_spanning_tree()` is similar to `boost::dijkstra_shortest_paths()`. It returns the predecessors of points in a container that is passed with `boost::predecessor_map`. In contrast to `boost::dijkstra_shortest_paths()`, the starting point isn't passed directly as a parameter to `boost::random_spanning_tree()`. It must be passed as a named parameter. That's why `root_vertex()` is called on the object of type `boost::predecessor_map`. Example 31.14 finds random paths to the bottom left field.
Because `boost::random_spanning_tree()` is looking for a random path, a random number generator has to be passed as the second parameter. Example 31.14 uses `std::mt19937`, which has been part of the standard library since C++11. You could also use a random number generator from Boost.Random.
Example 31.14 displays either 1, 0, and 3 or 1, 2, and 3. 1 is the top right field, 3 the bottom left field. There are only two possible paths from the top right field to the bottom left field: through the top left field or through the bottom right field. `boost::random_spanning_tree()` must return one of these two paths.

31.3 Containers

All examples in this chapter so far have used `boost::adjacency_list` to define graphs. This section introduces the two other graph containers provided by Boost.Graph: `boost::adjacency_matrix` and `boost::compressed_sparse_row_graph`.

> Note
>
> There is a missing include in `boost/graph/adjacency_matrix.hpp` in Boost 1.56.0. To compile Example 31.15 with Boost 1.56.0, include `boost/functional/hash.hpp` before `boost/graph/adjacency_matrix.hpp`.

Example 31.15 Graphs with `boost::adjacency_matrix`

```
#include <boost/graph/adjacency_matrix.hpp>
#include <array>
#include <utility>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_matrix<boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};
}
```

`boost::adjacency_matrix` is used like `boost::adjacency_list` (see Example 31.15). However, the two template parameters that pass selectors don't exist with `boost::adjacency_matrix`. With `boost::adjacency_matrix`, no selectors, such as `boost::vecS` and `boost::setS`, are used. `boost::adjacency_matrix` stores the graph in a matrix, and the internal structure is hardcoded. You can think of the matrix as a two-dimensional table: the table is a square with as many rows and columns as the graph has points. A line is created by marking the cell where the row and column that correspond with the two end points of the line intersect.
The internal structure of `boost::adjacency_matrix` makes it possible to add and remove lines quickly. However, memory consumption is higher. The rule of thumb is to use `boost::adjacency_list` when there are relatively few lines compared to points. The more lines there are, the more it makes sense to use `boost::adjacency_matrix`.

Example 31.16 Graphs with `boost::compressed_sparse_row_graph`

```
#include <boost/graph/compressed_sparse_row_graph.hpp>
#include <array>
#include <utility>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};
  typedef boost::compressed_sparse_row_graph<boost::bidirectionalS> graph;
  graph g{boost::edges_are_unsorted_multi_pass, edges.begin(),
    edges.end(), 4};
}
```

`boost::compressed_sparse_row_graph` is used in the same way as `boost::adjacency_list` and `boost::adjacency_matrix` (see Example 31.16). The most important difference is that graphs can't be changed with `boost::compressed_sparse_row_graph`. Once the graph has been created, points and lines can't be added or removed. Thus, `boost::compressed_sparse_row_graph` makes only sense when using an immutable graph.

`boost::compressed_sparse_row_graph` only supports directed lines. You can't instantiate `boost::compressed_sparse_row_graph` with the template parameter `boost::undirectedS`.

The main advantage of `boost::compressed_sparse_row_graph` is low memory consumption. `boost::compressed_sparse_row_graph` is especially useful if you have a huge graph and you need to keep memory consumption low.

Part VI

Communication

The following libraries facilitate communication with other programs.

- Boot.Asio is for communicating over networks. Boost.Asio does not just support network operations. Asio stands for asynchronous input/output. You can use this library to process data asynchronously, for example, when your program communicates with devices, such as network cards, that can handle tasks concurrently with code executed in your program.
- Boost.Interprocess is for communicating through shared memory.

Chapter 32

Boost.Asio

This chapter introduces the library Boost.Asio. Asio stands for asynchronous input/output. This library makes it possible to process data asynchronously. Asynchronous means that when operations are initiated, the initiating program does not need to wait for the operation to end. Instead, Boost.Asio notifies a program when an operation has ended. The advantage is that other operations can be executed concurrently.
Boost.Thread is another library that makes it possible to execute operations concurrently. The difference between Boost.Thread and Boost.Asio is that with Boost.Thread, you access resources inside of a program, and with Boost.Asio, you access resources outside of a program. For example, if you develop a function which needs to run a time-consuming calculation, you can call this function in a thread and make it execute on another CPU core. Threads allow you to access and use CPU cores. From the point of view of your program, CPU cores are an internal resource. If you want to access external resources, you use Boost.Asio.
Network connections are an example of external resources. If data has to be sent or received, a network card is told to execute the operation. For a send operation, the network card gets a pointer to a buffer with the data to send. For a receive operation the network card gets a pointer to a buffer it should fill with the data being received. Since the network card is an external resource for your program, it can execute the operations independently. It only needs time – time you could use in your program to execute other operations. Boost.Asio allows you to use the available devices more efficiently by benefiting from their ability to execute operations concurrently.
Sending and receiving data over a network is implemented as an asynchronous operation in Boost.Asio. Think of an asynchronous operation as a function that immediately returns, but without any result. The result is handed over later.
In the first step, an asynchronous operation is started. In the second step, a program is notified when the asynchronous operation has ended. This separation between starting and ending makes it possible to access external resources without having to call blocking functions.

32.1 I/O Services and I/O Objects

Programs that use Boost.Asio for asynchronous data processing are based on *I/O services* and *I/O objects*. I/O services abstract the operating system interfaces that process data asynchronously. I/O objects initiate asynchronous operations. These two concepts are required to separate tasks cleanly: I/O services look towards the operating system API, and I/O objects look towards tasks developers need to do.
As a user of Boost.Asio you normally don't connect with I/O services directly. I/O services are managed by an *I/O service object*. An I/O service object is like a registry where I/O services are listed. Every I/O object knows its I/O service and gets access to its I/O service through the I/O service object.
Boost.Asio defines `boost::asio::io_service`, a single class for an I/O service object. Every program based on Boost.Asio uses an object of type `boost::asio::io_service`. This can also be a global variable.
While there is only one class for an I/O service object, several classes for I/O objects exist. Because I/O objects are task oriented, the class that needs to be instantiated depends on the task. For example, if data has to be sent or received over a TCP/IP connection, an I/O object of type `boost::asio::ip::tcp::socket` can be used. If data has to be transmitted asynchronously over a serial port, `boost::asio::serial_port` can be instantiated. If you want to wait for a time period to expire, you can use the I/O object `boost::asio::steady_timer`. `boost::asio::steady_timer` is like an alarm clock. Instead of waiting for a blocking function to return when the alarm clock rings, your program will be notified. Because `boost::asio::steady_timer` just waits

for a time period to expire, it would seem as though no external resource is accessed. However, in this case the external resource is the capability of the operating system to notify a program when a time period expires. This frees a program from creating a new thread just to call a blocking function. Since `boost::asio::steady_timer` is a very simple I/O object, it will be used to introduce Boost.Asio.

> Note
>
> Because of a bug in Boost.Asio, it is not possible to compile some of the following examples with Clang. The bug has been reported in ticket 8835. As a workaround, if you replace the types from `std::chrono` with the respective types from `boost::chrono`, you can compile the examples with Clang.

Example 32.1 creates an I/O service object, **`ioservice`**, and uses it to initialize the I/O object **`timer`**. Like `boost::asio::steady_timer`, all I/O objects expect an I/O service object as a first parameter in their constructor. Since **`timer`** represents an alarm clock, a second parameter can be passed to the constructor of `boost::asio::steady_timer` that defines the specific time or time period when the alarm clock should ring. In Example 32.1, the alarm clock is set to ring after 3 seconds. The time starts with the definition of **`timer`**.

Example 32.1 Using `boost::asio::steady_timer`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  steady_timer timer{ioservice, std::chrono::seconds{3}};
  timer.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  ioservice.run();
}
```

Instead of calling a blocking function that will return when the alarm clock rings, Boost.Asio lets you start an asynchronous operation. To do this, call the member function `async_wait()`, which expects a *handler* as the sole parameter. A handler is a function or function object that is called when the asynchronous operation ends. In Example 32.1, a lambda function is passed as a handler.

`async_wait()` returns immediately. Instead of waiting three seconds until the alarm clock rings, the lambda function is called after three seconds. When `async_wait()` returns, a program can do something else.

A member function like `async_wait()` is called non-blocking. I/O objects usually also provide blocking member functions as alternatives. For example, you can call the blocking member function `wait()` on `boost::asio::steady_timer`. Because this member function is blocking, no handler is passed. `wait()` returns at a specific time or after a time period.

The last statement in `main()` in Example 32.1 is a call to `run()` on the I/O service object. This call is required because operating system-specific functions have to take over control. Remember that it is the I/O service in the I/O service object which implements asynchronous operations based on operating system-specific functions.

While `async_wait()` initiates an asynchronous operation and returns immediately, `run()` blocks. Many operating systems support asynchronous operations only through a blocking function. The following example shows why this usually isn't a problem.

Example 32.2 Two asynchronous operations with `boost::asio::steady_timer`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <iostream>
```

```
using namespace boost::asio;

int main()
{
  io_service ioservice;

  steady_timer timer1{ioservice, std::chrono::seconds{3}};
  timer1.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  steady_timer timer2{ioservice, std::chrono::seconds{4}};
  timer2.async_wait([](const boost::system::error_code &ec)
    { std::cout << "4 sec\n"; });

  ioservice.run();
}
```

In Example 32.2, two objects of type `boost::asio::steady_timer` are used. The first I/O object is an alarm clock that rings after three seconds. The other is an alarm clock ringing after four seconds. After both time periods expire, the lambda functions that were passed to `async_wait()` will be called.
`run()` is called on the only I/O service object in this example. This call passes control to the operating system functions that execute asynchronous operations. With their help, the first lambda function is called after three seconds and the second lambda function after four seconds.
It might come as a surprise that asynchronous operations require a call to a blocking function. However, this is not a problem because the program has to be prevented from exiting. If `run()` wouldn't block, `main()` would return, and the program would exit. If you don't want to wait for `run()` to return, you only need to call `run()` in a new thread.
The reason why the examples above exit after a while is that `run()` returns if there are no pending asynchronous operations. Once all alarm clocks have rung, no asynchronous operations exist that the program needs to wait for.

32.2 Scalability and Multithreading

Developing a program based on a library like Boost.Asio differs from the usual C++ style. Functions that may take longer to return are no longer called in a sequential manner. Instead of calling blocking functions, Boost.Asio starts asynchronous operations. Functions which should be called after an operation has finished are now called within the corresponding handler. The drawback of this approach is the physical separation of sequentially executed functions, which can make code more difficult to understand.
A library such as Boost.Asio is typically used to achieve greater efficiency. With no need to wait for an operation to finish, a program can perform other tasks in between. Therefore, it is possible to start several asynchronous operations that are all executed concurrently – remember that asynchronous operations are usually used to access resources outside of a process. Since these resources can be different devices, they can work independently and execute operations concurrently.
Scalability describes the ability of a program to effectively benefit from additional resources. With Boost.Asio it is possible to benefit from the ability of external devices to execute operations concurrently. If threads are used, several functions can be executed concurrently on available CPU cores. Boost.Asio with threads improves the scalability because your program can take advantage of internal and external devices that can execute operations independently or in cooperation with each other.
If the member function `run()` is called on an object of type `boost::asio::io_service`, the associated handlers are invoked within the same thread. By using multiple threads, a program can call `run()` multiple times. Once an asynchronous operation is complete, the I/O service object will execute the handler in one of these threads. If a second operation is completed shortly after the first one, the I/O service object can execute the handler in a different thread. Now, not only can operations outside of a process be executed concurrently, but handlers within the process can be executed concurrently, too.

Example 32.3 Two threads for the I/O service object to execute handlers concurrently

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <thread>
```

```
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  steady_timer timer1{ioservice, std::chrono::seconds{3}};
  timer1.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  steady_timer timer2{ioservice, std::chrono::seconds{3}};
  timer2.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  std::thread thread1{[&ioservice](){ ioservice.run(); }};
  std::thread thread2{[&ioservice](){ ioservice.run(); }};
  thread1.join();
  thread2.join();
}
```

The previous example has been converted to a multithreaded program in Example 32.3. With `std::thread`, two threads are created in `main()`. `run()` is called on the only I/O service object in each thread. This makes it possible for the I/O service object to use both threads to execute handlers when asynchronous operations complete.

In Example 32.3, both alarm clocks should ring after three seconds. Because two threads are available, both lambda functions can be executed concurrently. If the second alarm clock rings while the handler of the first alarm clock is being executed, the handler can be executed in the second thread. If the handler of the first alarm clock has already returned, the I/O service object can use any thread to execute the second handler.

Of course, it doesn't always make sense to use threads. Example 32.3 might not write the messages sequentially to the standard output stream. Instead, they might be mixed up. Both handlers, which may run in two threads concurrently, share the global resource **`std::cout`**. To avoid interruptions, access to **`std::cout`** would need to be synchronized. The advantage of threads is lost if handlers can't be executed concurrently.

Example 32.4 One thread for each of two I/O service objects to execute handlers concurrently

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/steady_timer.hpp>
#include <chrono>
#include <thread>
#include <iostream>

using namespace boost::asio;

int main()
{
  io_service ioservice1;
  io_service ioservice2;

  steady_timer timer1{ioservice1, std::chrono::seconds{3}};
  timer1.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  steady_timer timer2{ioservice2, std::chrono::seconds{3}};
  timer2.async_wait([](const boost::system::error_code &ec)
    { std::cout << "3 sec\n"; });

  std::thread thread1{[&ioservice1](){ ioservice1.run(); }};
  std::thread thread2{[&ioservice2](){ ioservice2.run(); }};
  thread1.join();
  thread2.join();
}
```

Calling `run()` repeatedly on a single I/O service object is the recommended method to make a program based on Boost.Asio more scalable. However, instead of providing several threads to one I/O service object, you could also create multiple I/O service objects.
Two I/O service objects are used next to two alarm clocks of type `boost::asio::steady_timer` in Example 32.4. The program is based on two threads, with each thread bound to another I/O service object. The two I/O objects **`timer1`** and **`timer2`** aren't bound to the same I/O service object anymore. They are bound to different objects.
Example 32.4 works the same as before. It's not possible to give general advice about when it makes sense to use more than one I/O service object. Because `boost::asio::io_service` represents an operating system interface, any decision depends on the particular interface.
On Windows, `boost::asio::io_service` is usually based on IOCP, on Linux, it is based on `epoll()`. Having several I/O service objects means that several I/O completion ports will be used or `epoll()` will be called multiple times. Whether this is better than using just one I/O completion port or one call to `epoll()` depends on the individual case.

32.3 Network programming

Even though Boost.Asio can process any kind of data asynchronously, it is mainly used for network programming. This is because Boost.Asio supported network functions long before additional I/O objects were added. Network functions are a perfect use for asynchronous operations because the transmission of data over a network may take a long time, which means acknowledgments and errors may not be available as fast as the functions that send or receive data can execute.
Boost.Asio provides many I/O objects to develop network programs. Example 32.5 uses the class `boost::asio::ip::tcp::socket` to establish a connection with another computer. This example sends a HTTP request to a web server to download the homepage.
Example 32.5 uses three handlers: `connect_handler()` and `read_handler()` are called when the connection is established and data is received. `resolve_handler()` is used for name resolution.
Because data can only be received after a connection has been established, and because a connection can only be established after the name has been resolved, the various asynchronous operations are started in handlers. In `resolve_handler()`, the iterator **`it`**, which points to an endpoint resolved from the name, is used with **`tcp_socket`** to establish a connection. In `connect_handler()`, **`tcp_socket`** is accessed to send a HTTP request and start receiving data. Since all operations are asynchronous, handlers are passed to the respective functions. Depending on the operations, additional parameters may need to be passed. For example, the iterator **`it`** refers to an endpoint resolved from a name. The array **`bytes`** is used to store data received.
In `main()`, `boost::asio::ip::tcp::resolver::query` is instantiated to create an object **`q`**. **`q`** represents a query for the name resolver, an I/O object of type `boost::asio::ip::tcp::resolver`. By passing **`q`** to `async_resolve()`, an asynchronous operation is started to resolve the name. Example 32.5 resolves the name `theboostcpplibraries.com`. After the asynchronous operation has been started, `run()` is called on the I/O service object to pass control to the operating system.

Example 32.5 A web client with `boost::asio::ip::tcp::socket`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/buffer.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <array>
#include <string>
#include <iostream>

using namespace boost::asio;
using namespace boost::asio::ip;

io_service ioservice;
tcp::resolver resolv{ioservice};
tcp::socket tcp_socket{ioservice};
std::array<char, 4096> bytes;

void read_handler(const boost::system::error_code &ec,
  std::size_t bytes_transferred)
{
  if (!ec)
  {
    std::cout.write(bytes.data(), bytes_transferred);
    tcp_socket.async_read_some(buffer(bytes), read_handler);
  }
}

void connect_handler(const boost::system::error_code &ec)
{
  if (!ec)
  {
    std::string r =
      "GET / HTTP/1.1\r\nHost: theboostcpplibraries.com\r\n\r\n";
    write(tcp_socket, buffer(r));
    tcp_socket.async_read_some(buffer(bytes), read_handler);
  }
}

void resolve_handler(const boost::system::error_code &ec,
  tcp::resolver::iterator it)
{
  if (!ec)
    tcp_socket.async_connect(*it, connect_handler);
}

int main()
{
  tcp::resolver::query q{"theboostcpplibraries.com", "80"};
  resolv.async_resolve(q, resolve_handler);
  ioservice.run();
}
```

When the name has been resolved, `resolve_handler()` is called. The handler first checks whether the name resolution has been successful. In this case **`ec`** is 0. Only then is the socket accessed to establish a connection. The address of the server to connect to is provided by the second parameter, which is of type `boost::asio::ip::tcp::resolver::iterator`. This parameter is the result of the name resolution.
The call to `async_connect()` is followed by a call to the handler `connect_handler()`. Again **`ec`** is checked first to find out whether a connection could be established. If so, `async_read_some()` is called on the socket. With this call, reading data begins. Data being received is stored in the array **`bytes`**, which is passed as a first parameter to `async_read_some()`.
`read_handler()` is called when one or more bytes have been received and copied to **`bytes`**. The parameter **`bytes_transferred`** of type std::size_t contains the number of bytes that have been received. As usual, the handler should check first **`ec`** whether the asynchronous operation was completed successfully. Only if this is the case is data written to standard output.
Please note that `read_handler()` calls `async_read_some()` again after data has been written to **`std::cout`**. This is required because you can't be sure that the entire homepage was downloaded and copied into **`bytes`** in a single asynchronous operation. The repeated calls to `async_read_some()` followed by the repeated calls to `read_handler()` only end when the connection is closed, which happens when the web server has sent the entire homepage. Then `read_handler()` reports an error in **`ec`**. At this point, no further data is written to **`std::cout`** and `async_read()` is not called on the socket. Because there are no pending asynchronous operations, the program exits.
Example 32.6 is a time server. You can connect with a telnet client to get the current time. Afterwards the time server shuts down.
The time server uses the I/O object `boost::asio::ip::tcp::acceptor` to accept an incoming connection from another program. You must initialize the object so it knows which protocol to use on which port. In the example, the variable **`tcp_endpoint`** of type `boost::asio::ip::tcp::endpoint` is used to tell **`tcp_acceptor`** to accept incoming connections of version 4 of the internet protocol on port 2014.
After the acceptor has been initialized, `listen()` is called to make the acceptor start listening. Then `async_accept()` is called to accept the first connection attempt. A socket has to be passed as a first parameter to `async_accept()`, which will be used to send and receive data on a new connection.
Once another program establishes a connection, `accept_handler()` is called. If the connection was established

successfully, the current time is sent with `boost::asio::async_write()`. This function writes all data in **`data`** to the socket. `boost::asio::ip::tcp::socket` also provides the member function `async_write_some()`. This function calls the handler when at least one byte has been sent. Then the handler must check how many bytes were sent and how many still have to be sent. Then, once again, it has to call `async_write_some()`. Repeatedly calculating the number of bytes left to send and calling `async_write_some()` can be avoided by using `boost::asio::async_write()`. The asynchronous operation that started with this function is only complete when all bytes in **`data`** have been sent.

Example 32.6 A time server with `boost::asio::ip::tcp::acceptor`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/buffer.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <string>
#include <ctime>

using namespace boost::asio;
using namespace boost::asio::ip;

io_service ioservice;
tcp::endpoint tcp_endpoint{tcp::v4(), 2014};
tcp::acceptor tcp_acceptor{ioservice, tcp_endpoint};
tcp::socket tcp_socket{ioservice};
std::string data;

void write_handler(const boost::system::error_code &ec,
  std::size_t bytes_transferred)
{
  if (!ec)
    tcp_socket.shutdown(tcp::socket::shutdown_send);
}

void accept_handler(const boost::system::error_code &ec)
{
  if (!ec)
  {
    std::time_t now = std::time(nullptr);
    data = std::ctime(&now);
    async_write(tcp_socket, buffer(data), write_handler);
  }
}

int main()
{
  tcp_acceptor.listen();
  tcp_acceptor.async_accept(tcp_socket, accept_handler);
  ioservice.run();
}
```

After the data has been sent, `write_handler()` is called. This function calls `shutdown()` with the parameter **`boost::asio::ip::tcp::socket::shutdown_send`**, which says the program is done sending data through the socket. Since there are no pending asynchronous operations, Example 32.6 exits. Please note that although **`data`** is only used in `accept_handler()`, it can't be a local variable. **`data`** is passed by reference through `boost::asio::buffer()` to `boost::asio::async_write()`. When `boost::asio::async_write()` and `accept_handler()` return, the asynchronous operation has started, but has not completed. **`data`** must exist until the asynchronous operation has completed. If **`data`** is a global variable, this is guaranteed.

32.4 Coroutines

Since version 1.54.0, Boost.Asio supports coroutines. While you could use Boost.Coroutine directly, explicit support of coroutines in Boost.Asio makes it easier to use them.

Example 32.7 Coroutines with Boost.Asio

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/spawn.hpp>
#include <boost/asio/write.hpp>
#include <boost/asio/buffer.hpp>
#include <boost/asio/ip/tcp.hpp>
#include <list>
#include <string>
#include <ctime>

using namespace boost::asio;
using namespace boost::asio::ip;

io_service ioservice;
tcp::endpoint tcp_endpoint{tcp::v4(), 2014};
tcp::acceptor tcp_acceptor{ioservice, tcp_endpoint};
std::list<tcp::socket> tcp_sockets;

void do_write(tcp::socket &tcp_socket, yield_context yield)
{
  std::time_t now = std::time(nullptr);
  std::string data = std::ctime(&now);
  async_write(tcp_socket, buffer(data), yield);
  tcp_socket.shutdown(tcp::socket::shutdown_send);
}

void do_accept(yield_context yield)
{
  for (int i = 0; i < 2; ++i)
  {
    tcp_sockets.emplace_back(ioservice);
    tcp_acceptor.async_accept(tcp_sockets.back(), yield);
    spawn(ioservice, [](yield_context yield)
      { do_write(tcp_sockets.back(), yield); });
  }
}

int main()
{
  tcp_acceptor.listen();
  spawn(ioservice, do_accept);
  ioservice.run();
}
```

Coroutines let you create a structure that mirrors the actual program logic. Asynchronous operations don't split functions, because there are no handlers to define what should happen when an asynchronous operation completes. Instead of having handlers call each other, the program can use a sequential structure.

The function to call to use coroutines with Boost.Asio is `boost::asio::spawn()`. The first parameter passed must be an I/O service object. The second parameter is the function that will be the coroutine. This function must accept as its only parameter an object of type `boost::asio::yield_context`. It must have no return value. Example 32.7 uses `do_accept()` and `do_write()` as coroutines. If the function signature is different, as is the case for `do_write()`, you must use an adapter like `std::bind` or a lambda function.

Instead of a handler, you can pass an object of type `boost::asio::yield_context` to asynchronous functions. `do_accept()` passes the parameter **`yield`** to `async_accept()`. In `do_write()`, **`yield`** is passed to `async_write()`. These function calls still start asynchronous operations, but no handlers will be called when the operations complete. Instead, the context in which the asynchronous operations were started is restored. When these asynchronous operations complete, the program continues where it left off.

`do_accept()` contains a `for` loop. A new socket is passed to `async_accept()` every time the function is called. Once a client establishes a connection, `do_write()` is called as a coroutine with `boost::asio::spawn()` to send the current time to the client.

The `for` loop makes it easy to see that the program can serve two clients before it exits. Because the example is based on coroutines, the repeated execution of an asynchronous operation can be implemented in a `for` loop. This improves the readability of the program since you don't have to trace potential calls to handlers to find out when the last asynchronous operation will be completed. If the time server needs to support more than two clients, only the `for` loop has to be adapted.

32.5 Platform-specific I/O Objects

So far, all of the examples in this chapter have been platform independent. I/O objects such as `boost::asio::steady_timer` and `boost::asio::ip::tcp::socket` are supported on all platforms. However, Boost.Asio also provides platform-specific I/O objects because some asynchronous operations are only available on certain platforms, for example, Windows or Linux.
Example 32.8 uses the I/O object `boost::asio::windows::object_handle`, which is only available on Windows. `boost::asio::windows::object_handle`, which is based on the Windows function `RegisterWaitForSingleObject()`, lets you start asynchronous operations for object handles. All handles accepted by `RegisterWaitForSingleObject()` can be used with `boost::asio::windows::object_handle`. With `async_wait()`, it is possible to wait asynchronously for an object handle to change.

Example 32.8 Using `boost::asio::windows::object_handle`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/windows/object_handle.hpp>
#include <boost/system/error_code.hpp>
#include <iostream>
#include <Windows.h>

using namespace boost::asio;
using namespace boost::system;

int main()
{
  io_service ioservice;

  HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED, NULL);

  char buffer[1024];
  DWORD transferred;
  OVERLAPPED overlapped;
  ZeroMemory(&overlapped, sizeof(overlapped));
  overlapped.hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
  ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer), FALSE,
    FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, &overlapped, NULL);

  windows::object_handle obj_handle{ioservice, overlapped.hEvent};
  obj_handle.async_wait([&buffer, &overlapped](const error_code &ec) {
    if (!ec)
    {
      DWORD transferred;
      GetOverlappedResult(overlapped.hEvent, &overlapped, &transferred,
        FALSE);
      auto notification = reinterpret_cast<FILE_NOTIFY_INFORMATION*>(buffer);
      std::wcout << notification->Action << '\n';
      std::streamsize size = notification->FileNameLength / sizeof(wchar_t);
      std::wcout.write(notification->FileName, size);
    }
  });

  ioservice.run();
}
```

Example 32.8 initializes the object **`obj_handle`** of type `boost::asio::windows::object_handle` with an object handle created with the Windows function `CreateEvent()`. The handle is part of an `OVERLAPPED` structure whose address is passed to the Windows function `ReadDirectoryChangesW()`. Windows uses `OVERLAPPED` structures to start asynchronous operations.

`ReadDirectoryChangesW()` can be used to monitor a directory and wait for changes. To call the function asynchronously, an `OVERLAPPED` structure must be passed to `ReadDirectoryChangesW()`. To report the completion of the asynchronous operation through Boost.Asio, an event handler is stored in the `OVERLAPPED` structure before it is passed to `ReadDirectoryChangesW()`. This event handler is passed to **`obj_handle`** afterwards. When `async_wait()` is called on **`obj_handle`**, the handler is executed when a change is detected in the observed directory.

When you run Example 32.8, create a new file in the directory from which you will run the example. The program will detect the new file and write a message to the standard output stream.

Example 32.9 uses `ReadDirectoryChangesW()` like the previous one to monitor a directory. This time, the asynchronous call to `ReadDirectoryChangesW()` isn't linked to an event handle. The example uses the class `boost::asio::windows::overlapped_ptr`, which uses an `OVERLAPPED` structure internally. `get()` retrieves a pointer to the internal `OVERLAPPED` structure. In the example, the pointer is then passed to `ReadDirectoryChangesW()`.

`boost::asio::windows::overlapped_ptr` is an I/O object that has no member function to start an asynchronous operation. The asynchronous operation is started by passing a pointer to the internal `OVERLAPPED` variable to a Windows function. In addition to an I/O service object, the constructor of `boost::asio::windows::overlapped_ptr` expects a handler that will be called when the asynchronous operation completes.

Example 32.9 uses `boost::asio::use_service()` to get a reference to a service in the I/O service object **`ioservice`**. `boost::asio::use_service()` is a function template. The type of the I/O service you want to fetch has to be passed as a template parameter. In the example, boost::asio::detail::io_service_impl is passed. This type of the I/O service is closest to the operating system. On Windows, boost::asio::detail::io_service_impl uses IOCP, and on Linux it uses `epoll()`. boost::asio::detail::io_service_impl is a type definition that is set to `boost::asio::detail::win_iocp_io_service` on Windows and to `boost::asio::detail::task_io_service` on Linux.

`boost::asio::detail::win_iocp_io_service` provides the member function `register_handle()` to link a handle to an IOCP handle. `register_handle()` calls the Windows function `CreateIoCompletionPort()`. This call is required for the example to work correctly. The handle returned by `CreateFileA()` may be passed through **`overlapped`** to `ReadDirectoryChangesW()` only after it is linked to an IOCP handle.

Example 32.9 checks whether `ReadDirectoryChangesW()` has failed. If `ReadDirectoryChangesW()` failed, `complete()` is called on **`overlapped`** to complete the asynchronous operation for Boost.Asio. The parameters passed to `complete()` are forwarded to the handler.

If `ReadDirectoryChangesW()` succeeds, `release()` is called. The asynchronous operation is then pending and is only completed after the operation which was initiated with the Windows function `ReadDirectoryChangesW()` has completed.

Example 32.9 Using `boost::asio::windows::overlapped_ptr`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/windows/overlapped_ptr.hpp>
#include <boost/system/error_code.hpp>
#include <iostream>
#include <Windows.h>

using namespace boost::asio;
using namespace boost::system;

int main()
{
  io_service ioservice;

  HANDLE file_handle = CreateFileA(".", FILE_LIST_DIRECTORY,
    FILE_SHARE_READ | FILE_SHARE_WRITE | FILE_SHARE_DELETE, NULL,
    OPEN_EXISTING, FILE_FLAG_BACKUP_SEMANTICS | FILE_FLAG_OVERLAPPED, NULL);

  error_code ec;
  auto &io_service_impl = use_service<detail::io_service_impl>(ioservice);
  io_service_impl.register_handle(file_handle, ec);

  char buffer[1024];
  auto handler = [&buffer](const error_code &ec, std::size_t) {
    if (!ec)
    {
      auto notification =
        reinterpret_cast<FILE_NOTIFY_INFORMATION*>(buffer);
      std::wcout << notification->Action << '\n';
      std::streamsize size = notification->FileNameLength / sizeof(wchar_t);
      std::wcout.write(notification->FileName, size);
    }
  };
  windows::overlapped_ptr overlapped{ioservice, handler};
  DWORD transferred;
  BOOL ok = ReadDirectoryChangesW(file_handle, buffer, sizeof(buffer),
    FALSE, FILE_NOTIFY_CHANGE_FILE_NAME, &transferred, overlapped.get(),
    NULL);
  int last_error = GetLastError();
  if (!ok && last_error != ERROR_IO_PENDING)
  {
    error_code ec{last_error, error::get_system_category()};
    overlapped.complete(ec, 0);
  }
  else
  {
    overlapped.release();
  }

  ioservice.run();
}
```

Example 32.10 Using `boost::asio::posix::stream_descriptor`

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/posix/stream_descriptor.hpp>
#include <boost/asio/write.hpp>
#include <boost/system/error_code.hpp>
#include <iostream>
#include <unistd.h>

using namespace boost::asio;

int main()
{
  io_service ioservice;

  posix::stream_descriptor stream{ioservice, STDOUT_FILENO};
  auto handler = [](const boost::system::error_code&, std::size_t) {
    std::cout << ", world!\n";
  };
  async_write(stream, buffer("Hello"), handler);

  ioservice.run();
}
```

Example 32.10 introduces an I/O object for POSIX platforms.
`boost::asio::posix::stream_descriptor` can be initialized with a file descriptor to start an asynchronous operation on that file descriptor. In the example, **`stream`** is linked to the file descriptor `STDOUT_FILENO` to write a string asynchronously to the standard output stream.

Chapter 33

Boost.Interprocess

Interprocess communication describes mechanisms to exchange data between programs running on the same computer. It does not include network communication. To exchange data between programs running on different computers connected through a network, see Chapter 32, which covers Boost.Asio.
This chapter presents the library Boost.Interprocess, which contains numerous classes that abstract operating system specific interfaces for interprocess communication. Even though the concepts of interprocess communication are similar between different operating systems, the interfaces can vary greatly. Boost.Interprocess provides platform-independent access.
While Boost.Asio can be used to exchange data between processes running on the same computer, Boost.Interprocess usually provides better performance. Boost.Interprocess calls operating system functions optimized for data exchange between processes running on the same computer and thus should be the first choice to exchange data without a network.

33.1 Shared Memory

Shared memory is typically the fastest form of interprocess communication. It provides a memory area that is shared between processes. One process can write data to the area and another process can read it.
In Boost.Interprocess the class `boost::interprocess::shared_memory_object` is used to represent shared memory. Include the header file `boost/interprocess/shared_memory_object.hpp` to use this class.

Example 33.1 Creating shared memory

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object shdmem{open_or_create, "Boost", read_write};
  shdmem.truncate(1024);
  std::cout << shdmem.get_name() << '\n';
  offset_t size;
  if (shdmem.get_size(size))
    std::cout << size << '\n';
}
```

The constructor of `boost::interprocess::shared_memory_object` expects three parameters. The first parameter specifies whether the shared memory should be created or just opened. Example 33.1 handles both cases. `boost::interprocess::open_or_create` will open shared memory if it already exists or create shared memory if it doesn't.
Opening existing shared memory assumes that it has been created before. To uniquely identify shared memory, a name is assigned. That name is specified by the second parameter passed to the constructor of `boost::interprocess::shared_memory_object`.

The third parameter determines how a process can access shared memory. In Example 33.1, `boost::interprocess::read_write` says the process has read-write access.
After creating an object of type `boost::interprocess::shared_memory_object`, a corresponding shared memory block will exist within the operating system. The size of this memory area is initially 0. To use the area, call `truncate()`, passing in the size of the shared memory in bytes. In Example 33.1, the shared memory provides space for 1,024 bytes. `truncate()` can only be called if the shared memory has been opened with `boost::interprocess::read_write`. If not, an exception of type `boost::interprocess::interprocess_exception` is thrown. `truncate()` can be called repeatedly to adjust the size of the shared memory.
After creating shared memory, member functions such as `get_name()` and `get_size()` can be used to query the name and the size of the shared memory.
Because shared memory is used to exchange data between different processes, each process needs to map the shared memory into its address space. The class `boost::interprocess::mapped_region` is used to do this. It may come as a surprise that two classes (`boost::interprocess::shared_memory_object` and `boost::interprocess::mapped_region`) are needed to access shared memory. This is done so that the class `boost::interprocess::mapped_region` can also be used to map other objects into the address space of a process.

Example 33.2 Mapping shared memory into the address space of a process

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object shdmem{open_or_create, "Boost", read_write};
  shdmem.truncate(1024);
  mapped_region region{shdmem, read_write};
  std::cout << std::hex << region.get_address() << '\n';
  std::cout << std::dec << region.get_size() << '\n';
  mapped_region region2{shdmem, read_only};
  std::cout << std::hex << region2.get_address() << '\n';
  std::cout << std::dec << region2.get_size() << '\n';
}
```

To use the class `boost::interprocess::mapped_region`, include the header file `boost/interprocess/mapped_region.hpp`. An object of type `boost::interprocess::shared_memory_object` must be passed as the first parameter to the constructor of `boost::interprocess::mapped_region`. The second parameter determines whether access to the memory area is read-only or read-write.
Example 33.2 creates two objects of type `boost::interprocess::mapped_region`. The shared memory named Boost is mapped twice into the address space of the process. The address and the size of the mapped memory area is written to standard output using the member functions `get_address()` and `get_size()`. `get_size()` returns 1024 in both cases, but the return value of `get_address()` is different for each object.

> Note
>
> Example 33.2, and some of the examples that follow, will cause a compiler error with Visual C++ 2013 and Boost 1.55.0. The bug is described in ticket 9332. This bug has been fixed in Boost 1.56.0.

Example 33.3 Writing and reading a number in shared memory

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object shdmem{open_or_create, "Boost", read_write};
  shdmem.truncate(1024);
  mapped_region region{shdmem, read_write};
  int *i1 = static_cast<int*>(region.get_address());
  *i1 = 99;
  mapped_region region2{shdmem, read_only};
  int *i2 = static_cast<int*>(region2.get_address());
  std::cout << *i2 << '\n';
}
```

Example 33.3 uses the mapped memory area to write and read a number. **`region`** writes the number 99 to the beginning of the shared memory. **`region2`** then reads the same location in shared memory and writes the number to the standard output stream. Even though **`region`** and **`region2`** represent different memory areas within the process, as seen by the return values of `get_address()` in the previous example, the program prints `99` because both memory areas access the same underlying shared memory.

Example 33.4 Deleting shared memory

```
#include <boost/interprocess/shared_memory_object.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  bool removed = shared_memory_object::remove("Boost");
  std::cout << std::boolalpha << removed << '\n';
}
```

To delete shared memory, `boost::interprocess::shared_memory_object` offers the static member function `remove()`, which takes the name of the shared memory to be deleted as a parameter (see Example 33.4).
Boost.Interprocess partially supports the RAII idiom through a class called `boost::interprocess::remove_shared_memory_on_destroy`. Its constructor expects the name of an existing shared memory. If an object of this class is destroyed, the shared memory is automatically deleted in the destructor.
The constructor of `boost::interprocess::remove_shared_memory_on_destroy` does not create or open the shared memory. Therefore, this class is not a typical representative of the RAII idiom.
If `remove()` is never called, the shared memory continues to exist even if the program terminates. Whether or not the shared memory is automatically deleted depends on the underlying operating system. Windows and many Unix operating systems, including Linux, automatically delete shared memory once the system is restarted.
Windows provides a special kind of shared memory that is automatically deleted once the last process using it has been terminated. Access the class `boost::interprocess::windows_shared_memory`, which is defined in `boost/interprocess/windows_shared_memory.hpp`, to use this kind of shared memory (see Example 33.5).

Example 33.5 Using Windows-specific shared memory

```
#include <boost/interprocess/windows_shared_memory.hpp>
#include <boost/interprocess/mapped_region.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  windows_shared_memory shdmem{open_or_create, "Boost", read_write, 1024};
  mapped_region region{shdmem, read_write};
  int *i1 = static_cast<int*>(region.get_address());
  *i1 = 99;
  mapped_region region2{shdmem, read_only};
  int *i2 = static_cast<int*>(region2.get_address());
  std::cout << *i2 << '\n';
}
```

`boost::interprocess::windows_shared_memory` does not provide a member function `truncate()`. Instead, the size of the shared memory needs to be passed as the fourth parameter to the constructor.
Even though the class `boost::interprocess::windows_shared_memory` is not portable and can only be used on Windows, it is useful when data needs to be exchanged with an existing Windows program that uses this special kind of shared memory.

33.2 Managed Shared Memory

The previous section introduced the class `boost::interprocess::shared_memory_object`, which can be used to create and manage shared memory. In practice, this class is rarely used because it requires the program to read and write individual bytes from and to the shared memory. C++ style favors creating objects of classes and hiding the specifics of where and how data is stored in memory.
Boost.Interprocess provides `boost::interprocess::managed_shared_memory`, a class that is defined in `boost/interprocess/managed_shared_memory.hpp`, to support *managed shared memory*. This class lets you instantiate objects that have their memory located in shared memory, making the objects automatically available to any program that accesses the same shared memory.

Example 33.6 Using managed shared memory

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  int *i = managed_shm.construct<int>("Integer")(99);
  std::cout << *i << '\n';
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
  if (p.first)
    std::cout << *p.first << '\n';
}
```

Example 33.6 opens the shared memory named Boost with a size of 1,024 bytes. If the shared memory does not exist, it will be automatically created.
In regular shared memory, individual bytes are directly accessed to read or write data. Managed shared memory uses member functions such as `construct()`, which expects a type as a template parameter (in Example 33.6, int). The member function expects a name to denote the object created in the managed shared memory. Example 33.6 uses the name Integer.
Because `construct()` returns a proxy object, parameters can be passed to it to initialize the created object. The syntax looks like a call to a constructor. This ensures that objects can be created and initialized in managed shared memory.
To access a particular object in managed shared memory, the member function `find()` is used. By passing the name of the object to find, `find()` returns either a pointer to the object, or in case no object with the given name was found, 0.
As seen in Example 33.6, `find()` returns an object of type `std::pair`. The pointer to the object is provided as the member variable **`first`**. Example 33.7 shows what is received in **`second`**.

Example 33.7 Creating arrays in managed shared memory

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  int *i = managed_shm.construct<int>("Integer")[10](99);
  std::cout << *i << '\n';
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
  if (p.first)
  {
    std::cout << *p.first << '\n';
    std::cout << p.second << '\n';
  }
}
```

In Example 33.7, an array with ten elements of type int is created by providing the value 10 enclosed by square brackets after the call to `construct()`. The same `10` is written to the standard output stream using the member variable **`second`**. Thanks to this member variable, you can tell whether objects returned by `find()` are single objects or arrays. For the former, **`second`** is set to 1, while for the latter, **`second`** is set to the number of elements in the array.

Please note that all ten elements in the array are initialized with the value 99. If you want to initialize elements with different values, pass an iterator.

`construct()` will fail if an object already exists with the given name in the managed shared memory. In this case, `construct()` returns 0 and no initialization occurs. To use an existing object, use the member function `find_or_construct()`, which returns a pointer to an existing object or creates a new one.

There are other cases that will cause `construct()` to fail. Example 33.8 tries to create an array of type int with 4,096 elements. The managed shared memory, however, only contains 1,024 bytes. This causes an exception of type `boost::interprocess::bad_alloc` to be thrown.

Once objects have been created in a managed shared memory, they can be deleted with the member function `destroy()`.

Example 33.8 An exception of type `boost::interprocess::bad_alloc`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  try
  {
    shared_memory_object::remove("Boost");
    managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
    int *i = managed_shm.construct<int>("Integer")[4096](99);
  }
  catch (boost::interprocess::bad_alloc &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

In Example 33.9, the name of the object to be deleted is passed as the only parameter to `destroy()`. The return value of type bool can be checked to verify whether the given object was found and deleted successfully. Because an object will always be deleted if found, a return value of `false` indicates that no object with the given name was found.

The member function `destroy_ptr()` can be used to pass a pointer to an object in the managed shared memory. It can also be used to delete arrays.

Example 33.9 Removing objects in shared memory

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")(99);
  std::cout << *i << '\n';
  managed_shm.destroy<int>("Integer");
  std::pair<int*, std::size_t> p = managed_shm.find<int>("Integer");
  std::cout << p.first << '\n';
}
```

Because managed shared memory makes it fairly easy to share objects between processes, it seems natural to use containers from the standard library as well. However, these containers allocate memory using `new`. In order to use these containers in managed shared memory, they need to be told to allocate memory in the shared memory. Many implementations of the standard library are not flexible enough to use containers such as `std::string` or `std::list` with Boost.Interprocess. This includes the implementations shipped with Visual C++ 2013, GCC and Clang.

To allow developers to use the containers from the standard library, Boost.Interprocess provides a more flexible implementation in the namespace `boost::interprocess`. For example, `boost::interprocess::string` acts exactly like its C++ counterpart `std::string`, except that strings can be safely stored in a managed shared memory (see Example 33.10).

Example 33.10 Putting strings into shared memory

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/containers/string.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  typedef allocator<char,
    managed_shared_memory::segment_manager> CharAllocator;
  typedef basic_string<char, std::char_traits<char>, CharAllocator> string;
  string *s = managed_shm.find_or_construct<string>("String")("Hello!",
    managed_shm.get_segment_manager());
  s->insert(5, ", world");
  std::cout << *s << '\n';
}
```

To create a string that will allocate memory in the same managed shared memory it resides in, a corresponding type must be defined. The new string type must use an allocator provided by Boost.Interprocess instead of the default allocator provided by the standard.

For this purpose, Boost.Interprocess provides the class `boost::interprocess::allocator`, which is defined in `boost/interprocess/allocators/allocator.hpp`. With this class, an allocator can be created that internally uses the *segment manager* of the managed shared memory. The segment manager is responsible for managing the memory within a managed shared memory block. Using the newly created allocator, a corresponding type for the string can be defined. As indicated above, use `boost::interprocess::basic_string` instead of `std::basic_string`. The new type – called `string` in Example 33.10 – is based on `boost::interprocess::basic_string` and accesses the segment manager via its allocator. To let the particular instance of `string` created by a call to `find_or_construct()` know which segment manager it should access, pass a pointer to the corresponding segment manager as the second parameter to the constructor.

Boost.Interprocess provides implementations for many other containers from the standard library. For example, `boost::interprocess::vector` and `boost::interprocess::map` are defined in `boost/interprocess/containers/vector.hpp` and `boost/interprocess/containers/map.hpp`, respectively.

Please note that the containers from Boost.Container support Boost.Interprocess and can be put into shared memory. They can be used instead of containers from `boost::interprocess`. Boost.Container is introduced in Chapter 20.

Whenever the same managed shared memory is accessed from different processes, operations such as creating, finding, and destroying objects are automatically synchronized. If two programs try to create objects with different names in the managed shared memory, the access is serialized accordingly. To execute multiple operations at one time without being interrupted by operations from a different process, use the member function `atomic_func()` (see Example 33.11).

Example 33.11 Atomic access on a managed shared memory

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <functional>
#include <iostream>

using namespace boost::interprocess;

void construct_objects(managed_shared_memory &managed_shm)
{
  managed_shm.construct<int>("Integer")(99);
  managed_shm.construct<float>("Float")(3.14);
}

int main()
{
  shared_memory_object::remove("Boost");
  managed_shared_memory managed_shm{open_or_create, "Boost", 1024};
  auto atomic_construct = std::bind(construct_objects,
    std::ref(managed_shm));
  managed_shm.atomic_func(atomic_construct);
  std::cout << *managed_shm.find<int>("Integer").first << '\n';
  std::cout << *managed_shm.find<float>("Float").first << '\n';
}
```

`atomic_func()` expects as its single parameter a function that takes no parameters and has no return value. The passed function will be called in a fashion that ensures exclusive access to the managed shared memory. However, exclusive access is only ensured if all other processes that access the managed shared memory also use `atomic_func()`. If another process has a pointer to an object within the managed shared memory, it could access and modify this object using its pointer.
Boost.Interprocess can also be used to synchronize object access. Since Boost.Interprocess does not know who can access individual objects at a particular time, synchronization needs to be explicitly handled. The following section introduces the classes provided for synchronization.

33.3 Synchronization

Boost.Interprocess allows multiple processes to use shared memory concurrently. Because shared memory is, by definition, shared between processes, Boost.Interprocess needs to support some kind of synchronization.
Thinking about synchronization, classes from the C++11 standard library or Boost.Thread come to mind. But these classes can only be used to synchronize threads within the same process; they do not support synchronization of different processes. However, since the challenge in both cases is the same, the concepts are also the same.
While synchronization objects such as mutexes and condition variables reside in the same address space in multithreaded applications, and therefore are available to all threads, the challenge with shared memory is that independent processes need to share these objects. For example, if one process creates a mutex, it somehow needs to be accessible from a different process.
Boost.Interprocess provides two kinds of synchronization objects: anonymous objects are directly stored in the shared memory, which makes them automatically available to all processes. Named objects are managed by the operating system, are not stored in the shared memory, and can be referenced from programs by name.

Example 33.12 Using a named mutex with `boost::interprocess::named_mutex`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")();
  named_mutex named_mtx{open_or_create, "mtx"};
  named_mtx.lock();
  ++(*i);
  std::cout << *i << '\n';
  named_mtx.unlock();
}
```

Example 33.12 creates and uses a named mutex using the class `boost::interprocess::named_mutex`, which is defined in `boost/interprocess/sync/named_mutex.hpp`.

The constructor of `boost::interprocess::named_mutex` expects a parameter specifying whether the mutex should be created or opened and a name for the mutex. Every process that knows the name can open the same mutex. To access the data in shared memory, the program needs to take ownership of the mutex by calling the member function `lock()`. Because mutexes can only be owned by one process at a time, another process may need to wait until the mutex has been released by `unlock()`. Once a process takes ownership of a mutex, it has exclusive access to the resource the mutex guards. In Example 33.12, the resource is a variable of type int that is incremented and written to the standard output stream.

If the sample program is started multiple times, each instance will print a value incremented by 1 compared to the previous value. Thanks to the mutex, access to the shared memory and the variable itself is synchronized between different processes.

Example 33.13 uses an anonymous mutex of type `boost::interprocess::interprocess_mutex`, which is defined in `boost/interprocess/sync/interprocess_mutex.hpp`. In order for the mutex to be accessible for all processes, it is stored in the shared memory.

Example 33.13 Using an anonymous mutex with `interprocess_mutex`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")();
  interprocess_mutex *mtx =
    managed_shm.find_or_construct<interprocess_mutex>("mtx")();
  mtx->lock();
  ++(*i);
  std::cout << *i << '\n';
  mtx->unlock();
}
```

Example 33.13 behaves exactly like the previous one. The only difference is the mutex, which is now stored directly in shared memory. This can be done with the member functions `construct()` or `find_or_construct()` from the class `boost::interprocess::managed_shared_memory`.

In addition to `lock()`, both `boost::interprocess::named_mutex` and `boost::interprocess::interprocess_mutex` provide the member functions `try_lock()` and `timed_lock()`. They behave exactly like their counterparts in the standard library and Boost.Thread. If recursive mutexes are required, Boost.Interprocess provides two classes: `boost::interprocess::named_recursive_mutex` and `boost::interprocess::interprocess_recursive_mutex`.

While mutexes guarantee exclusive access to a shared resource, *condition variables* control who has exclusive access at what time. In general, the condition variables provided by Boost.Interprocess work like the ones provided by the C++11 standard library and Boost.Thread. They have similar interfaces, which makes users of these libraries feel immediately at home when using these variables in Boost.Interprocess.

Example 33.14 Using a named condition with `boost::interprocess::named_condition`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/named_mutex.hpp>
#include <boost/interprocess/sync/named_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>
```

```
using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")(0);
  named_mutex named_mtx{open_or_create, "mtx"};
  named_condition named_cnd{open_or_create, "cnd"};
  scoped_lock<named_mutex> lock{named_mtx};
  while (*i < 10)
  {
    if (*i % 2 == 0)
    {
      ++(*i);
      named_cnd.notify_all();
      named_cnd.wait(lock);
    }
    else
    {
      std::cout << *i << std::endl;
      ++(*i);
      named_cnd.notify_all();
      named_cnd.wait(lock);
    }
  }
  named_cnd.notify_all();
  shared_memory_object::remove("shm");
  named_mutex::remove("mtx");
  named_condition::remove("cnd");
}
```

Example 33.14 uses a condition variable of type `boost::interprocess::named_condition`, which is defined in `boost/interprocess/sync/named_condition.hpp`. Because it is a named variable, it does not need to be stored in shared memory.
The application uses a `while` loop to increment a variable of type int, which is stored in shared memory. Although the variable is incremented with each iteration of the loop, it will only be written to the standard output stream with every second iteration – only odd numbers are written.
Every time the variable is incremented by 1, the member function `wait()` of the condition variable **`named_cnd`** is called. A *lock* – in Example 33.14, the variable named **`lock`** – is passed to this member function. This is based on the RAII idiom of taking ownership of a mutex inside the constructor and releasing it inside the destructor.
The lock is created before the `while` loop and takes ownership of the mutex for the entire execution of the program. However, if passed to `wait()` as a parameter, the lock is automatically released.
Condition variables are used to wait for a signal indicating that the wait is over. Synchronization is controlled by the member functions `wait()` and `notify_all()`. When a program calls `wait()`, ownership of the corresponding mutex is released. The program then waits until `notify_all()` is called on the same condition variable.
When started, Example 33.14 does not seem to do much. After the variable **`i`** is incremented from 0 to 1 within the `while` loop, the program waits for a signal by calling `wait()`. In order to fire the signal, a second instance of the program needs to be started.
The second instance tries to take ownership of the same mutex before entering the `while` loop. This succeeds since the first instance released the mutex by calling `wait()`. Because the variable has been incremented once, the second instance executes the `else` branch of the `if` expression and writes the current value to the standard output stream. Then the value is incremented by 1.
Now the second instance also calls `wait()`. However, before it does, it calls `notify_all()`, which ensures that the two instances cooperate correctly. The first instance is notified and tries to take ownership of the mutex again, which is still owned by the second instance. However, because the second instance calls `wait()` right after calling `notify_all()`, which automatically releases ownership, the first instance will take ownership at that point.
Both instances alternate, incrementing the variable in the shared memory. However, only one instance writes the value to the standard output stream. As soon as the variable reaches the value 10, the `while` loop is finished. In

order to avoid having the other instance wait for a signal forever, `notify_all()` is called one more time after the loop. Before terminating, the shared memory, the mutex, and the condition variable are destroyed.
Just as there are two types of mutexes – an anonymous type that must be stored in shared memory and a named type – there are also two types of condition variables. Example 33.15 is a rewrite of the previous example using an anonymous condition variable.

Example 33.15 Using an anonymous condition with `interprocess_condition`

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/interprocess_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>
#include <iostream>

using namespace boost::interprocess;

int main()
{
  managed_shared_memory managed_shm{open_or_create, "shm", 1024};
  int *i = managed_shm.find_or_construct<int>("Integer")(0);
  interprocess_mutex *mtx =
    managed_shm.find_or_construct<interprocess_mutex>("mtx")();
  interprocess_condition *cnd =
    managed_shm.find_or_construct<interprocess_condition>("cnd")();
  scoped_lock<interprocess_mutex> lock{*mtx};
  while (*i < 10)
  {
    if (*i % 2 == 0)
    {
      ++(*i);
      cnd->notify_all();
      cnd->wait(lock);
    }
    else
    {
      std::cout << *i << std::endl;
      ++(*i);
      cnd->notify_all();
      cnd->wait(lock);
    }
  }
  cnd->notify_all();
  shared_memory_object::remove("shm");
}
```

Example 33.15 works exactly like the previous one and also needs to be started twice to increment the int variable ten times.
Besides mutexes and condition variables, Boost.Interprocess also supports *semaphores* and *file locks*. Semaphores are similar to condition variables except they do not distinguish between two states; instead, they are based on a counter. File locks behave like mutexes, except they are used with files on a hard drive, rather than objects in memory.
In the same way that the C++11 standard library and Boost.Thread distinguish between different types of mutexes and locks, Boost.Interprocess provides several mutexes and locks. For example, mutexes can be owned exclusively or non-exclusively. This is helpful if multiple processes need to read data simultaneously since an exclusive mutex is only required to write data. Different classes for locks are available to apply the RAII idiom to individual mutexes.
Names should be unique unless anonymous synchronization objects are used. Even though mutexes and condition variables are objects based on different classes, this may not necessarily hold true for the operating system dependent interfaces wrapped by Boost.Interprocess. On Windows, the same operating system functions are used for both mutexes and condition variables. If the same name is used for two objects, one of each type, the program will not behave correctly on Windows.

Part VII

Streams and Files

The following libraries facilitate working with streams and files.

- Boost.IOStreams provides streams that go far beyond what the standard library offers. Boost.IOStreams gives you access to more streams to read and write data from and to many different sources and sinks. In addition, filters enable a variety of operations such as compressing/uncompressing data while reading or writing.
- Boost.Filesystem provides access to the filesystem. With Boost.Filesystem you can, for example, copy files or iterate over files in a directory.

Chapter 34

Boost.IOStreams

This chapter introduces the library Boost.IOStreams. Boost.IOStreams breaks up the well-known streams from the standard library into smaller components. The library defines two concepts: *device*, which describes data sources and sinks, and *stream*, which describes an interface for formatted input/output based on the interface from the standard library. A stream defined by Boost.IOStreams isn't automatically connected to a data source or sink.

Boost.IOStreams provides numerous implementations of the two concepts. For example, there is the device `boost::iostreams::mapped_file`, which loads a file partially or completely into memory. The stream `boost::iostreams::stream` can be connected to a device like `boost::iostreams::mapped_file` to use the familiar stream operators `operator<<` and `operator>>` to read and write data.

In addition to `boost::iostreams::stream`, Boost.IOStreams provides the stream `boost::iostreams::filtering_stream`, which lets you add data filters. For example, you can use `boost::iostreams::gzip_compressor` to write data compressed in the GZIP format.

Boost.IOStreams can also be used to connect to platform-specific objects. The library provides devices to connect to a Windows handle or a file descriptor. That way objects from low-level APIs can be made available in platform-independent C++ code.

The classes and functions provided by Boost.IOStreams are defined in the namespace `boost::iostreams`. There is no master header file. Because Boost.IOStreams contains more than header files, it must be prebuilt. This can be important because, depending on how Boost.IOStreams has been prebuilt, support for some features could be missing.

34.1 Devices

Devices are classes that provide read and write access to objects that are usually outside of a process – for example, files. However, you can also access internal objects, such as arrays, as devices.

A device is nothing more than a class with the member function `read()` or `write()`. A device can be connected with a stream so you can read and write formatted data rather than using the `read()` and `write()` member functions directly.

Example 34.1 Using an array as a device with `boost::iostreams::array_sink`

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/stream.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  stream<array_sink> os{sink};
  os << "Boost" << std::flush;
  std::cout.write(buffer, 5);
}
```

Example 34.1 uses the device boost::iostreams::array_sink to write data to an array. The array is passed as a parameter to the constructor. Afterwards, the device is connected with a stream of type boost::iostreams::stream. A reference to the device is passed to the constructor of boost::iostreams::stream, and the type of the device is passed as a template parameter to boost::iostreams::stream.
The example uses the operator operator<< to write “Boost” to the stream. The stream forwards the data to the device. Because the device is connected to the array, “Boost” is stored in the first five elements of the array. Since the array’s contents are written to standard output, Boost will be displayed when you run the example.

Example 34.2 Using an array as a device with boost::iostreams::array_source

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/stream.hpp>
#include <string>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  stream<array_sink> os{sink};
  os << "Boost" << std::endl;

  array_source source{buffer};
  stream<array_source> is{source};
  std::string s;
  is >> s;
  std::cout << s << '\n';
}
```

Example 34.2 is based on the previous example. The string written with boost::iostreams::array_sink to an array is read with boost::iostreams::array_source.
boost::iostreams::array_source is used like boost::iostreams::array_sink. While boost::iostreams::array_sink supports only write operations, boost::iostreams::array_source supports only read. boost::iostreams::array supports both write and read operations.
Please note that boost::iostreams::array_source and boost::iostreams::array_sink receive a reference to an array. The array must not be destroyed while the devices are still in use.
Example 34.3 uses a device of type boost::iostreams::back_insert_device, instead of boost::iostreams::array_sink. This device can be used to write data to any container that provides the member function insert(). The device calls this member function to forward data to the container.
The example uses boost::iostreams::back_insert_device to write “Boost” to a vector. Afterwards, “Boost” is read from boost::iostreams::array_source. The address of the beginning of the vector and the size are passed as parameters to the constructor of boost::iostreams::array_source.
Example 34.3 displays Boost.

Example 34.3 Using a vector as a device with boost::iostreams::back_insert_device

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/device/back_inserter.hpp>
#include <boost/iostreams/stream.hpp>
#include <vector>
#include <string>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  std::vector<char> v;
  back_insert_device<std::vector<char>> sink{v};
  stream<back_insert_device<std::vector<char>>> os{sink};
  os << "Boost" << std::endl;
  array_source source{v.data(), v.size()};
  stream<array_source> is{source};
  std::string s;
  is >> s;
  std::cout << s << '\n';
}
```

Example 34.4 uses the device `boost::iostreams::file_source` to read files. While the previously introduced devices don't provide member functions, `boost::iostreams::file_source` provides `is_open()` to test whether a file was opened successfully. It also provides the member function `close()` to explicitly close a file. You don't need to call `close()` because the destructor of `boost::iostreams::file_source` closes a file automatically.

Example 34.4 Using a file as a device with `boost::iostreams::file_source`

```
#include <boost/iostreams/device/file.hpp>
#include <boost/iostreams/stream.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  file_source f{"main.cpp"};
  if (f.is_open())
  {
    stream<file_source> is{f};
    std::cout << is.rdbuf() << '\n';
    f.close();
  }
}
```

Besides `boost::iostreams::file_source`, Boost.IOStreams also provides the device `boost::iostreams::mapped_file_source` to load a file partially or completely into memory. `boost::iostreams::mapped_file_source` defines a member function `data()` to receive a pointer to the respective memory area. That way, data can be randomly accessed in a file without having to read the file sequentially.
For write access to files, Boost.IOStreams provides the devices `boost::iostreams::file_sink` and `boost::iostreams::mapped_file_sink`.

Example 34.5 Using `file_descriptor_source` and `file_descriptor_sink`

```
#include <boost/iostreams/device/file_descriptor.hpp>
#include <boost/iostreams/stream.hpp>
#include <iostream>
#include <Windows.h>

using namespace boost::iostreams;

int main()
{
  HANDLE handles[2];
  if (CreatePipe(&handles[0], &handles[1], nullptr, 0))
  {
    file_descriptor_source src{handles[0], close_handle};
    stream<file_descriptor_source> is{src};

    file_descriptor_sink snk{handles[1], close_handle};
    stream<file_descriptor_sink> os{snk};

    os << "Boost" << std::endl;
    std::string s;
    std::getline(is, s);
    std::cout << s;
  }
}
```

Example 34.5 uses the devices boost::iostreams::file_descriptor_source and boost::iostreams::file_descriptor_sink. These devices support read and write operations on platform-specific objects. On Windows these objects are handles, and on POSIX operating systems they are file descriptors.
Example 34.5 calls the Windows function CreatePipe() to create a pipe. The read and write ends of the pipe are received in the array **handles**. The read end of the pipe is passed to the device boost::iostreams::file_descriptor_source, and the write end is passed to the device boost::iostreams::file_descriptor_sink. Everything written to the stream **os**, which is connected to the write end, can be read from the stream **is**, which is connected to the read end. Example 34.5 sends “Boost” through the pipe and to standard output.
The constructors of boost::iostreams::file_descriptor_source and boost::iostreams::file_descriptor_sink expect two parameters. The first parameter is a Windows handle or – if the program is run on a POSIX system – a file descriptor. The second parameter must be either **boost::iostreams::close_handle** or **boost::iostreams::never_close_handle**. This parameter specifies whether or not the destructor closes the Windows handle or file descriptor.

34.2 Filters

Besides devices, Boost.IOStreams also provides filters, which operate in front of devices to filter data read from or written to devices. The following examples use boost::iostreams::filtering_istream and boost::iostreams::filtering_ostream. They replace boost::iostreams::stream, which doesn’t support filters.

Example 34.6 Using boost::iostreams::regex_filter

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/regex.hpp>
#include <boost/regex.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  filtering_ostream os;
  os.push(regex_filter{boost::regex{"Bo+st"}, "C++"});
  os.push(sink);
  os << "Boost" << std::flush;
  os.pop();
  std::cout.write(buffer, 3);
}
```

Example 34.6 uses the device boost::iostreams::array_sink to write data to an array. The data is sent through a filter of type boost::iostreams::regex_filter, which replaces characters. The filter expects a regular expression and a format string. The regular expression describes what to replace. The format string specifies what the characters should be replaced with. The example replaces “Boost” with “C++”. The filter will match one or more consecutive instances of the letter “o” in “Boost,” but not zero instances.
The filter and the device are connected with the stream boost::iostreams::filtering_ostream. This class provides a member function push(), which the filter and the device are passed to.
The filter(s) must be passed before the device; the order is important. You can pass one or more filters, but once a device has been passed, the stream is complete, and you must not call push() again.
The filter boost::iostreams::regex_filter can’t process data character by character because regular expressions need to look at character groups. That’s why boost::iostreams::regex_filter starts filtering only after a write operation is complete and all data is available. This happens when the device is removed from the stream with the member function pop(). Example 34.6 calls pop() after “Boost” has been written to the stream. Without the call to pop(), boost::iostreams::regex_filter won’t process any data and won’t forward data to the device.
Please note that you must not use a stream that isn’t connected with a device. However, you can complete a stream if you add a device with push() after a call to pop().

Example 34.6 displays C++.

Example 34.7 Accessing filters in boost::iostreams::filtering_ostream

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/counter.hpp>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  char buffer[16];
  array_sink sink{buffer};
  filtering_ostream os;
  os.push(counter{});
  os.push(sink);
  os << "Boost" << std::flush;
  os.pop();
  counter *c = os.component<counter>(0);
  std::cout << c->characters() << '\n';
  std::cout << c->lines() << '\n';
}
```

Example 34.7 uses the filter boost::iostreams::counter, which counts characters and lines. This class provides the member functions characters() and lines().

boost::iostreams::filtering_stream provides the member function component() to access a filter. The index of the respective filter must be passed as a parameter. Because component() is a template, the type of the filter must be passed as a template parameter. component() returns a pointer to the filter. It returns 0 if an incorrect filter type is passed as a template parameter.

Example 34.7 writes five characters to the stream. It does not write a newline ("\n"). Thus, the example displays 5 and 0.

Example 34.8 Writing and reading data compressed with ZLIB

```
#include <boost/iostreams/device/array.hpp>
#include <boost/iostreams/device/back_inserter.hpp>
#include <boost/iostreams/filtering_stream.hpp>
#include <boost/iostreams/filter/zlib.hpp>
#include <vector>
#include <string>
#include <iostream>

using namespace boost::iostreams;

int main()
{
  std::vector<char> v;
  back_insert_device<std::vector<char>> snk{v};
  filtering_ostream os;
  os.push(zlib_compressor{});
  os.push(snk);
  os << "Boost" << std::flush;
  os.pop();

  array_source src{v.data(), v.size()};
  filtering_istream is;
  is.push(zlib_decompressor{});
  is.push(src);
  std::string s;
  is >> s;
  std::cout << s << '\n';
}
```

Example 34.8 uses the stream `boost::iostreams::filtering_istream` in addition to `boost::iostreams::filtering_ostream`. This stream is used when you want to read data with filters. In the example, compressed data is written and read again.
Boost.IOStreams provides several data compression filters. The class `boost::iostreams::zlib_compressor` compresses data in the ZLIB format. To uncompress data in the ZLIB format, use the class `boost::iostreams::zlib_decompressor`. These filters are added to the streams using `push()`.
Example 34.8 writes "Boost" to the vector **`v`** in compressed form and to the string **`s`** in uncompressed form. The example displays `Boost`.

Note

Please note that on Windows, the Boost.IOStreams prebuilt library doesn't support data compression because, by default, the library is built with the macro `NO_ZLIB` on Windows. You must undefine this macro, define `ZLIB_LIBPATH` and `ZLIB_SOURCE`, and rebuild to get ZLIB support on Windows.

Chapter 35

Boost.Filesystem

The library Boost.Filesystem makes it easy to work with files and directories. It provides a class called `boost::filesystem::path` that processes paths. In addition, many free-standing functions are available to handle tasks like creating directories or checking whether a file exists.
Boost.Filesystem has been revised several times. This chapter introduces Boost.Filesystem 3, the current version. This version has been the default since the Boost C++ Libraries 1.46.0. Boost.Filesystem 2 was last shipped with version 1.49.0.

35.1 Paths

`boost::filesystem::path` is the central class in Boost.Filesystem for representing and processing paths. Definitions can be found in the namespace `boost::filesystem` and in the header file `boost/filesystem.hpp`. Paths can be built by passing a string to the constructor of `boost::filesystem::path` (see Example 35.1).

Example 35.1 Using `boost::filesystem::path`

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

using namespace boost::filesystem;

int main()
{
  path p1{"C:\\"};
  path p2{"C:\\Windows"};
  path p3{L"C:\\Boost C++ \u5E93"};
}
```

`boost::filesystem::path` can be initialized with wide strings. Wide strings are interpreted as Unicode and make it easy to create paths using characters from nearly any language. This is a crucial difference from Boost.Filesystem 2, which provided separate classes, such as `boost::filesystem::path` and `boost::filesystem::wpath`, for different string types.
Please note that Boost.Filesystem doesn't support `std::u16string` or `std::u32string`. Your compiler aborts with an error if you try to initialize `boost::filesystem::path` with one of these string types.
None of the constructors of `boost::filesystem::path` validate paths or check whether the given file or directory exists. Thus, `boost::filesystem::path` can be instantiated even with meaningless paths.

Example 35.2 Meaningless paths with `boost::filesystem::path`

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

using namespace boost::filesystem;

int main()
{
  path p1{"..."};
  path p2{"\\"};
  path p3{"@:"};
}
```

Example 35.2 runs without any problems because paths are just strings. `boost::filesystem::path` only processes strings; the file system is not accessed.
Because `boost::filesystem::path` processes strings, the class provides several member functions to retrieve a path as a string.
In general, Boost.Filesystem differentiates between *native paths* and *generic paths*. Native paths are operating system specific and must be used when calling operating system functions. Generic paths are portable and independent of the operating system.
The member functions `native()`, `string()` and `wstring()` all return paths in the native format. When run on Windows, Example 35.3 writes `C:\Windows\System` to the standard output stream three times.
The member functions `generic_string()` and `generic_wstring()` both return paths in a generic format. These are portable paths; the string is normalized based on rules of the POSIX standard. Generic paths are therefore identical to paths used on Linux. For example, the slash is used as the separator for directories. If Example 35.3 is run on Windows, both `generic_string()` and `generic_wstring()` write `C:/Windows/System` to the standard output stream.

Example 35.3 Retrieving paths from `boost::filesystem::path` as strings

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\System"};

#ifdef BOOST_WINDOWS_API
  std::wcout << p.native() << '\n';
#else
  std::cout << p.native() << '\n';
#endif

  std::cout << p.string() << '\n';
  std::wcout << p.wstring() << '\n';

  std::cout << p.generic_string() << '\n';
  std::wcout << p.generic_wstring() << '\n';
}
```

The return value of member functions returning native paths depends on the operating system the program is executed on. The return value of member functions returning generic paths is independent of the operating system. Generic paths uniquely identify files and directories independently from the operating system and therefore make it easy to write platform-independent code.
Because `boost::filesystem::path` can be initialized with different string types, several member functions are provided to retrieve paths in different string types. While `string()` and `generic_string()` return a string of type `std::string`, `wstring()` and `generic_wstring()` return a string of type `std::wstring`.
The return type of `native()` depends on the operating system the program was compiled for. On Windows a string of type `std::wstring` is returned. On Linux it is a string of type `std::string`.
The constructor of `boost::filesystem::path` supports both generic and platform-dependent paths. In Example 35.3, the path “C:\\Windows\\System” is Windows specific and not portable. Notice also that because the backslash is an escape character in C++, it must be escaped itself. This path will only be recognized correctly by Boost.Filesystem if the program is run on Windows. When executed on a POSIX compliant operating system such as Linux, this example will return `C:\Windows\System` for all member functions called. Since the backslash is not used as a separator on Linux in either the portable or the native format, Boost.Filesystem does not recognize this character as a separator for files and directories.
The macro `BOOST_WINDOWS_API` comes from Boost.System and is defined if the example is compiled on Windows. The respective macro for POSIX operating systems is called `BOOST_POSIX_API`.

Example 35.4 Initializing `boost::filesystem::path` with a portable path

```
#include <boost/filesystem.hpp>
#include <iostream>
```

```
using namespace boost::filesystem;

int main()
{
  path p{"/"};
  std::cout << p.string() << '\n';
  std::cout << p.generic_string() << '\n';
}
```

Example 35.4 uses a portable path to initialize `boost::filesystem::path`.
Because `generic_string()` returns a portable path, its value will be a slash ("/"), the same as was used to initialize `boost::filesystem::path`. However, the member function `string()` returns different values depending on the platform. On Windows and Linux it returns "/". The output is the same because Windows accepts the slash as a directory separator even though it prefers the backslash.

Example 35.5 Accessing components of a path

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\System"};
  std::cout << p.root_name() << '\n';
  std::cout << p.root_directory() << '\n';
  std::cout << p.root_path() << '\n';
  std::cout << p.relative_path() << '\n';
  std::cout << p.parent_path() << '\n';
  std::cout << p.filename() << '\n';
}
```

`boost::filesystem::path` provides several member functions to access certain components of a path. If Example 35.5 is executed on Windows, the string "C:\\Windows\\System" is interpreted as a platform-dependent path. Consequently, `root_name()` returns `"C:"`, `root_directory()` returns `"\"`, `root_path()` returns `"C:\"`, `relative_path()` returns `"Windows\System"`, `parent_path()` returns `"C:\Windows"`, and `filename()` returns `"System"`.
All member functions return platform-dependent paths because `boost::filesystem::path` stores paths in a platform-dependent format internally. To retrieve paths in a portable format, member functions such as `generic_string()` need to be called explicitly.
If Example 35.5 is executed on Linux, the returned values are different. Most of the member functions return an empty string, except `relative_path()` and `filename()`, which return `"C:\Windows\System"`. This means that the string "C:\\Windows\\System" is interpreted as a file name on Linux, which is understandable given that it is neither a portable encoding of a path nor a platform-dependent encoding on Linux. Therefore, Boost.Filesystem has no choice but to interpret it as a file name.
Boost.Filesystem provides additional member functions to verify whether a path contains a specific substring. These member functions are: `has_root_name()`, `has_root_directory()`, `has_root_path()`, `has_relative_path()`, `has_parent_path()`, and `has_filename()`. Each member function returns a value of type bool.
There are two more member functions that can split a file name into its components. They should only be called if `has_filename()` returns `true`. Otherwise, they will return empty strings because there is nothing to split if there is no file name.

Example 35.6 Receiving file name and file extension

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"photo.jpg"};
  std::cout << p.stem() << '\n';
  std::cout << p.extension() << '\n';
}
```

Example 35.6 returns "`photo`" for `stem()` and "`.jpg`" for `extension()`.
Instead of accessing the components of a path via member function calls, you can also iterate over the components.

Example 35.7 Iterating over components of a path

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\System"};
  for (const path &pp : p)
    std::cout << pp << '\n';
}
```

If executed on Windows, Example 35.7 will successively output "`C:`", "`/`", "`Windows`" and "`System`". On Linux, the output will be "`C:\Windows\System`".
While the previous examples introduced various member functions to access different components of a path, Example 35.8 uses a member function to modify a path.

Example 35.8 Concatenating paths with `operator/=`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\"};
  p /= "Windows\\System";
  std::cout << p.string() << '\n';
}
```

Using `operator/=`, Example 35.8 appends one path to another. On Windows, the program outputs `C:\Windows\System`. On Linux, the output is `C:\/Windows\System`, since the slash is used as a separator for files and directories. The slash is also the reason why `operator/=` has been overloaded; after all, the slash is part of the operator.
Besides `operator/=`, Boost.Filesystem provides the member functions `remove_filename()`, `replace_extension()`, `make_absolute()`, and `make_preferred()` to modify paths. The last member function mentioned is particularly designed for use on Windows.

Example 35.9 Preferred notation with `make_preferred()`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:/Windows/System"};
  std::cout << p.make_preferred() << '\n';
}
```

Even though the backslash is used as the separator for files and directories by default, Windows still accepts the slash. “C:/Windows/System” is therefore a valid native path. With `make_preferred()` such a path can be converted to the preferred notation on Windows. Example 35.9 displays "`C:\Windows\System`".
The member function `make_preferred()` has no effect on POSIX compliant operating systems such as Linux.

Please note that `make_preferred()` not only returns the converted path, but also modifies the object it has been called on. **p** contains “C:\Windows\System” after the call.

35.2 Files and Directories

The member functions presented with `boost::filesystem::path` simply process strings. They access individual components of a path, append paths to one another, and so on.
In order to work with physical files and directories on the hard drive, several free-standing functions are provided. They expect one or more parameters of type `boost::filesystem::path` and call operating system functions internally.
Prior to introducing the various functions, it is important to understand what happens in case of an error. All of the functions call operating system functions that may fail. Therefore, Boost.Filesystem provides two variants of the functions that behave differently in case of an error:

- The first variant throws an exception of type `boost::filesystem::filesystem_error`. This class is derived from `boost::system::system_error` and thus fits into the Boost.System framework.
- The second variant expects an object of type `boost::system::error_code` as an additional parameter. This object is passed by reference and can be examined after the function call. In case of a failure, the object stores the corresponding error code.

`boost::system::system_error` and `boost::system::error_code` are presented in Chapter 55. In addition to the inherited interface from `boost::system::system_error`, `boost::filesystem::filesystem_error` provides two member functions called `path1()` and `path2()`, both of which return an object of type `boost::filesystem::path`. Since there are functions that expect two parameters of type `boost::filesystem::path`, these two member functions provide an easy way to retrieve the corresponding paths in case of a failure.
Example 35.10 introduces `boost::filesystem::status()`, which queries the status of a file or directory. This function returns an object of type `boost::filesystem::file_status`, which can be passed to additional helper functions for evaluation. For example, `boost::filesystem::is_directory()` returns `true` if the status for a directory was queried. Besides `boost::filesystem::is_directory()`, other functions are available, including `boost::filesystem::is_regular_file()`, `boost::filesystem::is_symlink()`, and `boost::filesystem::exists()`, all of which return a value of type bool.

Example 35.10 Using `boost::filesystem::status()`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\"};
  try
  {
    file_status s = status(p);
    std::cout << std::boolalpha << is_directory(s) << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

The function `boost::filesystem::symlink_status()` queries the status of a symbolic link. With `boost::filesystem::status()` the status of the file referred to by the symbolic link is queried. On Windows, symbolic links are identified by the file extension `lnk`.

Example 35.11 Using `boost::filesystem::file_size()`

```
#include <boost/filesystem.hpp>
#include <iostream>
```

```
using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\win.ini"};
  boost::system::error_code ec;
  boost::uintmax_t filesize = file_size(p, ec);
  if (!ec)
    std::cout << filesize << '\n';
  else
    std::cout << ec << '\n';
}
```

A different category of functions makes it possible to query attributes. The function `boost::filesystem::file_size()` returns the size of a file in bytes. The return value is of type boost::uintmax_t, which is a type definition for unsigned long long. The type is provided by Boost.Integer.
Example 35.11 uses an object of type `boost::system::error_code`, which needs to be evaluated explicitly to determine whether the call to `boost::filesystem::file_size()` was successful.

Example 35.12 Using `boost::filesystem::last_write_time()`

```
#include <boost/filesystem.hpp>
#include <iostream>
#include <ctime>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Windows\\win.ini"};
  try
  {
    std::time_t t = last_write_time(p);
    std::cout << std::ctime(&t) << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

To determine the time a file was modified last, `boost::filesystem::last_write_time()` can be used (see Example 35.12).

Example 35.13 Using `boost::filesystem::space()`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\"};
  try
  {
    space_info s = space(p);
    std::cout << s.capacity << '\n';
    std::cout << s.free << '\n';
    std::cout << s.available << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

`boost::filesystem::space()` retrieves the total and remaining disk space (see Example 35.13). It returns an object of type `boost::filesystem::space_info`, which provides three public member variables: **`capacity`**, **`free`**, and **`available`**, all of type boost::uintmax_t. The disk space is in bytes.
While the functions presented so far leave files and directories untouched, there are several functions that can be used to create, rename, or delete files and directories.

Example 35.14 Creating, renaming, and deleting directories

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"C:\\Test"};
  try
  {
    if (create_directory(p))
    {
      rename(p, "C:\\Test2");
      boost::filesystem::remove("C:\\Test2");
    }
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Example 35.14 should be self-explanatory. Looking closely, one can see that it's not always an object of type `boost::filesystem::path` that is passed to functions, but rather a simple string. This is possible because `boost::filesystem::path` provides a non-explicit constructor that will convert strings to objects of type `boost::filesystem::path`. This makes it easy to use Boost.Filesystem since it's not required to create paths explicitly.

> Note
>
> In Example 35.14, `boost::filesystem::remove()` is explicitly called using its namespace. Otherwise, Visual C++ 2013 would confuse the function with `remove()` from the header file `stdio.h`.

Additional functions such as `create_symlink()` to create symbolic links or `copy_file()` and `copy_directory()` to copy files and directories are available as well.

Example 35.15 Using `boost::filesystem::absolute()`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  try
  {
    std::cout << absolute("photo.jpg") << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Example 35.15 presents a function that creates an absolute path based on a file name or section of a path. The path displayed depends on which directory the program is started in. For example, if the program was started in `C:\`, the output would be `"C:\photo.jpg"`.
To retrieve an absolute path relative to a different directory, a second parameter can be passed to `boost::filesystem::absolute()`.

Example 35.16 Creating an absolute path relative to another directory

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  try
  {
    std::cout << absolute("photo.jpg", "D:\\") << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Example 35.16 displays `"D:\photo.jpg"`.
The last example in this section, Example 35.17, introduces a useful helper function to retrieve the current working directory.

Example 35.17 Using `boost::filesystem::current_path()`

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  try
  {
    std::cout << current_path() << '\n';
    current_path("C:\\");
    std::cout << current_path() << '\n';
  }
  catch (filesystem_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Example 35.17 calls `boost::filesystem::current_path()` multiple times. If the function is called without parameters, the current working directory is returned. If an object of type `boost::filesystem::path` is passed, the current working directory is set.

35.3 Directory Iterators

Boost.Filesystem provides the iterator `boost::filesystem::directory_iterator` to iterate over files in a directory (see Example 35.18).

Example 35.18 Iterating over files in a directory

```
#include <boost/filesystem.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p = current_path();
  directory_iterator it{p};
  while (it != directory_iterator{})
    std::cout << *it++ << '\n';
}
```

`boost::filesystem::directory_iterator` is initialized with a path to retrieve an iterator pointing to the beginning of a directory. To retrieve the end of a directory, the class must be instantiated with the default constructor.
Entries can be created or deleted while iterating without invalidating the iterator. However, whether changes become visible during the iteration is undefined. For example, the iterator might not point to newly created files. To ensure that all current entries are accessible, restart the iteration.
To recursively iterate over a directory and subdirectories, Boost.Filesystem provides the iterator `boost::filesystem::recursive_directory_iterator`.

35.4 File Streams

The standard defines various file streams in the header file `fstream`. These streams do not accept parameters of type `boost::filesystem::path`. If you want to open file streams with objects of type `boost::filesystem::path`, include the header file `boost/filesystem/fstream.hpp`.

Example 35.19 Using `boost::filesystem::ofstream`

```
#include <boost/filesystem/fstream.hpp>
#include <iostream>

using namespace boost::filesystem;

int main()
{
  path p{"test.txt"};
  ofstream ofs{p};
  ofs << "Hello, world!\n";
}
```

Example 35.19 opens a file with the help of the class `boost::filesystem::ofstream`. An object of type `boost::filesystem::path` can be passed to the constructor of `boost::filesystem::ofstream`. The member function `open()` also accepts a parameter of type `boost::filesystem::path`.

Part VIII

Time

The following libraries process time values.

- Boost.DateTime defines classes for time points and periods – for both time of day and calendar dates – and functions to process them. For example, it is possible to iterate over dates.
- Boost.Chrono and Boost.Timer provide clocks to measure time. The clocks provided by Boost.Timer are specialized for measuring code execution time and are only used when optimizing code.

Chapter 36

Boost.DateTime

The library Boost.DateTime can be used to process time data such as calendar dates and times. In addition, Boost.DateTime provides extensions to account for time zones and supports formatted input and output of calendar dates and times. If you are looking for functions to get the current time or measure time, see Boost.Chrono in Chapter 37.

36.1 Calendar Dates

Boost.DateTime only supports calendar dates based on the *Gregorian calendar*, which in general is not a problem since this is the most widely used calendar. If you arrange a meeting with someone for May 12, 2014, you don't need to say that the date is based on the Gregorian calendar.
The Gregorian calendar was introduced by Pope Gregory XIII in 1582. Boost.DateTime supports calendar dates for the years 1400 to 9999, which means that support goes back before the year 1582. Thus, you can use Boost.DateTime for any calendar date after the year 1400 as long as that date is converted to the Gregorian calendar. To process earlier dates, Boost.DateTime has to be extended by a new calendar.
The header file `boost/date_time/gregorian/gregorian.hpp` contains definitions for all classes and functions that process calendar dates. These functions and classes can be found in the namespace `boost::gregorian`. To create a date, use the class `boost::gregorian::date`.
`boost::gregorian::date` provides several constructors to create dates. The most basic constructor takes a year, a month, and a day as parameters. If an invalid value is given, an exception will be thrown of type `boost::gregorian::bad_day_of_month`, `boost::gregorian::bad_year`, or `boost::gregorian::bad_month`. All of these classes are derived from `std::out_of_range`.

Example 36.1 Creating a date with `boost::gregorian::date`

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

int main()
{
  boost::gregorian::date d{2014, 1, 31};
  std::cout << d.year() << '\n';
  std::cout << d.month() << '\n';
  std::cout << d.day() << '\n';
  std::cout << d.day_of_week() << '\n';
  std::cout << d.end_of_month() << '\n';
}
```

As shown in Example 36.1, there are many member functions available. Some member functions, such as `year()`, `month()`, and `day()`, return the respective parts of a date, and others, such as `day_of_week()` and `end_of_month()`, calculate values.
The constructor of `boost::gregorian::date` expects numeric values for year, month, and day to set a date. However, the output of the sample program is `Jan` for the month and `Fri` for the day of the week. The return values of `month()` and `day_of_week()` are not regular numeric values, but values of type boost::gregorian::date::month_type and boost::gregorian::date::day_of_week_type. Boost.DateTime provides comprehensive support for formatted input and output, so it is possible to adjust the output from, for example, `Jan` to `1`.

The default constructor of `boost::gregorian::date` creates an invalid date. An invalid date can also be created explicitly by passing `boost::date_time::not_a_date_time` as the sole parameter to the constructor. Besides calling a constructor directly, an object of type `boost::gregorian::date` can be created via free-standing functions and member functions of other classes.
Example 36.2 uses the class `boost::gregorian::day_clock`, which returns the current date. The member function `universal_day()` returns a UTC date, which is independent of time zones and daylight savings. UTC is the international abbreviation for the universal time. `boost::gregorian::day_clock` also provides a member function called `local_day()`, which takes local settings into account. To retrieve the current date within the local time zone, use `local_day()`.
The namespace `boost::gregorian` contains free-standing functions to convert a date stored as a string into an object of type `boost::gregorian::date`. Example 36.2 converts a date in the ISO 8601 format using the function `boost::gregorian::date_from_iso_string()`. Other functions include: `boost::gregorian::from_simple_string()` and `boost::gregorian::from_us_string()`.

Example 36.2 Getting a date from a clock or a string

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d = day_clock::universal_day();
  std::cout << d.year() << '\n';
  std::cout << d.month() << '\n';
  std::cout << d.day() << '\n';

  d = date_from_iso_string("20140131");
  std::cout << d.year() << '\n';
  std::cout << d.month() << '\n';
  std::cout << d.day() << '\n';
}
```

While `boost::gregorian::date` marks a specific time, `boost::gregorian::date_duration` denotes a duration.

Example 36.3 Using `boost::gregorian::date_duration`

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d1{2014, 1, 31};
  date d2{2014, 2, 28};
  date_duration dd = d2 - d1;
  std::cout << dd.days() << '\n';
}
```

Because `boost::gregorian::date` overloads `operator-`, two points in time can be subtracted (see Example 36.3). The return value is of type `boost::gregorian::date_duration` and marks the duration between the two dates.
The most important member function offered by `boost::gregorian::date_duration` is `days()`, which returns the number of days in the duration specified.

Example 36.4 Specialized durations

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
```

```
{
  date_duration dd{4};
  std::cout << dd.days() << '\n';
  weeks ws{4};
  std::cout << ws.days() << '\n';
  months ms{4};
  std::cout << ms.number_of_months() << '\n';
  years ys{4};
  std::cout << ys.number_of_years() << '\n';
}
```

Objects of type `boost::gregorian::date_duration` can also be created by passing the number of days as a single parameter to the constructor. To create a duration that involves weeks, months, or years, use `boost::gregorian::weeks`, `boost::gregorian::months`, or `boost::gregorian::years` (see Example 36.4). Neither `boost::gregorian::months` nor `boost::gregorian::years` will allow you to determine the number of days, because months and years vary in length. Nonetheless, using these classes can still make sense, as shown in Example 36.5.

Example 36.5 Processing specialized durations

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d{2014, 1, 31};
  months ms{1};
  date d2 = d + ms;
  std::cout << d2 << '\n';
  date d3 = d2 - ms;
  std::cout << d3 << '\n';
}
```

Example 36.5 adds one month to the given date of January 31, 2014, which results in **`d2`** being February 28, 2014. In the next step, one month is subtracted and **`d3`** becomes January 31, 2014, again. As shown, points in time as well as durations can be used in calculations. However, some specifics need to be taken into account. For example, starting at the last day of a month, `boost::gregorian::months` always arrives at the last day of another month, which can lead to surprises.

Example 36.6 Surprises when processing specialized durations

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d{2014, 1, 30};
  months ms{1};
  date d2 = d + ms;
  std::cout << d2 << '\n';
  date d3 = d2 - ms;
  std::cout << d3 << '\n';
}
```

Example 36.6 is identical to the previous one, except **`d`** is initialized to be January 30, 2014. Even though this is not the last day in January, jumping forward by one month results in **`d2`** becoming February 28, 2014, because there is no February 30. However, jumping backwards by one month again results in **`d3`** becoming January 31, 2014. Since February 28, 2014, is the last day in February, jumping backwards returns to the last day in January. To change this behavior, undefine the macro `BOOST_DATE_TIME_OPTIONAL_GREGORIAN_TYPES`. After this macro is undefined, the classes `boost::gregorian::weeks`, `boost::gregorian::months`, and `boost::`

`gregorian::years` will no longer be available. The only class still available will be `boost::gregorian::date_duration`, which simply jumps forwards and backwards by a specified number of days and does not give special consideration to the first and last day of the month.

Example 36.7 Using `boost::gregorian::date_period`

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d1{2014, 1, 1};
  date d2{2014, 2, 28};
  date_period dp{d1, d2};
  date_duration dd = dp.length();
  std::cout << dd.days() << '\n';
}
```

While `boost::gregorian::date_duration` only works with durations, `boost::gregorian::date_period` supports ranges between two dates.

The constructor of `boost::gregorian::date_period` can accept two kinds of input. You can pass two parameters of type `boost::gregorian::date`, one for the beginning date and one for the end date. Or you can specify the beginning date and a duration of type `boost::gregorian::date_duration`. Please note that the day before the end date is actually the last day of the period. This is important in order to understand the output of Example 36.8.

Example 36.8 Testing whether a period contains dates

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  date d1{2014, 1, 1};
  date d2{2014, 2, 28};
  date_period dp{d1, d2};
  std::cout.setf(std::ios::boolalpha);
  std::cout << dp.contains(d1) << '\n';
  std::cout << dp.contains(d2) << '\n';
}
```

Example 36.8 checks whether a specific date is within a period by calling `contains()`. Notice that although **`d2`** defines the end of the period, it is not considered part of the period. Therefore, the member function `contains()` will return `true` when called with **`d1`** and `false` when called with **`d2`**.

`boost::gregorian::date_period` provides additional member functions for operations such as shifting a period or calculating the intersection of two overlapping periods.

Boost.DateTime also provides iterators and other useful free-standing functions as shown in Example 36.9.

Use the iterator `boost::gregorian::day_iterator` to jump forward or backward by a day from a specific date. Use `boost::gregorian::week_iterator`, `boost::gregorian::month_iterator`, and `boost::gregorian::year_iterator` to jump by weeks, months, or years, respectively.

Example 36.9 also uses the function `boost::date_time::next_weekday()`, which returns the date of the next weekday based on a given date. Example 36.9 displays `2014-May-16`, which is the first Friday following May 13, 2014.

Example 36.9 Iterating over dates

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost;

int main()
```

```
{
  gregorian::date d{2014, 5, 12};
  gregorian::day_iterator it{d};
  std::cout << *++it << '\n';
  std::cout << date_time::next_weekday(*it,
    gregorian::greg_weekday(date_time::Friday)) << '\n';
}
```

36.2 Location-independent Times

The class `boost::posix_time::ptime` defines a location-independent time. It uses the type `boost::gregorian::date`, but also stores a time. To use `boost::posix_time::ptime`, include the header file `boost/date_time/posix_time/posix_time.hpp`.

Example 36.10 Using `boost::posix_time::ptime`

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  date d = pt.date();
  std::cout << d << '\n';
  time_duration td = pt.time_of_day();
  std::cout << td << '\n';
}
```

To initialize an object of type `boost::posix_time::ptime`, pass a date of type `boost::gregorian::date` and a duration of type `boost::posix_time::time_duration` as the first and second parameters to the constructor. The constructor of `boost::posix_time::time_duration` takes three parameters, which determine the time. Example 36.10 specifies 12 PM on May 12, 2014, as the point in time. To query date and time, use the member functions `date()` and `time_of_day()`.
Just as the default constructor of `boost::gregorian::date` creates an invalid date, the default constructor of `boost::posix_time::ptime` creates an invalid time. An invalid time can also be created explicitly by passing `boost::date_time::not_a_date_time` to the constructor. Boost.DateTime provides free-standing functions and member functions to create times that are analogous to those used to create calendar dates of type `boost::gregorian::date`.

Example 36.11 Creating a timepoint with a clock or a string

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>

using namespace boost::posix_time;

int main()
{
  ptime pt = second_clock::universal_time();
  std::cout << pt.date() << '\n';
  std::cout << pt.time_of_day() << '\n';

  pt = from_iso_string("20140512T120000");
  std::cout << pt.date() << '\n';
  std::cout << pt.time_of_day() << '\n';
}
```

The class `boost::posix_time::second_clock` returns the current time. The member function `universal_time()` returns the UTC time (see Example 36.11). `local_time()` returns the local time. If you need a higher resolution, `boost::posix_time::microsec_clock` returns the current time including microseconds.
The free-standing function `boost::posix_time::from_iso_string()` converts a time stored in a string formatted using the ISO 8601 standard into an object of type `boost::posix_time::ptime`.

Example 36.12 Using `boost::posix_time::time_duration`

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;

int main()
{
  time_duration td{16, 30, 0};
  std::cout << td.hours() << '\n';
  std::cout << td.minutes() << '\n';
  std::cout << td.seconds() << '\n';
  std::cout << td.total_seconds() << '\n';
}
```

Boost.DateTime also provides the class `boost::posix_time::time_duration`, which specifies a duration. This class has been mentioned before because the constructor of `boost::posix_time::ptime` expects an object of type `boost::posix_time::time_duration` as its second parameter. You can also use `boost::posix_time::time_duration` independently.
`hours()`, `minutes()`, and `seconds()` return the respective parts of a time duration, while member functions such as `total_seconds()`, which returns the total number of seconds, provide additional information (see Example 36.12). There is no upper limit, such as 24 hours, to the values you can legally pass to `boost::posix_time::time_duration`.

Example 36.13 Processing timepoints

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt1{date{2014, 5, 12}, time_duration{12, 0, 0}};
  ptime pt2{date{2014, 5, 12}, time_duration{18, 30, 0}};
  time_duration td = pt2 - pt1;
  std::cout << td.hours() << '\n';
  std::cout << td.minutes() << '\n';
  std::cout << td.seconds() << '\n';
}
```

As with calendar dates, calculations can be performed with points in time and durations. If two times of type `boost::posix_time::ptime` are subtracted from each other, as in Example 36.13, the result is an object of type `boost::posix_time::time_duration` that specifies the duration between the two times.

Example 36.14 Processing time durations

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt1{date{2014, 5, 12}, time_duration{12, 0, 0}};
  time_duration td{6, 30, 0};
  ptime pt2 = pt1 + td;
  std::cout << pt2.time_of_day() << '\n';
}
```

As shown in Example 36.14, a duration can be added to a time, resulting in a new point in time. This example writes `18:30:00` to the standard output stream.

Boost.DateTime uses the same concepts for calendar dates and times. Just as there are classes for times and durations, there is also one for periods. For calendar dates, this is `boost::gregorian::date_period`; for times it is `boost::posix_time::time_period`. The constructors of both classes expect two parameters: `boost::gregorian::date_period` expects two calendar dates as parameters and `boost::posix_time::time_period` expects two points in time.

Example 36.15 Using `boost::posix_time::time_period`

```
#include <boost/date_time/posix_time/posix_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt1{date{2014, 5, 12}, time_duration{12, 0, 0}};
  ptime pt2{date{2014, 5, 12}, time_duration{18, 30, 0}};
  time_period tp{pt1, pt2};
  std::cout.setf(std::ios::boolalpha);
  std::cout << tp.contains(pt1) << '\n';
  std::cout << tp.contains(pt2) << '\n';
}
```

In general, `boost::posix_time::time_period` works just like `boost::gregorian::date_period`. It provides a member function, `contains()`, which returns `true` for every point in time within the period. Because the end time, which is passed to the constructor of `boost::posix_time::time_period`, is not part of the period, the second call to `contains()` in Example 36.15 returns `false`.

`boost::posix_time::time_period` provides additional member functions such as `intersection()` and `merge()`, which respectively, calculate the intersection of two overlapping periods and merge two intersecting periods.

Finally, the iterator `boost::posix_time::time_iterator` iterates over points in time.

Example 36.16 uses the iterator **`it`** to jump forward 6.5 hours from the time **`pt`**. Because the iterator is incremented twice, the output is `2014-May-12 18:30:00` and `2014-May-13 01:00:00`.

Example 36.16 Iterating over points in time

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  time_iterator it{pt, time_duration{6, 30, 0}};
  std::cout << *++it << '\n';
  std::cout << *++it << '\n';
}
```

36.3 Location-dependent Times

Unlike the location-independent times introduced in the previous section, location-dependent times account for time zones. Boost.DateTime provides the class `boost::local_time::local_date_time`, which is defined in `boost/date_time/local_time/local_time.hpp`. This class stores time-zone related data using `boost::local_time::posix_time_zone`.

Example 36.17 Using `boost::local_time::local_date_time`

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::local_time;
using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  time_zone_ptr tz{new posix_time_zone{"CET+1"}};
  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  local_date_time dt{pt, tz};
  std::cout << dt.utc_time() << '\n';
  std::cout << dt << '\n';
  std::cout << dt.local_time() << '\n';
  std::cout << dt.zone_name() << '\n';
}
```

The constructor of `boost::local_time::local_date_time` expects its first parameter to be an object of type `boost::posix_time::ptime` and its second parameter to be an object of type boost::local_time::time_zone_ptr. boost::local_time::time_zone_ptr is a type definition for boost::shared_ptr<boost::local_time::time_zone>. The type definition is based on `boost::local_time::time_zone`, not `boost::local_time::posix_time_zone`. That's fine because `boost::local_time::posix_time_zone` is derived from `boost::local_time::time_zone`. This makes it possible to extend Boost.DateTime with user-defined types for time zones.
No object of type `boost::local_time::posix_time_zone` is passed. Instead, a smart pointer referring to the object is passed. This allows multiple objects of type `boost::local_time::local_date_time` to share time-zone data. When the last object is destroyed, the object representing the time zone will automatically be released.
To create an object of type `boost::local_time::posix_time_zone`, a string describing the time zone is passed to the constructor as the only parameter. Example 36.17 specifies Central Europe as the time zone (CET is the abbreviation for Central European Time). Since CET is one hour ahead of UTC, the deviation is represented as +1. Boost.DateTime is not able to interpret abbreviations for time zones and thus does not know the meaning of CET. Therefore, the deviation must always be provided in hours; use the value +0 if there is no deviation.
The program writes the strings `2014-May-12 12:00:00`, `2014-May-12 13:00:00 CET`, `2014-May-12 13:00:00`, and `CET` to the standard output stream. Values used to initialize objects of type `boost::posix_time::ptime` and `boost::local_time::local_date_time` always relate to the UTC time zone by default. When an object of type `boost::local_time::local_date_time` is written to the standard output stream or a call to the member function `local_time()` is made, the deviation in hours is used to calculate the local time.

Example 36.18 Location-dependent points in time and different time zones

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::local_time;
using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  time_zone_ptr tz{new posix_time_zone{"CET+1"}};

  ptime pt{date{2014, 5, 12}, time_duration{12, 0, 0}};
  local_date_time dt{pt, tz};
  std::cout << dt.local_time() << '\n';

  time_zone_ptr tz2{new posix_time_zone{"EET+2"}};
  std::cout << dt.local_time_in(tz2).local_time() << '\n';
}
```

With `local_time()`, the deviation for the time zone is respected. In order to calculate the CET time, one hour needs to be added to the UTC time of 12 PM stored in **`dt`**, since CET is one hour ahead of UTC. That's why `local_time()` writes `2014-May-12 13:00:00` to standard output in Example 36.18.
In contrast, the member function `local_time_in()` interprets the time stored in **`dt`** as being in the time zone that is passed as a parameter. This means that 12 PM UTC equals 2 PM EET which stands for Eastern European Time and is two hours ahead of UTC.
Finally, Boost.DateTime provides the class `boost::local_time::local_time_period` for location-dependent periods.

Example 36.19 Using `boost::local_time::local_time_period`

```
#include <boost/date_time/local_time/local_time.hpp>
#include <iostream>

using namespace boost::local_time;
using namespace boost::posix_time;
using namespace boost::gregorian;

int main()
{
  time_zone_ptr tz{new posix_time_zone{"CET+0"}};

  ptime pt1{date{2014, 12, 5}, time_duration{12, 0, 0}};
  local_date_time dt1{pt1, tz};

  ptime pt2{date{2014, 12, 5}, time_duration{18, 0, 0}};
  local_date_time dt2{pt2, tz};

  local_time_period tp{dt1, dt2};

  std::cout.setf(std::ios::boolalpha);
  std::cout << tp.contains(dt1) << '\n';
  std::cout << tp.contains(dt2) << '\n';
}
```

The constructor of `boost::local_time::local_time_period` in Example 36.19 expects two parameters of type `boost::local_time::local_date_time`. As with other types provided for periods, the second parameter, which represents the end time, is not part of the period. With the help of member functions such as `contains()`, `intersection()`, `merge()`, and others, you can process periods based on `boost::local_time::local_time_period`.

36.4 Formatted Input and Output

The sample programs described so far in this chapter write results in the format `2014-May-12`. Boost.DateTime lets you display results in different formats. Calendar dates and times can be formatted using `boost::date_time::date_facet` and `boost::date_time::time_facet`.
Boost.DateTime uses the concept of locales from the standard. To format a calendar date, an object of type `boost::date_time::date_facet` must be created and installed within a locale. A string describing the new format is passed to the constructor of `boost::date_time::date_facet`. Example 36.20 passes "%A, %d %B %Y", which specifies that the day of the week is followed by the date with the month written in full: `Monday, 12 May 2014`.

Example 36.20 A user-defined format for a date

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <iostream>
#include <locale>

using namespace boost::gregorian;

int main()
{
  date d{2014, 5, 12};
  date_facet *df = new date_facet{"%A, %d %B %Y"};
  std::cout.imbue(std::locale{std::cout.getloc(), df});
  std::cout << d << '\n';
}
```

Boost.DateTime provides numerous format flags, each of which consists of the percent sign followed by a character. The documentation for Boost.DateTime contains a complete overview of all supported flags.
If a program is used by people located in Germany or German-speaking countries, it is preferable to display both the weekday and the month in German rather than in English.
The names for weekdays and months can be changed by passing vectors containing the desired names to the member functions `long_month_names()` and `long_weekday_names()` of the class `boost::date_time::date_facet`. Example 36.21 now writes `Montag, 12.Mai 2014` to the standard output stream.

Note

To run the example on a POSIX operating system, replace “German” with “de_DE” and make sure the locale for German is installed.

Example 36.21 Changing names of weekdays and months

```
#include <boost/date_time/gregorian/gregorian.hpp>
#include <string>
#include <vector>
#include <locale>
#include <iostream>

using namespace boost::gregorian;

int main()
{
  std::locale::global(std::locale{"German"});
  std::string months[12]{"Januar", "Februar", "M\xe4rz", "April",
    "Mai", "Juni", "Juli", "August", "September", "Oktober",
    "November", "Dezember"};
  std::string weekdays[7]{"Sonntag", "Montag", "Dienstag",
    "Mittwoch", "Donnerstag", "Freitag", "Samstag"};
  date d{2014, 5, 12};
  date_facet *df = new date_facet{"%A, %d. %B %Y"};
  df->long_month_names(std::vector<std::string>{months, months + 12});
  df->long_weekday_names(std::vector<std::string>{weekdays,
    weekdays + 7});
  std::cout.imbue(std::locale{std::cout.getloc(), df});
  std::cout << d << '\n';
}
```

Boost.DateTime is flexible with regard to formatted input and output. Besides the output classes `boost::date_time::date_facet` and `boost::date_time::time_facet`, the classes `boost::date_time::date_input_facet` and `boost::date_time::time_input_facet` are available for formatted input. All four classes provide member functions to configure the input and output of different objects provided by Boost.DateTime.
For example, it is possible to specify how periods of type `boost::gregorian::date_period` are input and output. To see all of the possibilities for formatted input and output, review the documentation for Boost.DateTime.

Chapter 37

Boost.Chrono

The library Boost.Chrono provides a variety of clocks. For example, you can get the current time or you can measure the time passed in a process.
Parts of Boost.Chrono were added to C++11. If your development environment supports C++11, you have access to several clocks defined in the header file `chrono`. However, C++11 doesn't support some features, for example clocks to measure CPU time. Furthermore, only Boost.Chrono supports user-defined output formats for time.
You have access to all Boost.Chrono clocks through the header file `boost/chrono.hpp`. The only extension is user-defined formatting, which requires the header file `boost/chrono_io.hpp`.
Example 37.1 introduces all of the clocks provided by Boost.Chrono. All clocks have in common the member function `now()`, which returns a timepoint. All timepoints are relative to a universally valid timepoint. This reference timepoint is called *epoch*. An often used epoch is 1 January 1970. Example 37.1 writes the epoch for every timepoint displayed.

Example 37.1 All clocks from Boost.Chrono

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << system_clock::now() << '\n';
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
  std::cout << steady_clock::now() << '\n';
#endif
  std::cout << high_resolution_clock::now() << '\n';

#ifdef BOOST_CHRONO_HAS_PROCESS_CLOCKS
  std::cout << process_real_cpu_clock::now() << '\n';
  std::cout << process_user_cpu_clock::now() << '\n';
  std::cout << process_system_cpu_clock::now() << '\n';
  std::cout << process_cpu_clock::now() << '\n';
#endif

#ifdef BOOST_CHRONO_HAS_THREAD_CLOCK
  std::cout << thread_clock::now() << '\n';
#endif
}
```

Boost.Chrono includes the following clocks:

- `boost::chrono::system_clock` returns the system time. This is the time usually displayed on the desktop of your computer. If you change the time on your computer, `boost::chrono::system_clock` returns the new time. Example 37.1 writes a string to standard output that looks like the following: `13919594042183544 [1/10000000]seconds since Jan 1, 1970`.

  The epoch isn't standardized for `boost::chrono::system_clock`. The epoch 1 January 1970, which is used in these examples, is implementation dependent. However, if you specifically want to get the time since 1 January 1970, call `to_time_t()`. `to_time_t()` is a static member function that returns the current system time as the number of seconds since 1 January 1970 as a std::time_t.

- `boost::chrono::steady_clock` is a clock that will always return a later time when it is accessed later. Even if the time is set back on a computer, `boost::chrono::steady_clock` will return a later time. This time is known as *monotonic time*. Example 37.1 displays the number of nanoseconds since the system was booted. The message looks like the following: `10594369282958 nanoseconds since boot`. `boost::chrono::steady_clock` measures the time elapsed since the last boot. However, starting the measurement since the last boot is an implementation detail. The reference point could change with a different implementation.

  `boost::chrono::steady_clock` isn't supported on all platforms. The clock is only available if the macro `BOOST_CHRONO_HAS_CLOCK_STEADY` is defined.

- `boost::chrono::high_resolution_clock` is a type definition for `boost::chrono::system_clock` or `boost::chrono::steady_clock`, depending on which clock measures time more precisely. Thus, the output is identical to the output of the clock `boost::chrono::high_resolution_clock` is based on.

- `boost::chrono::process_real_cpu_clock` returns the CPU time a process has been running. The clock measures the time since program start. Example 37.1 writes a string to standard output that looks like the following: `1000000 nanoseconds since process start-up`.

  You could also get this time using `std::clock()` from `ctime`. In fact, the current implementation of `boost::chrono::process_real_cpu_clock` is based on `std::clock()`.

  The `boost::chrono::process_real_cpu_clock` clock and other clocks measuring CPU time can only be used if the macro `BOOST_CHRONO_HAS_PROCESS_CLOCKS` is defined.

- `boost::chrono::process_user_cpu_clock` returns the CPU time a process spent in *user space*. User space refers to code that runs separately from operating system functions. The time it takes to execute code in operating system functions called by a program is not counted as user space time.

  `boost::chrono::process_user_cpu_clock` returns only the time spent running in user space. If a program is halted for a while, for example through the Windows `Sleep()` function, the time spent in `Sleep()` isn't measured by `boost::chrono::process_user_cpu_clock`.

  Example 37.1 writes a string to standard output that looks like the following: `15600100 nanoseconds since process start-up`.

- `boost::chrono::process_system_cpu_clock` is similar to `boost::chrono::process_user_cpu_clock`. However, this clock measures the time spent in *kernel space*. `boost::chrono::process_system_cpu_clock` returns the CPU time a process spends executing operating system functions.

  Example 37.1 writes a string to the standard output that looks like the following: `0 nanoseconds since process start-up`. Because this example doesn't call operating system functions directly and because Boost.Chrono uses only a few operating system functions, `boost::chrono::process_system_cpu_clock` may return 0.

- `boost::chrono::process_cpu_clock` returns a tuple with the CPU times which are returned by `boost::chrono::process_real_cpu_clock`, `boost::chrono::process_user_cpu_clock` and `boost::chrono::process_system_cpu_clock`. Example 37.1 writes a string to standard output that looks like the following: `{1000000;15600100;0} nanoseconds since process start-up`.

- `boost::chrono::thread_clock` returns the time used by a thread. The time measured by `boost::chrono::thread_clock` is comparable to CPU time, except it is per thread, rather than per process. `boost::chrono::thread_clock` returns the CPU time the thread has been running. It does not distinguish between time spent in user and kernel space.

  `boost::chrono::thread_clock` isn't supported on all platforms. You can only use `boost::chrono::thread_clock` if the macro `BOOST_CHRONO_HAS_THREAD_CLOCK` is defined.

  Boost.Chrono provides the macro, `BOOST_CHRONO_THREAD_CLOCK_IS_STEADY`, to detect whether `boost::chrono::thread_clock` measures monotonic time like `boost::chrono::steady_clock`.

  Example 37.1 writes a string to standard output that looks like the following: `15600100 nanoseconds since thread start-up`.

All of the clocks in Boost.Chrono depend on operating system functions; thus, the operating system determines how precise and reliable the returned times are.

Example 37.2 Adding and subtracting durations using Boost.Chrono

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << p - nanoseconds{1} << '\n';
  std::cout << p + milliseconds{1} << '\n';
  std::cout << p + seconds{1} << '\n';
  std::cout << p + minutes{1} << '\n';
  std::cout << p + hours{1} << '\n';
}
```

`now()` returns an object of type `boost::chrono::time_point` for all clocks. This type is tightly coupled with a clock because the timepoint is measured relative to a reference timepoint that is defined by a clock. `boost::chrono::time_point` is a template that expects the type of a clock as a parameter. Each clock type provides a type definition for its specialized `boost::chrono::time_point`. For example, the type definition for `process_real_cpu_clock` is process_real_cpu_clock::time_point.

Boost.Chrono also provides the class `boost::chrono::duration`, which describes durations. Because `boost::chrono::duration` is also a template, Boost.Chrono provides the six classes `boost::chrono::nanoseconds`, `boost::chrono::milliseconds`, `boost::chrono::microseconds`, `boost::chrono::seconds`, `boost::chrono::minutes`, and `boost::chrono::hours`, which are easier to use.

Boost.Chrono overloads several operators to process timepoints and durations. Example 37.2 subtracts durations from or adds durations to **p** to get new timepoints, which are written to standard output.

Example 37.2 displays all timepoints in nanoseconds. Boost.Chrono automatically uses the smallest unit when timepoints and durations are processed to make sure that results are as precise as possible. If you want to use a timepoint with another unit, you have to cast it.

Example 37.3 Casting timepoints with `boost::chrono::time_point_cast()`

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << time_point_cast<minutes>(p) << '\n';
}
```

The `boost::chrono::time_point_cast()` function is used like a cast operator. Example 37.3 uses `boost::chrono::time_point_cast()` to convert a timepoint based on nanoseconds to a timepoint in minutes. You must use `boost::chrono::time_point_cast()` in this case because the timepoint cannot be expressed in a less precise unit (minutes) without potentially losing precision. You don't require `boost::chrono::time_point_cast()` to convert from less precise to more precise units.

Boost.Chrono also provides cast operators for durations.

Example 37.4 Casting durations with `boost::chrono::duration_cast()`

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  minutes m{1};
  seconds s{35};

  std::cout << m + s << '\n';
  std::cout << duration_cast<minutes>(m + s) << '\n';
}
```

Example 37.4 uses the function `boost::chrono::duration_cast()` to cast a duration from seconds to minutes. This example writes `1 minute` to standard output.

Example 37.5 Rounding durations

```
#include <boost/chrono.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << floor<minutes>(minutes{1} + seconds{45}) << '\n';
  std::cout << round<minutes>(minutes{1} + seconds{15}) << '\n';
  std::cout << ceil<minutes>(minutes{1} + seconds{15}) << '\n';
}
```

Boost.Chrono also provides functions to round durations when casting. `boost::chrono::round()` rounds up or down, `boost::chrono::floor()` rounds down, and `boost::chrono::ceil()` rounds up. `boost::chrono::floor()` uses `boost::chrono::duration_cast()` – there is no difference between these two functions.

Example 37.5 writes `1 minute`, `1 minute`, and `2 minutes` to standard output.

Example 37.6 Stream manipulators for user-defined output

```
#define BOOST_CHRONO_VERSION 2
#include <boost/chrono.hpp>
#include <boost/chrono/chrono_io.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << symbol_format << minutes{10} << '\n';

  std::cout << time_fmt(boost::chrono::timezone::local, "%H:%M:%S") <<
    system_clock::now() << '\n';
}
```

Boost.Chrono provides various stream manipulators to format the output of timepoints and durations. For example, with the manipulator `boost::chrono::symbol_format()`, the time unit is written as a symbol instead of a name. Thus, Example 37.6 displays `10 min`.

The manipulator `boost::chrono::time_fmt()` can be used to set a timezone and a format string. The timezone must be set to `boost::chrono::timezone::local` or `boost::chrono::timezone::utc`. The format string can use flags to refer to various components of a timepoint. For example, Example 37.6 writes a string to the standard output that looks like the following: `15:46:44`.

Beside stream manipulators, Boost.Chrono provides facets for many different customizations. For example, there is a facet that makes it possible to output timepoints in another language.

Note

There are two versions of the input/output functions since Boost 1.52.0. Since Boost 1.55.0, the newer version is used by default. If you use a version older than 1.55.0, you must define the macro `BOOST_CHRONO_VERSION` and set it to 2 for Example 37.6 to work.

Chapter 38

Boost.Timer

Boost.Timer provides clocks to measure code performance. At first, it may seem like this library competes with Boost.Chrono. However, while Boost.Chrono provides clocks to measure arbitrary periods, Boost.Timer measures the time it takes to execute code. Although Boost.Timer uses Boost.Chrono, when you want to measure code performance, you should use Boost.Timer rather than Boost.Chrono.

Since version 1.48.0 of the Boost libraries, there have been two versions of Boost.Timer. The first version of Boost.Timer has only one header file: `boost/timer/timer.hpp`. Do not use this header file. It belongs to the first version of Boost.Timer, which shouldn't be used anymore.

The clocks provided by Boost.Timer are implemented in the classes `boost::timer::cpu_timer` and `boost::timer::auto_cpu_timer`. `boost::timer::auto_cpu_timer` is derived from `boost::timer::cpu_timer` and automatically stops the time in the destructor. It then writes the time to an output stream.

Example 38.1 starts by introducing the class `boost::timer::cpu_timer`. This example and the following examples do some calculations to make sure enough time elapses to be measurable. Otherwise the timers would always measure 0, and it would be difficult to introduce the clocks from this library.

Measurement starts when `boost::timer::cpu_timer` is instantiated. You can call the member function `format()` at any point to get the elapsed time. Example 38.1 displays output in the following format: `0.099170s wall, 0.093601s user + 0.000000s system =0.093601s CPU (94.4%)`.

Boost.Timer measures wall and CPU time. The wall time is the time which passes according to a wall clock. You could measure this time yourself with a stop watch. The CPU time says how much time the program spent executing code. On today's multitasking systems a processor isn't available for a program all the time. A program may also need to halt and wait for user input. In these cases the wall time moves on but not the CPU time.

Example 38.1 Measuring time with `boost::timer::cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

CPU time is divided between time spent in *user space* and time spent in *kernel space*. Kernel space refers to code that is part of the operating system. User space is code that doesn't belong to the operating system. User space includes your program code and code from third-party libraries. For example, the Boost libraries are included in user space. The amount of time spent in kernel space depends on the operating system functions called and how much time those functions need.

Example 38.2 Stopping and resuming timers

```
#include <boost/timer/timer.hpp>
#include <iostream>
```

```
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.stop();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.resume();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

`boost::timer::cpu_timer` provides the member functions `stop()` and `resume()`, which stop and resume timers. In Example 38.2, the timer is stopped before the second `for` loop runs and resumed afterwards. Thus, the second `for` loop isn't measured. This is similar to a stop watch that is stopped and then resumed after a while. The time returned by the second call to `format()` in Example 38.2 is the same as if the second `for` loop didn't exist.

`boost::timer::cpu_timer` also provides a member function `start()`. If you call `start()`, instead of `resume()`, the timer restarts from zero. The constructor of `boost::timer::cpu_timer` calls `start()`, which is why the timer starts immediately when `boost::timer::cpu_timer` is instantiated.

Example 38.3 Getting wall and CPU time as a tuple

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);

  cpu_times times = timer.elapsed();
  std::cout << times.wall << '\n';
  std::cout << times.user << '\n';
  std::cout << times.system << '\n';
}
```

While `format()` returns the measured wall and CPU time as a string, it is also possible to receive the times in a tuple (see Example 38.3). `boost::timer::cpu_timer` provides the member function `elapsed()` for that. `elapsed()` returns a tuple of type `boost::timer::times`. This tuple has three member variables: **`wall`**, **`user`**, and **`system`**. These member variables contain the wall and CPU times in nanoseconds. Their type is boost::int_least64_t.

`boost::timer::times` provides the member function `clear()` to set **`wall`**, **`user`**, and **`system`** to 0.

Example 38.4 Measuring times automatically with `boost::timer::auto_cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <cmath>
```

```
using namespace boost::timer;

int main()
{
  auto_cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
}
```

You can measure the wall and CPU time of a code block with `boost::timer::auto_cpu_timer`. Because the destructor of this class stops measuring time and writes the time to the standard output stream, Example 38.4 does the same thing as Example 38.1.

`boost::timer::auto_cpu_timer` provides several constructors. For example, you can pass an output stream that will be used to display the time. By default, the output stream is **`std::cout`**.

You can specify the format of reported times for `boost::timer::auto_cpu_timer` and `boost::timer::cpu_timer`. Boost.Timer provides format flags similar to the format flags supported by Boost.Format or `std::printf()`. The documentation contains an overview of the format flags.

Part IX

Functional Programming

In the functional programming model, functions are objects that, like other objects, can be passed as parameters to functions or stored in containers. There are numerous Boost libraries that support the functional programming model.

- Boost.Phoenix is the most extensive and, as of today, most important of these libraries. It replaces the library Boost.Lambda, which is introduced briefly, but only for completeness.
- Boost.Function provides a class that makes it easy to define a function pointer without using the syntax that originated with the C programming language.
- Boost.Bind is an adapter that lets you pass functions as parameters to other functions even if the actual signature is different from the expected signature.
- Boost.Ref can be used to pass a reference to an object, even if a function passes the parameter by copy.
- Boost.Lambda could be called a predecessor of Boost.Phoenix. It is a rather old library and allowed using lambda functions many years before they were added with C++11 to the programming language.

Chapter 39

Boost.Phoenix

Boost.Phoenix is the most important Boost library for functional programming. While libraries like Boost.Bind or Boost.Lambda provide some support for functional programming, Boost.Phoenix includes the features of these libraries and goes beyond them.
In functional programming, functions are objects and can be processed like objects. With Boost.Phoenix, it is possible for a function to return another function as a result. It is also possible to pass a function as a parameter to another function. Because functions are objects, it's possible to distinguish between instantiation and execution. Accessing a function isn't equal to executing it.
Boost.Phoenix supports functional programming with function objects: Functions are objects based on classes which overload the operator `operator()`. That way function objects behave like other objects in C++. For example, they can be copied and stored in a container. However, they also behave like functions because they can be called.
Functional programming isn't new in C++. You can pass a function as a parameter to another function without using Boost.Phoenix.
Example 39.1 uses the algorithm `std::count_if()` to count odd numbers in vector **v**. `std::count_if()` is called three times, once with a predicate as a free-standing function, once with a lambda function, and once with a Phoenix function.
The Phoenix function differs from free-standing and lambda functions because it has no frame. While the other two functions have a function header with a signature, the Phoenix function seems to consist of a function body only.

Example 39.1 Predicates as global function, lambda function, and Phoenix function

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  std::cout << std::count_if(v.begin(), v.end(), is_odd) << '\n';

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

The crucial component of the Phoenix function is **`boost::phoenix::placeholders::arg1`**. **`arg1`** is a global instance of a function object. You can use it like **`std::cout`**: These objects exist once the respective header file is included.

`arg1` is used to define an unary function. The expression `arg1 % 2 ==1` creates a new function that expects one parameter. The function isn't executed immediately but stored in **`phoenix`**. **`phoenix`** is passed to `std::count_if()` which calls the predicate for every number in **`v`**.
`arg1` is a placeholder for the value passed when the Phoenix function is called. Since only **`arg1`** is used here, a unary function is created. Boost.Phoenix provides additional placeholders such as **`boost::phoenix::placeholders::arg2`** and **`boost::phoenix::placeholders::arg3`**. A Phoenix function always expects as many parameters as the placeholder with the greatest number.
Example 39.1 writes `3` three times to standard output.
Example 39.2 highlights a crucial difference between Phoenix and lambda functions. In addition to requiring no function header with a parameter list, Phoenix function parameters have no types. The lambda function **`lambda`** expects a parameter of type int. The Phoenix function **`phoenix`** will accept any type that the modulo operator can handle.
Think of Phoenix functions as function templates. Like function templates, Phoenix functions can accept any type. This makes it possible in Example 39.2 to use **`phoenix`** as a predicate for the containers **`v`** and **`v2`** even though they store numbers of different types. If you try to use the predicate **`lambda`** with **`v2`**, you get a compiler error.

Example 39.2 Phoenix function versus lambda function

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  std::vector<long> v2;
  v2.insert(v2.begin(), v.begin(), v.end());

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
  std::cout << std::count_if(v2.begin(), v2.end(), phoenix) << '\n';
}
```

Example 39.3 uses a Phoenix function as a predicate with `std::count_if()` to count odd numbers greater than 2. The Phoenix function accesses **`arg1`** twice: Once to test if the placeholder is greater than 2 and once to test whether it's an odd number. The conditions are linked with &&.

Example 39.3 Phoenix functions as deferred C++ code

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > 2 && arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

You can think of Phoenix functions as C++ code that isn't executed immediately. The Phoenix function in Example 39.3 looks like a condition that uses multiple logical and arithmetic operators. However, the condition isn't executed immediately. It is only executed when it is accessed from within `std::count_if()`. The access in `std::count_if()` is a normal function call.
Example 39.3 writes `2` to standard output.

Example 39.4 Explicit Phoenix types

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > val(2) && arg1 % val(2) == val(1);
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.4 uses explicit types for all operands in the Phoenix function. Strictly speaking, you don't see types, just the helper function `boost::phoenix::val()`. This function returns a function object initialized with the values passed to `boost::phoenix::val()`. The actual type of the function object doesn't matter. What is important is that Boost.Phoenix overloads operators like `>`, `&&`, `%` and `==` for different types. Thus, conditions aren't checked immediately. Instead, function objects are combined to create more powerful function objects. Depending on the operands they may be automatically used as function objects. Otherwise you can call helper functions like `val()`.

Example 39.5 `boost::phoenix::placeholders::arg1` and `boost::phoenix::val()`

```
#include <boost/phoenix/phoenix.hpp>
#include <iostream>

int main()
{
  using namespace boost::phoenix::placeholders;
  std::cout << arg1(1, 2, 3, 4, 5) << '\n';

  auto v = boost::phoenix::val(2);
  std::cout << v() << '\n';
}
```

Example 39.5 illustrates how **`arg1`** and `val()` work. **`arg1`** is an instance of a function object. It can be used directly and called like a function. You can pass as many parameters as you like – **`arg1`** returns the first one. `val()` is a function to create an instance of a function object. The function object is initialized with the value passed as a parameter. If the instance is accessed like a function, the value is returned.
Example 39.5 writes `1` and `2` to standard output.

Example 39.6 Creating your own Phoenix functions

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

struct is_odd_impl
{
    typedef bool result_type;

    template <typename T>
    bool operator()(T t) const { return t % 2 == 1; }
};

boost::phoenix::function<is_odd_impl> is_odd;

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};
  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

Example 39.6 explains how you can create your own Phoenix function. You pass a function object to the template `boost::phoenix::function`. The example passes the class `is_odd_impl`. This class overloads the operator `operator()`: when an odd number is passed in, the operator returns `true`. Otherwise, the operator returns `false`.
Please note that you must define the type result_type. Boost.Phoenix uses it to detect the type of the return value of the operator `operator()`.
`is_odd()` is a function you can use like `val()`. Both functions return a function object. When called, parameters are forwarded to the operator `operator()`. For Example 39.6, this means that `std::count_if()` still counts odd numbers.
If you want to transform a free-standing function into a Phoenix function, you can proceed as in Example 39.7. You don't necessarily have to define a function object as in the previous example.
You use the macro `BOOST_PHOENIX_ADAPT_FUNCTION` to turn a free-standing function into a Phoenix function. Pass the type of the return value, the name of the Phoenix function to define, the name of the free-standing function, and the number of parameters to the macro.

Example 39.7 Transforming free-standing functions into Phoenix functions

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool is_odd_function(int i) { return i % 2 == 1; }

BOOST_PHOENIX_ADAPT_FUNCTION(bool, is_odd, is_odd_function, 1)

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

To use a free-standing function as a Phoenix function, you can also use `boost::phoenix::bind()` as in Example 39.8. `boost::phoenix::bind()` works like `std::bind()`. The name of the free-standing function is passed as the first parameter. All further parameters are forwarded to the free-standing function.

Example 39.8 Phoenix functions with `boost::phoenix::bind()`

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), bind(is_odd, arg1)) << '\n';
}
```

Tip

Avoid `boost::phoenix::bind()`. Create your own Phoenix functions. This leads to more readable code. Especially with complex expressions it's not helpful having to deal with the additional details of `boost::phoenix::bind()`.

Example 39.9 Arbitrarily complex Phoenix functions

```
#include <boost/phoenix/phoenix.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  int count = 0;
  std::for_each(v.begin(), v.end(), if_(arg1 > 2 && arg1 % 2 == 1)
    [
      ++ref(count)
    ]);
  std::cout << count << '\n';
}
```

Boost.Phoenix provides some function objects that simulate C++ keywords. For example, you can use the function `boost::phoenix::if_()` (see Example 39.9) to create a function object that acts like `if` and tests a condition. If the condition is true, the code passed to the function object with `operator[]` will be executed. Of course, that code also has to be based on function objects. That way, you can create complex Phoenix functions. Example 39.9 increments **`count`** for every odd number greater than 2. To use the increment operator on **`count`**, **`count`** is wrapped in a function object using `boost::phoenix::ref()`. In contrast to `boost::phoenix::val()`, no value is copied into the function object. The function object returned by `boost::phoenix::ref()` stores a reference – here a reference to **`count`**.

Tip

Don't use Boost.Phoenix to create complex functions. It is better to use lambda functions from C++11. While Boost.Phoenix comes close to C++ syntax, using keywords like `if_` or code blocks between square brackets doesn't necessarily improve readability.

Chapter 40

Boost.Function

Boost.Function provides a class called boost::function to encapsulate function pointers. It is defined in boost/function.hpp.
If you work in a development environment supporting C++11, you have access to the class std::function from the header file functional. In this case you can ignore Boost.Function because boost::function and std::function are equivalent.

Example 40.1 Using boost::function

```
#include <boost/function.hpp>
#include <iostream>
#include <cstdlib>
#include <cstring>

int main()
{
  boost::function<int(const char*)> f = std::atoi;
  std::cout << f("42") << '\n';
  f = std::strlen;
  std::cout << f("42") << '\n';
}
```

boost::function makes it possible to define a pointer to a function with a specific signature. Example 40.1 defines a pointer **f** that can point to functions that expect a parameter of type const char* and return a value of type int. Once defined, functions with matching signatures can be assigned to the pointer. Example 40.1 first assigns the function std::atoi() to **f** before std::strlen() is assigned to **f**.
Please note that types do not need to match exactly. Even though std::strlen() uses std::size_t as its return type, it can still be assigned to **f**.
Because **f** is a function pointer, the assigned function can be called using operator(). Depending on what function is currently assigned, either std::atoi() or std::strlen() is called.
If **f** is called without having a function assigned, an exception of type boost::bad_function_call is thrown (see Example 40.2).

Example 40.2 boost::bad_function_call thrown if boost::function is empty

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

int main()
{
  try
  {
    boost::function<int(const char*)> f;
    f("");
  }
  catch (boost::bad_function_call &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Note that assigning `nullptr` to a function pointer of type `boost::function` releases any currently assigned function. Calling it after it has been released will result in a `boost::bad_function_call` exception being thrown. To check whether or not a function pointer is currently assigned to a function, you can use the member functions `empty()` or `operator bool`.
It is also possible to assign class member functions to objects of type `boost::function` (see Example 40.3). When calling such a function, the first parameter passed indicates the particular object for which the function is called. Therefore, the first parameter after the open parenthesis inside the template definition must be a pointer to that particular class. The remaining parameters denote the signature of the corresponding member function.

Example 40.3 Binding a class member function to `boost::function`

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

struct world
{
  void hello(std::ostream &os)
  {
    os << "Hello, world!\n";
  }
};

int main()
{
  boost::function<void(world*, std::ostream&)> f = &world::hello;
  world w;
  f(&w, std::ref(std::cout));
}
```

Chapter 41

Boost.Bind

Boost.Bind is a library that simplifies and generalizes capabilities that originally required `std::bind1st()` and `std::bind2nd()`. These two functions were added to the standard library with C++98 and made it possible to connect functions even if their signatures aren't compatible.
Boost.Bind was added to the standard library with C++11. If your development environment supports C++11, you will find the function `std::bind()` in the header file `functional`. Depending on the use case, it may be better to use lambda functions or Boost.Phoenix than `std::bind()` or Boost.Bind.

Example 41.1 `std::for_each()` with a compatible function

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

void print(int i)
{
  std::cout << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), print);
}
```

The third parameter of `std::for_each()` is a function or function object that expects a sole parameter. In Example 41.1, `std::for_each()` passes the numbers in the container **v** as sole parameters, one after another, to `print()`.
If you need to pass in a function whose signature doesn't meet the requirements of an algorithm, it gets more difficult. For example, if you want `print()` to accept an output stream as an additional parameter, you can no longer use it as is with `std::for_each()`.
Like Example 41.1, Example 41.2 writes all numbers in **v** to standard output. However, this time, the output stream is passed to `print()` as a parameter. To do this, the function `print()` is defined as a function object derived from `std::binary_function`.

Example 41.2 `std::for_each()` with `std::bind1st()`

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

class print : public std::binary_function<std::ostream*, int, void>
{
public:
  void operator()(std::ostream *os, int i) const
  {
    *os << i << '\n';
  }
};
```

```
int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), std::bind1st(print{}, &std::cout));
}
```

With Boost.Bind, you don't need to transform `print()` from a function to a function object. Instead, you use the function template `boost::bind()`, which is defined in `boost/bind.hpp`.
Example 41.3 uses `print()` as a function, not as a function object. Because `print()` expects two parameters, the function can't be passed directly to `std::for_each()`. Instead, `boost::bind()` is passed to `std::for_each()` and `print()` is passed as the first parameter to `boost::bind()`.
Since `print()` expects two parameters, those two parameters must also be passed to `boost::bind()`. They are a pointer to **`std::cout`** and `_1`.
`_1` is a placeholder. Boost.Bind defines placeholders from `_1` to `_9`. These placeholders tell `boost::bind()` to return a function object that expects as many parameters as the placeholder with the greatest number. If, as in Example 41.3, only the placeholder `_1` is used, `boost::bind()` returns an unary function object – a function object that expects a sole parameter. This is required in this case since `std::for_each()` passes only one parameter.

Example 41.3 `std::for_each()` with `boost::bind()`

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

void print(std::ostream *os, int i)
{
  *os << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), boost::bind(print, &std::cout, _1));
}
```

`std::for_each()` calls a unary function object. The value passed to the function object – a number from the container **`v`** – takes the position of the placeholder `_1`. `boost::bind()` takes the number and the pointer to **`std::cout`** and forwards them to `print()`.
Please note that `boost::bind()`, like `std::bind1st()` and `std::bind2nd()`, takes parameters by value. To prevent the calling program from trying to copy **`std::cout`**, `print()` expects a pointer to a stream. Boost.Ref provides a function which allows you to pass a parameter by reference.
Example 41.4 illustrates how to define a binary function object with `boost::bind()`. It uses the algorithm `std::sort()`, which expects a binary function as its third parameter.
In Example 41.4, a binary function object is created because the placeholder `_2` is used. The algorithm `std::sort()` calls this binary function object with two values from the container **`v`** and evaluates the return value to sort the container. The function `compare()` is defined to sort **`v`** in descending order.

Example 41.4 `std::sort()` with `boost::bind()`

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _1, _2));
  for (int i : v)
    std::cout << i << '\n';
}
```

Since `compare()` is a binary function, it can be passed to `std::sort()` directly. However, it can still make sense to use `boost::bind()` because it lets you change the order of the parameters. For example, you can use `boost::bind()` if you want to sort the container in ascending order but don't want to change `compare()`. Example 41.5 sorts **`v`** in ascending order simply by swapping the placeholders: **`_2`** is passed first and **`_1`** second.

Example 41.5 `std::sort()` with `boost::bind()` and changed order of placeholders

```
#include <boost/bind.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _2, _1));
  for (int i : v)
    std::cout << i << '\n';
}
```

Chapter 42

Boost.Ref

The library Boost.Ref provides two functions, `boost::ref()` and `boost::cref()`, in the header file `boost/ref.hpp`. They are useful if you use, for example, `std::bind()` for a function which expects parameters by reference. Because `std::bind()` takes parameters by value, you have to deal with references explicitly.
Boost.Ref was added to the standard library in C++11, where you will find the functions `std::ref()` and `std::cref()` in the header file `functional`.

Example 42.1 Using `boost::ref()`

```
#include <boost/ref.hpp>
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

void print(std::ostream &os, int i)
{
  os << i << std::endl;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::bind(print, boost::ref(std::cout), std::placeholders::_1));
}
```

In Example 42.1, the function `print()` is passed to `std::for_each()` to write the numbers in **`v`** to an output stream. Because `print()` expects two parameters – an output stream and the number to be written – `std::bind()` is used. The first parameter passed to `print()` through `std::bind()` is **`std::cout`**. However, `print()` expects a reference to an output stream, while `std::bind()` passes parameters by value. Therefore, `boost::ref()` is used to wrap **`std::cout`**. `boost::ref()` returns a proxy object that contains a reference to the object passed to it. This makes it possible to pass a reference to **`std::cout`** even though `std::bind()` takes all parameters by value.
The function template `boost::cref()` lets you pass a `const` reference.

Chapter 43

Boost.Lambda

Before C++11, you needed to use a library like Boost.Lambda to take advantage of lambda functions. Since C++11, this library can be regarded as deprecated because lambda functions are now part of the programming language. If you work in a development environment that doesn't support C++11, you should consider Boost.Phoenix before you turn to Boost.Lambda. Boost.Phoenix is a newer library and probably the better choice if you need to use lambda functions without C++11.

The purpose of lambda functions is to make code more compact and easier to understand (see Example 43.1).

Example 43.1 `std::for_each()` with a lambda function

```
#include <boost/lambda/lambda.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::cout << boost::lambda::_1 << "\n");
}
```

Boost.Lambda provides several helpers to create nameless functions. Code is written where it should be executed, without needing to be wrapped in a function and without having to call a function explicitly. In Example 43.1, `std::cout << boost::lambda::_1 << "\n"` is a lambda function that expects one parameter, which it writes, followed by a new line, to standard output.

`boost::lambda::_1` is a placeholder that creates a lambda function that expects one parameter. The number in the placeholder determines the number of expected parameters, so **`boost::lambda::_2`** expects two parameters and **`boost::lambda::_3`** expects three parameters. Boost.Lambda only provides these three placeholders. The lambda function in Example 43.1 uses **`boost::lambda::_1`** because `std::for_each()` expects a unary function.

Include `boost/lambda/lambda.hpp` to use placeholders.

Please note that `\n`, instead of `std::endl`, is used in Example 43.1 to output a new line. If you use `std::endl`, the example won't compile because the type of the lambda function `std::cout << boost::lambda::_1` differs from what the unary function template `std::endl()` expects. Thus, you can't use `std::endl`.

Example 43.2 A lambda function with `boost::lambda::if_then()`

```
#include <boost/lambda/lambda.hpp>
#include <boost/lambda/if.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    boost::lambda::if_then(boost::lambda::_1 > 1,
    std::cout << boost::lambda::_1 << "\n"));
}
```

The header file `boost/lambda/if.hpp` defines constructs you can use to create `if` control structures in a lambda function. The simplest construct is the function template `boost::lambda::if_then()`, which expects two parameters: the first parameter is a condition. If the condition is true, the second parameter is executed. Both parameters can be lambda functions, as in Example 43.2.
In addition to `boost::lambda::if_then()`, Boost.Lambda provides the function templates `boost::lambda::if_then_else()` and `boost::lambda::if_then_else_return()`, both of which expect three parameters. Function templates are also provided `for` loops and cast operators and to throw exceptions in lambda functions. The many function templates defined by Boost.Lambda make it possible to define lambda functions that are in no way inferior to normal C++ functions.

Part X

Parallel Programming

The following libraries support the parallel programming model.

- Boost.Thread lets you create and manage your own threads.
- Boost.Atomic lets you access variables of integral types with atomic operations from multiple threads.
- Boost.Lockfree provides thread-safe containers.
- Boost.MPI originates from the supercomputer domain. With Boost.MPI your program is started multiple times and executed in multiple processes. You concentrate on programming the actual tasks that should be executed concurrently, and Boost.MPI coordinates the processes. With Boost.MPI you don't need to take care of details like synchronizing access on shared data. However, Boost.MPI does require an appropriate runtime environment.

Chapter 44

Boost.Thread

Boost.Thread is the library that allows you to use threads. Furthermore, it provides classes to synchronize access on data which is shared by multiple threads.
Threads have been supported by the standard library since C++11. You will also find classes in the standard library that threads can be created and synchronized with. While Boost.Thread resembles the standard library in many regards, it offers extensions. For example, you can interrupt threads created with Boost.Thread. You will also find special locks in Boost.Thread that will probably be added to the standard library with C++14. Thus, it can make sense to use Boost.Thread even if you work in a C++11 development environment.

44.1 Creating and Managing Threads

The most important class in this library is `boost::thread`, which is defined in `boost/thread.hpp`. This class is used to create a new thread. Example 44.1 is a simple example that creates a thread.
The name of the function that the new thread should execute is passed to the constructor of `boost::thread`. Once the variable **`t`** in Example 44.1 is created, the function `thread()` starts immediately executing in its own thread. At this point, `thread()` executes concurrently with the `main()` function.
To keep the program from terminating, `join()` is called on the newly created thread. `join()` blocks the current thread until the thread for which `join()` was called has terminated. This causes `main()` to wait until `thread()` returns.

Example 44.1 Using `boost::thread`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << '\n';
  }
}

int main()
{
  boost::thread t{thread};
  t.join();
}
```

A particular thread can be accessed using a variable – `t` in this example – to wait for its termination. However, the thread will continue to execute even if `t` goes out of scope and is destroyed. A thread is always bound to a variable of type `boost::thread` in the beginning, but once created, the thread no longer depends on that variable. There is even a member function called `detach()` that allows a variable of type `boost::thread` to be decoupled from its corresponding thread. It's not possible to call member functions like `join()` after calling `detach()` because the detached variable no longer represents a valid thread.

Anything that can be done inside a function can also be done inside a thread. Ultimately, a thread is no different from a function, except that it is executed concurrently to another function. In Example 44.1, five numbers are written to the standard output stream in a loop. To slow down the output, every iteration of the loop calls the `wait()` function to stall for one second. `wait()` uses the function `sleep_for()`, which is also provided by Boost.Thread and resides in the namespace `boost::this_thread`.

`sleep_for()` expects as its sole parameter a period of time that indicates how long the current thread should be stalled. By passing an object of type `boost::chrono::seconds`, a period of time is set. `boost::chrono::seconds` comes from Boost.Chrono which is introduced in Chapter 37.

`sleep_for()` only accepts types from Boost.Chrono. Even though Boost.Chrono has been part of the standard library with C++11, types from `std::chrono` cannot be used with Boost.Thread. Doing so will lead to compiler errors.

If you don't want to call `join()` at the end of `main()`, you can use the class `boost::scoped_thread`. The constructor of `boost::scoped_thread` expects an object of type `boost::thread`. In the destructor of `boost::scoped_thread` an action has access to that object. By default, `boost::scoped_thread` uses an action that calls `join()` on the thread. Thus, Example 44.2 works like Example 44.1.

Example 44.2 Waiting for a thread with `boost::scoped_thread`

```
#include <boost/thread.hpp>
#include <boost/thread/scoped_thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    std::cout << i << '\n';
  }
}

int main()
{
  boost::scoped_thread<> t{boost::thread{thread}};
}
```

You can pass a user-defined action as a template parameter. The action must be a class with an operator `operator()` that accepts an object of type `boost::thread`. `boost::scoped_thread` guarantees that the operator will be called in the destructor.

You can find the class `boost::scoped_thread` only in Boost.Thread. There is no counterpart in the standard library. Make sure you include the header file `boost/thread/scoped_thread.hpp` for `boost::scoped_thread`.

Example 44.3 introduces *interruption points*, which make it possible to interrupt threads. Interruption points are only supported by Boost.Thread and not by the standard library.

Calling `interrupt()` on a thread object interrupts the corresponding thread. In this context, interrupt means that an exception of type `boost::thread_interrupted` is thrown in the thread. However, this only happens when the thread reaches an interruption point.

Example 44.3 An interruption point with `boost::this_thread::sleep_for()`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
```

```
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << '\n';
    }
  }
  catch (boost::thread_interrupted&) {}
}

int main()
{
  boost::thread t{thread};
  wait(3);
  t.interrupt();
  t.join();
}
```

Simply calling `interrupt()` does not have an effect if the given thread does not contain an interruption point. Whenever a thread reaches an interruption point it will check whether `interrupt()` has been called. If it has been called, an exception of type `boost::thread_interrupted` will be thrown.
Boost.Thread defines a series of interruption points such as the `sleep_for()` function. Because `sleep_for()` is called five times in Example 44.3, the thread checks five times whether or not it has been interrupted. Between the calls to `sleep_for()`, the thread can not be interrupted.
Example 44.3 doesn't display five numbers, because `interrupt()` is called after three seconds in `main()`. Thus, the corresponding thread is interrupted and throws a `boost::thread_interrupted` exception. The exception is correctly caught inside the thread even though the `catch` handler is empty. Because the `thread()` function returns after the handler, the thread terminates as well. This, in turn, will cause the program to terminate because `main()` was waiting for the thread to terminate.
Boost.Thread defines about fifteen interruption points, including `sleep_for()`. These interruption points make it easy to interrupt threads in a timely manner. However, interruption points may not always be the best choice because they must be reached before the thread can check for a `boost::thread_interrupted` exception.

Example 44.4 Disabling interruption points with `disable_interruption`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{
  boost::this_thread::disable_interruption no_interruption;
  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << '\n';
    }
  }
  catch (boost::thread_interrupted&) {}
}

int main()
{
  boost::thread t{thread};
  wait(3);
  t.interrupt();
  t.join();
}
```

The class `boost::this_thread::disable_interruption` prevents a thread from being interrupted. If you instantiate `boost::this_thread::disable_interruption`, interruption points in a thread will be disabled as long as the object exists. Thus, Example 44.4 displays five numbers because the attempt to interrupt the thread is ignored.

Example 44.5 Setting thread attributes with `boost::thread::attributes`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

void thread()
{

  try
  {
    for (int i = 0; i < 5; ++i)
    {
      wait(1);
      std::cout << i << '\n';
    }
  }
  catch (boost::thread_interrupted&) {}
}

int main()
{
  boost::thread::attributes attrs;
  attrs.set_stack_size(1024);
  boost::thread t{attrs, thread};
  t.join();
}
```

`boost::thread::attributes` is used to set thread attributes. In version 1.56.0, you can only set one platform-independent attribute, the stack size. In Example 44.5, the stack size is set to 1024 bytes by `boost::thread::attributes::set_stack_size()`.

Example 44.6 Detecting the thread ID and number of available processors

```
#include <boost/thread.hpp>
#include <iostream>

int main()
{
  std::cout << boost::this_thread::get_id() << '\n';
  std::cout << boost::thread::hardware_concurrency() << '\n';
}
```

In the namespace `boost::this_thread`, free-standing functions are defined that apply to the current thread. One of these functions is `sleep_for()`, which we have seen before. Another one is `get_id()`, which returns a number to uniquely identify the current thread (see Example 44.6). `get_id()` is also provided as a member function by the class `boost::thread`.
The static member function `boost::thread::hardware_concurrency()` returns the number of threads that can physically be executed at the same time, based on the underlying number of CPUs or CPU cores. Calling this function on a dual-core processor returns a value of 2. This function provides a simple method to identify the theoretical maximum number of threads that should be used.
Boost.Thread also provides the class `boost::thread_group` to manage threads in groups. One function this class provides, the member function `join_all()`, waits for all threads in the group to terminate.

44.2 Synchronizing Threads

While using multiple threads can increase the performance of an application, it usually also increases complexity. If several functions execute at the same time, access to shared resources must be synchronized. This involves significant programming effort once the application reaches a certain size. This section introduces the classes provided by Boost.Thread to synchronize threads.

Example 44.7 Exclusive access with `boost::mutex`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::mutex mutex;

void thread()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    mutex.lock();
    std::cout << "Thread " << get_id() << ": " << i << std::endl;
    mutex.unlock();
  }
}

int main()
{
  boost::thread t1{thread};
  boost::thread t2{thread};
  t1.join();
  t2.join();
}
```

Multithreaded programs use *mutexes* for synchronization. Boost.Thread provides different mutex classes with `boost::mutex` being the simplest. The basic principle of a mutex is to prevent other threads from taking ownership while a particular thread owns the mutex. Once released, a different thread can take ownership. This causes threads to wait until the thread that owns the mutex has finished processing and releases its ownership of the mutex.
Example 44.7 uses a global mutex of type `boost::mutex` called **`mutex`**. The `thread()` function takes ownership of this object by calling `lock()`. This is done right before the function writes to the standard output stream. Once a message has been written, ownership is released by calling `unlock()`.
`main()` creates two threads, both of which are executing the `thread()` function. Each thread counts to five and writes a message to the standard output stream in each iteration of the `for` loop. Because **`std::cout`** is a global

object shared by the threads, access must be synchronized. Otherwise, messages could get mixed up. Synchronization guarantees that at any given time, only one thread has access to **`std::cout`**. Both threads try to acquire the mutex before writing to the standard output stream, but only one thread at a time actually accesses **`std::cout`**. No matter which thread successfully calls `lock()`, all other threads need to wait until `unlock()` has been called.

Example 44.8 `boost::lock_guard` with guaranteed mutex release

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::mutex mutex;

void thread()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::lock_guard<boost::mutex> lock{mutex};
    std::cout << "Thread " << get_id() << ": " << i << std::endl;
  }
}

int main()
{
  boost::thread t1{thread};
  boost::thread t2{thread};
  t1.join();
  t2.join();
}
```

Acquiring and releasing mutexes is a typical scheme and is supported by Boost.Thread through different types. For example, instead of using `lock()` and `unlock()`, you can use `boost::lock_guard`.
`boost::lock_guard` automatically calls `lock()` and `unlock()` in its constructor and its destructor, respectively. Access to the shared resource is synchronized in Example 44.8 just as it was when both member functions were called explicitly. The class `boost::lock_guard` is an example of the RAII idiom to make sure resources are released when they are no longer needed.
Besides `boost::mutex` and `boost::lock_guard`, Boost.Thread provides additional classes to support variants of synchronization. One of the essential ones is `boost::unique_lock` which provides several helpful member functions.
Example 44.9 uses two variants of the `thread()` function. Both variants still write five numbers in a loop to the standard output stream, but they now use the class `boost::unique_lock` to lock a mutex.
`thread1()` passes the variable **`mutex`** to the constructor of `boost::unique_lock`, which makes `boost::unique_lock` try to lock the mutex. In this case `boost::unique_lock` behaves no differently than `boost::lock_guard`. The constructor of `boost::unique_lock` calls `lock()` on the mutex.
However, the destructor of `boost::unique_lock` doesn't release the mutex in `thread1()`. In `thread1()` `release()` is called on the lock, which decouples the mutex from the lock. By default, the destructor of `boost::unique_lock` releases a mutex, like the destructor of `boost::lock_guard` – but not if the mutex is decoupled. That's why there is an explicit call to `unlock()` in `thread1()`.
`thread2()` passes **`mutex`** and **`boost::try_to_lock`** to the constructor of `boost::unique_lock`. This makes the constructor of `boost::unique_lock` not call `lock()` on the mutex but `try_lock()`. Thus, the constructor only tries to lock the mutex. If the mutex is owned by another thread, the try fails.
`owns_lock()` lets you detect whether `boost::unique_lock` was able to lock a mutex. If `owns_lock()` returns `true`, `thread2()` can access **`std::cout`** immediately. If `owns_lock()` returns `false`, `try_lock_for()` is called. This member function also tries to lock a mutex, but it waits for the mutex for a specified period of time before failing. In Example 44.9 the lock tries for one second to obtain the mutex. If `try_lock_for()`

returns `true`, **`std::cout`** may be accessed. Otherwise, `thread2()` gives up and skips a number. Thus, it is possible that the second thread in the example won't write five numbers to the standard output stream.
Please note that in Example 44.9, the type of **`mutex`** is `boost::timed_mutex`, not `boost::mutex`. The example uses `boost::timed_mutex` because this mutex is the only one that provides the member function `try_lock_for()`. This member function is called when `try_lock_for()` is called on the lock. `boost::mutex` provides only the member functions `lock()` and `try_lock()`.
`boost::unique_lock` is an *exclusive lock*. An exclusive lock is always the sole owner of a mutex. Another lock can only get control of the mutex after the exclusive lock has released it. Boost.Thread also supports *shared locks* with the class `boost::shared_lock`, which is used with `shared_mutex`.

Example 44.9 The versatile lock `boost::unique_lock`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::timed_mutex mutex;

void thread1()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::unique_lock<boost::timed_mutex> lock{mutex};
    std::cout << "Thread " << get_id() << ": " << i << std::endl;
    boost::timed_mutex *m = lock.release();
    m->unlock();
  }
}

void thread2()
{
  using boost::this_thread::get_id;
  for (int i = 0; i < 5; ++i)
  {
    wait(1);
    boost::unique_lock<boost::timed_mutex> lock{mutex,
      boost::try_to_lock};
    if (lock.owns_lock() || lock.try_lock_for(boost::chrono::seconds{1}))
    {
      std::cout << "Thread " << get_id() << ": " << i << std::endl;
    }
  }
}

int main()
{
  boost::thread t1{thread1};
  boost::thread t2{thread2};
  t1.join();
  t2.join();
}
```

Example 44.10 Shared locks with `boost::shared_lock`

```
#include <boost/thread.hpp>
#include <boost/chrono.hpp>
#include <iostream>
#include <vector>
```

```
#include <cstdlib>
#include <ctime>

void wait(int seconds)
{
  boost::this_thread::sleep_for(boost::chrono::seconds{seconds});
}

boost::shared_mutex mutex;
std::vector<int> random_numbers;

void fill()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::shared_mutex> lock{mutex};
    random_numbers.push_back(std::rand());
    lock.unlock();
    wait(1);
  }
}

void print()
{
  for (int i = 0; i < 3; ++i)
  {
    wait(1);
    boost::shared_lock<boost::shared_mutex> lock{mutex};
    std::cout << random_numbers.back() << '\n';
  }
}

int sum = 0;

void count()
{
  for (int i = 0; i < 3; ++i)
  {
    wait(1);
    boost::shared_lock<boost::shared_mutex> lock{mutex};
    sum += random_numbers.back();
  }
}

int main()
{
  boost::thread t1{fill}, t2{print}, t3{count};
  t1.join();
  t2.join();
  t3.join();
  std::cout << "Sum: " << sum << '\n';
}
```

Non-exclusive locks of type `boost::shared_lock` can be used if threads only need read-only access to a specific resource. A thread modifying the resource needs write access and thus requires an exclusive lock. Since a thread with read-only access is unaffected by other threads reading the same resource at the same time, it can use a non-exclusive lock and share a mutex.
In Example 44.10, both `print()` and `count()` only read the variable **`random_numbers`**. The `print()` function writes the last value in **`random_numbers`** to the standard output stream, and the `count()` function adds it to the variable **`sum`**. Because neither function modifies **`random_numbers`**, both can access it at the same time using a non-exclusive lock of type `boost::shared_lock`.
Inside the `fill()` function, an exclusive lock of type `boost::unique_lock` is required because it inserts new

random numbers into **`random_numbers`**. `fill()` releases the mutex using the `unlock()` member function and then waits for one second. Unlike the previous examples, `wait()` is called at the end of the `for` loop to guarantee that at least one random number is placed in the container before it is accessed by either `print()` or `count()`. Both of these functions call the `wait()` function at the beginning of their `for` loops.
Looking at the individual calls to the `wait()` function from different locations, one potential issue becomes apparent: The order of the function calls is directly affected by the order in which the CPU actually executes the individual threads. Using *condition variables*, the individual threads can be synchronized so that values added to **`random_numbers`** are immediately processed by a different thread.
Example 44.11 removes the `wait()` and `count()` functions. Threads no longer wait for one second in every iteration; rather, they execute as fast as possible. In addition, no total is calculated; numbers are just written to the standard output stream.
To ensure correct processing of the random numbers, the individual threads are synchronized using a condition variable, which can be checked for certain conditions between multiple threads.
As before, the `fill()` function generates a random number with each iteration and places it in the **`random_numbers`** container. To block other threads from accessing the container at the same time, an exclusive lock is used. Instead of waiting for one second, this example uses a condition variable. Calling `notify_all()` will wake up every thread that has been waiting for this notification with `wait()`.
Looking at the `for` loop of the `print()` function, you can see that the member function `wait()` is called for the same condition variable. When the thread is woken up by a call to `notify_all()`, it tries to acquire the mutex, which will only succeed after the mutex has been successfully released in the `fill()` function.

Example 44.11 Condition variables with `boost::condition_variable_any`

```
#include <boost/thread.hpp>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <ctime>

boost::mutex mutex;
boost::condition_variable_any cond;
std::vector<int> random_numbers;

void fill()
{
  std::srand(static_cast<unsigned int>(std::time(0)));
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::mutex> lock{mutex};
    random_numbers.push_back(std::rand());
    cond.notify_all();
    cond.wait(mutex);
  }
}

void print()
{
  std::size_t next_size = 1;
  for (int i = 0; i < 3; ++i)
  {
    boost::unique_lock<boost::mutex> lock{mutex};
    while (random_numbers.size() != next_size)
      cond.wait(mutex);
    std::cout << random_numbers.back() << '\n';
    ++next_size;
    cond.notify_all();
  }
}

int main()
{
  boost::thread t1{fill};
  boost::thread t2{print};
  t1.join();
  t2.join();
}
```

The trick here is that calling `wait()` also releases the mutex which was passed as a parameter. After calling `notify_all()`, the `fill()` function releases the mutex by calling `wait()`. It then blocks and waits for some other thread to call `notify_all()`, which happens in the `print()` function once the random number has been written to the standard output stream.
Notice that the call to the `wait()` member function inside the `print()` function actually happens within a separate `while` loop. This is done to handle the scenario where a random number has already been placed in the container before the `wait()` member function is called for the first time in `print()`. By comparing the number of stored elements in **`random_numbers`** with the expected number of elements, this scenario is successfully handled and the random number is written to the standard output stream.
Example 44.11 also works if the locks aren't local in the `for` loop but instantiated in the outer scope. In fact this makes more sense because the locks don't need to be destroyed and recreated in every iteration. Since the mutex is always released with `wait()`, you don't need to destroy the locks at the end of an iteration.

44.3 Thread Local Storage

Thread Local Storage (TLS) is a dedicated storage area that can only be accessed by one thread. TLS variables can be seen as global variables that are only visible to a particular thread and not the whole program.

Example 44.12 Synchronizing multiple threads with static variables

```
#include <boost/thread.hpp>
#include <iostream>

boost::mutex mutex;

void init()
{
  static bool done = false;
  boost::lock_guard<boost::mutex> lock{mutex};
  if (!done)
  {
    done = true;
    std::cout << "done" << '\n';
  }
}

void thread()
{
  init();
  init();
}

int main()
{
  boost::thread t[3];

  for (int i = 0; i < 3; ++i)
    t[i] = boost::thread{thread};

  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

Example 44.12 executes a function `thread()` in three threads. `thread()` calls another function `init()` twice, and `init()` checks whether the boolean variable **done** is `false`. If it is, the variable is set to `true` and `done` is written to standard output.
done is a static variable that is shared by all threads. If the first thread sets **done** to `true`, the second and third thread won't write `done` to standard output. The second call of `init()` in any thread won't write `done` to standard output either. The example will print `done` once.

A static variable like **done** can be used to do a one-time initialization in a process. To do a one-time initialization per thread, TLS can be used.

Example 44.13 Synchronizing multiple threads with TLS

```
#include <boost/thread.hpp>
#include <iostream>

boost::mutex mutex;

void init()
{
  static boost::thread_specific_ptr<bool> tls;
  if (!tls.get())
  {
    tls.reset(new bool{true});
    boost::lock_guard<boost::mutex> lock{mutex};
    std::cout << "done" << '\n';
  }
}

void thread()
{
  init();
  init();
}

int main()
{
  boost::thread t[3];

  for (int i = 0; i < 3; ++i)
    t[i] = boost::thread{thread};

  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

In Example 44.13, the static variable **done** has been replaced with a TLS variable, **tls**, which is based on the class template `boost::thread_specific_ptr` – instantiated with the type bool. In principle, **tls** works like **done**: It acts as a condition that indicates whether or not something has already been done. The crucial difference, however, is that the value stored by **tls** is only visible and available to the corresponding thread.
Once a variable of type `boost::thread_specific_ptr` is created, it can be set. This variable expects the address of a variable of type bool instead of the variable itself. Using the `reset()` member function, the address can be stored in **tls**. In Example 44.13, a variable of type bool is dynamically allocated and its address, returned by `new`, is stored in **tls**. To avoid setting **tls** every time `init()` is called, the member function `get()` is used to check whether an address has already been stored.
Because `boost::thread_specific_ptr` stores an address, this class behaves like a pointer. For example, it provides the member functions `operator*` and `operator->`, which work as you would expect them to work with pointers.
Example 44.13 prints `done` to standard output three times. Each thread prints `done` in the first call to `init()`. Because a TLS variable is used, each thread uses its own variable **tls**. When the first thread initializes **tls** with a pointer to a dynamically allocated boolean variable, the **tls** variables in the second and third thread are still uninitialized. Since TLS variables are global per thread, not global per process, using **tls** in one thread does not change the variable in any other thread.

44.4 Futures and Promises

Futures and promises are tools to pass data from one thread to another. While this could also be done with other capabilities, such as global variables, futures and promises work without them. Furthermore, you don't need to handle synchronization yourself.

A *future* is a variable which receives a value from another thread. If you access a future to get the value, you might need to wait until the other thread has provided the value. Boost.Thread provides `boost::future` to define a future. The class defines a member function `get()` to get the value. `get()` is a blocking function that may have to wait for another thread.
To set a value in a future, you need to use a *promise*, because `boost::future` does not provide a member function to set a value.
Boost.Thread provides the class `boost::promise`, which has a member function `set_value()`. You always use future and promise as a pair. You can get a future from a promise with `get_future()`. You can use the future and the promise in different threads. If a value is set in the promise in one thread, it can be fetched from the future in another thread.

Example 44.14 Using `boost::future` and `boost::promise`

```
#define BOOST_THREAD_PROVIDES_FUTURE
#include <boost/thread.hpp>
#include <boost/thread/future.hpp>
#include <functional>
#include <iostream>

void accumulate(boost::promise<int> &p)
{
  int sum = 0;
  for (int i = 0; i < 5; ++i)
    sum += i;
  p.set_value(sum);
}

int main()
{
  boost::promise<int> p;
  boost::future<int> f = p.get_future();
  boost::thread t{accumulate, std::ref(p)};
  std::cout << f.get() << '\n';
}
```

Example 44.14 uses a future and a promise. The future **`f`** is received from the promise **`p`**. A reference to the promise is then passed to the thread **`t`** which executes the `accumulate()` function. `accumulate()` calculates the total of all numbers between 0 and 5 and saves it in the promise. In `main()` `get()` is called on the future to write the total to standard output.
The future **`f`** and the promise **`p`** are linked. When `get()` is called on the future, the value stored in the promise with `set_value()` is returned. Because the example uses two threads, it is possible that `get()` will be called in `main()` before `accumulate()` has called `set_value()`. In that case, `get()` blocks until a value has been stored in the promise with `set_value()`.
Example 44.14 displays `10`.
`accumulate()` had to be adapted to be executed in a thread. It has to take a parameter of type `boost::promise` and store the result in it. Example 44.15 introduces `boost::packaged_task`, a class that forwards a value from any function to a future as long as the function returns the result with `return`.

Example 44.15 Using `boost::packaged_task`

```
#define BOOST_THREAD_PROVIDES_FUTURE
#include <boost/thread.hpp>
#include <boost/thread/future.hpp>
#include <utility>
#include <iostream>

int accumulate()
{
  int sum = 0;
  for (int i = 0; i < 5; ++i)
    sum += i;
  return sum;
}

int main()
```

```
{
  boost::packaged_task<int> task{accumulate};
  boost::future<int> f = task.get_future();
  boost::thread t{std::move(task)};
  std::cout << f.get() << '\n';
}
```

Example 44.15 is similar to the previous one, but this time `boost::promise` is not used. Instead, this example uses the class `boost::packaged_task`, which, like `boost::promise`, provides a member function `get_future()` that returns a future.
The constructor of `boost::packaged_task` expects as a parameter the function that will be executed in a thread, but `boost::packaged_task` doesn't start a thread itself. An object of type `boost::packaged_task` has to be passed to the constructor of `boost::thread` for the function to be executed in a new thread.
The advantage of `boost::packaged_task` is that it stores the return value of a function in a future. You don't need to adapt a function to store its value in a future. `boost::packaged_task` can be seen as an adapter which can store the return value of any function in a future.
While the example got rid of `boost::promise`, the following example doesn't use `boost::packaged_task` and `boost::thread` either.

Example 44.16 Using `boost::async()`

```
#define BOOST_THREAD_PROVIDES_FUTURE
#include <boost/thread.hpp>
#include <boost/thread/future.hpp>
#include <iostream>

int accumulate()
{
  int sum = 0;
  for (int i = 0; i < 5; ++i)
    sum += i;
  return sum;
}

int main()
{
  boost::future<int> f = boost::async(accumulate);
  std::cout << f.get() << '\n';
}
```

In Example 44.16 `accumulate()` is passed to the function `boost::async()`. This function unifies `boost::packaged_task` and `boost::thread`. It starts `accumulate()` in a new thread and returns a future.
It is possible to pass a launch policy to `boost::async()`. This additional parameter determines whether `boost::async()` will execute the function in a new thread or in the current thread. If you pass **`boost::launch::async`**, `boost::async()` will start a new thread; this is the default behavior. If you pass **`boost::launch::deferred`**, the function will be executed in the current thread when `get()` is called.
Although Boost 1.56.0 allows either **`boost::launch::async`** or **`boost::launch::deferred`** to be passed to `boost::async()`, executing the function in the current thread is not yet implemented. If you pass **`boost::launch::deferred`**, the program will terminate.

Chapter 45

Boost.Atomic

Boost.Atomic provides the class `boost::atomic`, which can be used to create atomic variables. They are called atomic variables because all access is atomic. Boost.Atomic is used in multithreaded programs when access to a variable in one thread shouldn't be interrupted by another thread accessing the same variable. Without `boost::atomic`, attempts to access shared variables from multiple threads would need to be synchronized with locks. `boost::atomic` depends on the target platform supporting atomic variable access. Otherwise, `boost::atomic` uses locks. The library allows you to detect whether a target platform supports atomic variable access.

If your development environment supports C++11, you don't need Boost.Atomic. The C++11 standard library provides a header file `atomic` that defines the same functionality as Boost.Atomic. For example, you will find a class named `std::atomic`.

Boost.Atomic supports more or less the same functionality as the standard library. While a few functions are overloaded in Boost.Atomic, they may have different names in the standard library. The standard library also provides some functions, such as `std::atomic_init()` and `std::kill_dependency()`, which are missing in Boost.Atomic.

Example 45.1 Using `boost::atomic`

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  ++a;
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.1 uses two threads to increment the int variable **a**. Instead of a lock, the example uses `boost::atomic` for atomic access to **a**. The example writes `2` to standard output.

`boost::atomic` works because some processors support atomic access on variables. If incrementing an int variable is an atomic operation, a lock isn't required. If this example is run on a platform that cannot increment a variable as an atomic operation, `boost::atomic` uses a lock.

Example 45.2 `boost::atomic` with or without lock

```
#include <boost/atomic.hpp>
#include <iostream>

int main()
{
  std::cout.setf(std::ios::boolalpha);

  boost::atomic<short> s;
  std::cout << s.is_lock_free() << '\n';

  boost::atomic<int> i;
  std::cout << i.is_lock_free() << '\n';

  boost::atomic<long> l;
  std::cout << l.is_lock_free() << '\n';
}
```

You can call `is_lock_free()` on an atomic variable to check whether accessing the variable is done without a lock. If you run the example on an Intel x86 processor, it displays `true` three times. If you run it on a processor without lock-free access on short, int and long variables, `false` is displayed.
Boost.Atomic provides the `BOOST_ATOMIC_INT_LOCK_FREE` and `BOOST_ATOMIC_LONG_LOCK_FREE` macros to check, at compile time, which data types support lock-free access.
Example 45.2 uses integral data types only. You should not use `boost::atomic` with classes like `std::string` or `std::vector`. Boost.Atomic supports integers, pointers, booleans (bool), and trivial classes. Examples of integral types include short, int and long. Trivial classes define objects that can be copied with `std::memcpy()`.

Example 45.3 `boost::atomic` with **`boost::memory_order_seq_cst`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_seq_cst);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.3 increments **a** twice – this time not with `operator++` but with a call to `fetch_add()`. The member function `fetch_add()` can take two parameters: the number by which **a** should be incremented and the *memory order*.
The memory order specifies the order in which access operations on memory must occur. By default, this order is undetermined and does not depend on the order of the lines of code. Compilers and processors are allowed to change the order as long as a program behaves as if memory access operations were executed in source code order. This rule only applies to a thread. If more than one thread is used, variations in the order of memory accesses can lead to a program acting erroneously. Boost.Atomic supports specifying a memory order when accessing variables to make sure memory accesses occur in the desired order in a multithreaded program.

Note

Specifying memory order optimizes performance, but it increases complexity and makes it more difficult to write correct code. Therefore, in practice, you should have a really good reason for using memory orders.

Example 45.3 uses the memory order **`boost::memory_order_seq_cst`** to increment **`a`** by 1. The memory order stands for *sequential consistency*. This is the most restrictive memory order. All memory accesses that appear before the `fetch_add()` call must occur before this member function is executed. All memory accesses that appear after the `fetch_add()` call must occur after this member function is executed. Compilers and processors may reorder memory accesses before and after the call to `fetch_add()`, but they must not move a memory access from before to after the call to `fetch_add()` or vice versa. **`boost::memory_order_seq_cst`** is a strict boundary for memory accesses in both directions.
`boost::memory_order_seq_cst` is the most restrictive memory order. It is used by default when memory order is not set. Therefore, in Example 45.1, when **`a`** is incremented with `operator++`, **`boost::memory_order_seq_cst`** will be used.
`boost::memory_order_seq_cst` isn't always necessary. For example, in Example 45.3 there is no need to synchronize memory accesses for other variables because **`a`** is the only variable the threads use. **`a`** is written to standard output in `main()`, but only after both threads have terminated. The call to `join()` guarantees that **`a`** is only read after both threads have finished.

Example 45.4 `boost::atomic` with **`boost::memory_order_relaxed`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_relaxed);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.4 sets the memory order to **`boost::memory_order_relaxed`**. This is the least restrictive memory order: it allows arbitrary reordering of memory accesses. This example works with this memory order because the threads access no variables except **`a`**. Therefore, no specific order is required.

Example 45.5 `boost::atomic` with **`memory_order_release`** and **`memory_order_acquire`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};
int b = 0;

void thread1()
{
  b = 1;
  a.store(1, boost::memory_order_release);
}

void thread2()
{
  while (a.load(boost::memory_order_acquire) != 1)
    ;
  std::cout << b << '\n';
}

int main()
{
  std::thread t1{thread1};
  std::thread t2{thread2};
  t1.join();
  t2.join();
}
```

There are choices between the most restrictive memory order, **`boost::memory_order_seq_cst`**, and the least restrictive one, **`boost::memory_order_relaxed`**. Example 45.5 introduces the memory orders **`boost::memory_order_release`** and **`boost::memory_order_acquire`**.
Memory accesses that appear in the code before the **`boost::memory_order_release`** statement are executed before the **`boost::memory_order_release`** statement is executed. Compilers and processors must not move memory accesses from before to after **`boost::memory_order_release`**. However, they may move memory accesses from after to before **`boost::memory_order_release`**.
`boost::memory_order_acquire` works like **`boost::memory_order_release`**, but refers to memory accesses after **`boost::memory_order_acquire`**. Compilers and processors must not move memory accesses from after the **`boost::memory_order_acquire`** statement to before it. However, they may move memory accesses from before to after **`boost::memory_order_acquire`**.
Example 45.5 uses **`boost::memory_order_release`** in the first thread to make sure that **`b`** is set to 1 before **`a`** is set to 1. **`boost::memory_order_release`** guarantees that the memory access on **`b`** occurs before the memory access on **`a`**.
To specify a memory order when accessing **`a`**, `store()` is called. This member function corresponds to an assignment with `operator=`.
The second thread reads **`a`** in a loop. This is done with the member function `load()`. Again, no assignment operator is used.
In the second thread, **`boost::memory_order_acquire`** makes sure that the memory access on **`b`** doesn't occur before the memory access on **`a`**. The second thread waits in the loop for **`a`** to be set to 1 by the first thread. Once this happens, **`b`** is read.
The example writes `1` to standard output. The memory orders ensure that all memory accesses occur in the right order. The first thread always writes 1 to **`b`** first before the second thread accesses and reads **`b`**.

Tip

For more details and examples on how memory orders work, you can find an explanation in the GCC Wiki article on memory model synchronization modes.

Chapter 46

Boost.Lockfree

Boost.Lockfree provides thread-safe and lock-free containers. Containers from this library can be accessed from multiple threads without having to synchronize access.

In version 1.56.0, Boost.Lockfree provides only two containers: a queue of type `boost::lockfree::queue` and a stack of type `boost::lockfree::stack`. For the queue, a second implementation is available: `boost::lockfree::spsc_queue`. This class is optimized for use cases where exactly one thread writes to the queue and exactly one thread reads from the queue. The abbreviation spsc in the class name stands for single producer/single consumer.

Example 46.1 uses the container `boost::lockfree::spsc_queue`. The first thread, which executes the function `produce()`, adds the numbers 1 to 100 to the container. The second thread, which executes `consume()`, reads the numbers from the container and adds them up in **`sum`**. Because the container `boost::lockfree::spsc_queue` explicitly supports concurrent access from two threads, it isn't necessary to synchronize the threads.

Please note that the function `consume()` is called a second time after the threads terminate. This is required to calculate the total of all 100 numbers, which is 5050. Because `consume()` accesses the queue in a loop, it is possible that it will read the numbers faster than they are inserted by `produce()`. If the queue is empty, `pop()` returns `false`. Thus, the thread executing `consume()` could terminate because `produce()` in the other thread couldn't fill the queue fast enough. If the thread executing `produce()` is terminated, then it's clear that all of the numbers were added to the queue. The second call to `consume()` makes sure that numbers that may not have been read yet are added to **`sum`**.

Example 46.1 Using `boost::lockfree::spsc_queue`

```
#include <boost/lockfree/spsc_queue.hpp>
#include <thread>
#include <iostream>

boost::lockfree::spsc_queue<int> q{100};
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  consume();
  std::cout << sum << '\n';
}
```

The size of the queue is passed to the constructor. Because `boost::lockfree::spsc_queue` is implemented with a circular buffer, the queue in Example 46.1 has a capacity of 100 elements. If a value can't be added because the queue is full, `push()` returns `false`. The example doesn't check the return value of `push()` because exactly 100 numbers are added to the queue. Thus, 100 elements is sufficient.

Example 46.2 works like the previous example, but this time the size of the circular buffer is set at compile time. This is done with the template `boost::lockfree::capacity`, which expects the capacity as a template parameter. **q** is instantiated with the default constructor – the capacity cannot be set at run time.

The function `consume()` has been changed to use `consume_one()`, rather than `pop()`, to read the number. A lambda function is passed as a parameter to `consume_one()`. `consume_one()` reads a number just like `pop()`, but the number isn't returned through a reference to the caller. It is passed as the sole parameter to the lambda function.

When the threads terminate, `main()` calls the member function `consume_all()`, instead of `consume()`. `consume_all()` works like `consume_one()` but makes sure that the queue is empty after the call. `consume_all()` calls the lambda function as long as there are elements in the queue.

Example 46.2 `boost::lockfree::spsc_queue` with `boost::lockfree::capacity`

```
#include <boost/lockfree/spsc_queue.hpp>
#include <boost/lockfree/policies.hpp>
#include <thread>
#include <iostream>

using namespace boost::lockfree;

spsc_queue<int, capacity<100>> q;
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  while (q.consume_one([](int i){ sum += i; }))
    ;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  q.consume_all([](int i){ sum += i; });
  std::cout << sum << '\n';
}
```

Once again, Example 46.2 writes `5050` to standard output.

Example 46.3 executes `consume()` in two threads. Because more than one thread reads from the queue, the class `boost::lockfree::spsc_queue` must not be used. This example uses `boost::lockfree::queue` instead. Thanks to `std::atomic`, access to the variable **`sum`** is also now thread safe.

The size of the queue is set to 100 – this is the parameter passed to the constructor. However, this is only the initial size. By default, `boost::lockfree::queue` is not implemented with a circular buffer. If more items are added to the queue than the capacity is set to, it is automatically increased. `boost::lockfree::queue` dynamically allocates additional memory if the initial size isn't sufficient.

Example 46.3 `boost::lockfree::queue` with variable container size

```
#include <boost/lockfree/queue.hpp>
#include <thread>
#include <atomic>
#include <iostream>

boost::lockfree::queue<int> q{100};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

That means that `boost::lockfree::queue` isn't necessarily lock free. The allocator used by `boost::lockfree::queue` by default is `boost::lockfree::allocator`, which is based on `std::allocator`. Thus, this allocator determines whether `boost::lockfree::queue` is lock free without constraints.
Example 46.4 uses a constant size of 10,000 elements. In this example, the queue doesn't allocate additional memory when it is full. 10,000 is a fixed upper limit.
The queue's capacity is constant because `boost::lockfree::fixed_sized` is passed as a template parameter. The capacity is passed as a parameter to the constructor and can be updated at any time using `reserve()`. If the capacity must be set at compile time, `boost::lockfree::capacity` can be passed as a template parameter to `boost::lockfree::queue`. `boost::lockfree::capacity` includes `boost::lockfree::fixed_sized`.

Example 46.4 `boost::lockfree::queue` with constant container size

```
#include <boost/lockfree/queue.hpp>
#include <thread>
#include <atomic>
#include <iostream>

using namespace boost::lockfree;

queue<int, fixed_sized<true>> q{10000};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

In Example 46.4, the queue has a capacity of 10,000 elements. Because `consume()` inserts 10,000 numbers into the queue, the upper limit isn't exceeded. If it were exceeded, `push()` would return `false`.
`boost::lockfree::queue` is similar to `boost::lockfree::spsc_queue` and also provides member functions like `consume_one()` and `consume_all()`.
The third class, `boost::lockfree::stack`, is similar to the other ones. As with `boost::lockfree::queue`, `boost::lockfree::fixed_size` and `boost::lockfree::capacity` can be passed as template parameters. The member functions are similar, too.

Chapter 47

Boost.MPI

Boost.MPI provides an interface to the MPI standard (Message Passing Interface). This standard simplifies the development of programs that execute tasks concurrently. You could develop such programs using threads or by making multiple processes communicate with each other through shared memory or network connections. The advantage of MPI is that you don't need to take care of such details. You can fully concentrate on parallelizing your program.

A disadvantage is that you need an MPI runtime environment. MPI is only an option if you control the runtime environment. For example, if you want to distribute a program that can be started with a double click, you won't be able to use MPI. While operating systems support threads, shared memory, and networks out of the box, they usually don't provide an MPI runtime environment. Users need to perform additional steps to start an MPI program.

47.1 Development and Runtime Environment

MPI defines functions for *parallel computing*. Parallel computing refers to programs that can execute tasks concurrently in runtime environments that support the parallel execution of tasks. Such runtime environments are usually based on multiple processors. Because a single processor can only execute code sequentially, linking multiple processors creates a runtime environment that can execute tasks in parallel. If thousands of processors are linked, the result is a parallel computer – a type of architecture usually found only in supercomputers. MPI comes from the search to find methods for programming supercomputers more easily.

If you want to use MPI, you need an implementation of the standard. While MPI defines many functions, they are usually not supported by operating systems out of the box. For example, the desktop editions of Windows aren't shipped with MPI support.

The most important MPI implementations are MPICH and Open MPI. MPICH is one of the earliest MPI implementations. It has existed since the mid 1990s. MPICH is a mature and portable implementation that is actively maintained and updated. The first version of Open MPI was released 2005. Because Open MPI is a collaborative effort that includes many developers who were responsible for earlier MPI implementations, Open MPI is seen as the future standard. However, that doesn't mean that MPICH can be ignored. There are several MPI implementations that are based on MPICH. For example, Microsoft ships an MPI implementation called Microsoft HPC Pack that is based on MPICH.

MPICH provides installation files for various operating systems, such as Windows, Linux, and OS X. If you need an MPI implementation and don't want to build it from source code, the MPICH installation files are the fastest path to start using MPI.

The MPICH installation files contain the required header files and libraries to develop MPI programs. Furthermore, they contain an MPI runtime environment. Because MPI programs execute tasks on several processors concurrently, they run in several processes. An MPI program is started multiple times, not just once. Several instances of the same MPI program run on multiple processors and communicate through functions defined by the MPI standard.

You can't start an MPI program with a double click. You use a helper program, which is usually called mpiexec. You pass your MPI program to mpiexec, which starts your program in the MPI runtime environment. Command line options determine how many processes are started and how they communicate – for example, through sockets or shared memory. Because the MPI runtime environment takes care of these details, you can concentrate on parallel programming.

If you decided to use the installation files from MPICH, note that MPICH only provides a 64-bit version. You must use a 64-bit compiler to develop MPI programs with MPICH and build a 64-bit version of Boost.MPI.

47.2 Simple Data Exchange

Boost.MPI is a C++ interface to the MPI standard. The library uses the namespace `boost::mpi`. It is sufficient to include the header file `boost/mpi.hpp` to get access to all classes and functions.

Example 47.1 MPI environment and communicator

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::cout << world.rank() << ", " << world.size() << '\n';
}
```

Example 47.1 is a simple MPI program. It uses two classes that you will find in all of the examples that follow. `boost::mpi::environment` initializes MPI. The constructor calls the function `MPI_Init()` from the MPI standard. The destructor calls `MPI_Finalize()`. `boost::mpi::communicator` is used to create a *communicator*. The communicator is one of the central concepts of MPI and supports data exchange between processes.
`boost::mpi::environment` is a very simple class that provides only a few member functions. You can call `initialized()` to check whether MPI has been initialized successfully. The member function returns a value of type bool. `processor_name()` returns the name of the current process as a `std::string`. And `abort()` stops an MPI program, not just the current process. You pass an int value to `abort()`. This value will be passed to the MPI runtime environment as the return value of the MPI program. For most MPI programs you won't need these member functions. You usually instantiate `boost::mpi::environment` at the beginning of a program and then don't use the object afterwards – as in Example 47.1 and the following examples in this chapter.
`boost::mpi::communicator` is more interesting. This class is a communicator that links the processes that are part of an MPI program. Every process has a rank, which is an integer – all processes are enumerated. A process can discover its rank by calling `rank()` on the communicator. If a process wants to know how many processes there are, it calls `size()`.
To run Example 47.1, you have to use a helper program provided by the MPI implementation you are using. With MPICH, the helper program is called mpiexec. You can run Example 47.1 using this helper with the following command:

```
mpiexec -n 4 sample.exe
```

mpiexec expects the name of an MPI program and an option that tells it how many processes to launch. The option `-n 4` tells mpiexec to launch four processes. Thus the MPI program is started four times. However, the four processes aren't independent. They are linked through the MPI runtime environment, and they all belong to the same communicator, which gave each process a rank. If you run Example 47.1 with four processes, `rank()` returns a number from 0 to 3 and `size()` 4.
Please note that the output can be mixed up. After all, four processes are writing to the standard output stream at the same time. For example, it's unknown whether the process with rank 0, or any other rank, is the first one to write to the standard output stream. It's also possible that one process will interrupt another one while writing to the standard output stream. The interrupted process might not be able to complete writing its rank and the size of the communicator before another process writes to the standard output stream, breaking up the output.

Example 47.2 Blocking functions to send and receive data

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    int i;
    world.recv(1, 16, i);
    std::cout << i << '\n';
  }
  else if (world.rank() == 1)
  {
    world.send(0, 16, 99);
  }
}
```

`boost::mpi::communicator` provides two simple member functions, `send()` and `recv()`, to exchange data between two processes. They are blocking functions that only return when data has been sent or received. This is especially important for `recv()`. If `recv()` is called without another process sending it data, the call blocks and the process will stall in the call.
In Example 47.2, the process with rank 0 receives data with `recv()`. The process with rank 1 sends data with `send()`. If you start the program with more than two processes, the other processes exit without doing anything. You pass three parameters to `send()`: The first parameter is the rank of the process to which data should be sent. The second parameter is a *tag* to identify data. The third parameter is the data.
The tag is always an integer. In Example 47.2 the tag is 16. The tag makes it possible to identify a call to `send()`. You'll see that the tag is used with `recv()`.
The third parameter passed to `send()` is 99. This number is sent from the process with rank 1 to the process with rank 0. Boost.MPI supports all primitive types. An int value like 99 can be sent directly
`recv()` expects similar parameters. The first parameter is the rank of the process from which data should be received. The second parameter is the tag that links the call to `recv()` with the call to `send()`. The third parameter is the variable to store the received data in.
If you run Example 47.2 with at least two processes, `99` is displayed.

Example 47.3 Receiving data from any process

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    int i;
    world.recv(boost::mpi::any_source, 16, i);
    std::cout << i << '\n';
  }
  else
  {
    world.send(0, 16, world.rank());
  }
}
```

Example 47.3 is based on Example 47.2. Instead of sending the number 99, it sends the rank of the process that calls `send()`. This could be any process with a rank greater than 0.
The call to `recv()` for the process with rank 0 has changed, too. **`boost::mpi::any_source`** is the first parameter. This means the call to `recv()` will accept data from any process that sends data with the tag 16.
If you start Example 47.3 with two processes, `1` will be displayed. After all, there is only one process that can call `send()` – the process with rank 1. If you start the program with more than two processes, it's unknown which number will be displayed. In this case, multiple processes will call `send()` and try to send their rank. Which process is first and, therefore, which rank is displayed, is random.
`recv()` has a return value of type `boost::mpi::status`. This class provides a member function `source()`, which returns the rank of the process from which data was received. Example 47.4 tells you from which process the number 99 was received.

Example 47.4 Detecting the sender with `boost::mpi::status`

```
#include <boost/mpi.hpp>
#include <iostream>
```

```
int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    int i;
    boost::mpi::status s = world.recv(boost::mpi::any_source, 16, i);
    std::cout << s.source() << ": " << i << '\n';
  }
  else
  {
    world.send(0, 16, 99);
  }
}
```

So far, all examples have used `send()` and `recv()` to transmit an int value. In Example 47.5 a string is transmitted.
`send()` and `recv()` can transmit arrays as well as single values. Example 47.5 transmits a string in a char array. Because `send()` and `recv()` support primitive types like char, the char array can be transmitted without any problems.

Example 47.5 Transmitting an array with `send()` and `recv()`

```
#include <boost/mpi.hpp>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    char buffer[14];
    world.recv(boost::mpi::any_source, 16, buffer, 13);
    buffer[13] = '\0';
    std::cout << buffer << '\n';
  }
  else
  {
    const char *c = "Hello, world!";
    world.send(0, 16, c, 13);
  }
}
```

`send()` takes a pointer to the string as its third parameter and the size of the string as its fourth parameter. The third parameter passed to `recv()` is a pointer to an array to store the received data. The fourth parameter tells `recv()` how many chars should be received and stored in **`buffer`**. Example 47.5 writes `Hello, world!` to the standard output stream.

Example 47.6 Transmitting a string with `send()` and `recv()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    std::string s;
    world.recv(boost::mpi::any_source, 16, s);
    std::cout << s << '\n';
  }
  else
  {
    std::string s = "Hello, world!";
    world.send(0, 16, s);
  }
}
```

Even though Boost.MPI supports only primitive types, that doesn't mean it's impossible to transmit objects of non-primitive types. Boost.MPI works together with Boost.Serialization. Objects that can be serialized according to the rules of Boost.Serialization can be transmitted with Boost.MPI.
Example 47.6 transmits "Hello, world!" This time the value transmitted is not a char array but a `std::string`. Boost.Serialization provides the header file `boost/serialization/string.hpp`, which only needs to be included to make `std::string` serializable.
If you want to transmit objects of user-defined types, see Chapter 64.

47.3 Asynchronous data exchange

In addition to the blocking functions `send()` and `recv()`, Boost.MPI also supports asynchronous data exchange with the member functions `isend()` and `irecv()`. The names start with an i to indicate that the functions return immediately.

Example 47.7 Receiving data asynchronously with `irecv()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    std::string s;
    boost::mpi::request r = world.irecv(boost::mpi::any_source, 16, s);
    if (r.test())
      std::cout << s << '\n';
    else
      r.cancel();
  }
  else
  {
    std::string s = "Hello, world!";
    world.send(0, 16, s);
  }
}
```

Example 47.7 uses the blocking function `send()` to send the string "Hello, world!" However, data is received with the asynchronous function `irecv()`. This member function expects the same parameters as `recv()`. The difference is that there is no guarantee that data has been received in **s** when `irecv()` returns.
`irecv()` returns an object of type `boost::mpi::request`. You can call `test()` to check whether data has been received. This member function returns a bool. You can call `test()` as often as you like. Because `irecv()` is an asynchronous member function, it is possible that the first call will return `false` and the second `true`. This would mean that the asynchronous operation was completed between the two calls.
Example 47.7 calls `test()` only once. If data has been received in **s**, the variable is written to the standard output stream. If no data has been received, the asynchronous operation is canceled with `cancel()`.
If you run Example 47.7 multiple times, sometimes `Hello, world!` is displayed and sometimes there is no output. The outcome depends on whether data is received before `test()` is called.

You can call `test()` on `boost::mpi::request` multiple times to detect when an asynchronous operation is complete. However, you can also call the blocking function `boost::mpi::wait_all()` as in Example 47.8. `boost::mpi::wait_all()` is a blocking function, but the advantage is that it can wait for multiple asynchronous operations to complete. `boost::mpi::wait_all()` returns when all asynchronous operations it is waiting for have been completed.

Example 47.8 Waiting for multiple asynchronous operations with `wait_all()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    boost::mpi::request requests[2];
    std::string s[2];
    requests[0] = world.irecv(1, 16, s[0]);
    requests[1] = world.irecv(2, 16, s[1]);
    boost::mpi::wait_all(requests, requests + 2);
    std::cout << s[0] << "; " << s[1] << '\n';
  }
  else if (world.rank() == 1)
  {
    std::string s = "Hello, world!";
    world.send(0, 16, s);
  }
  else if (world.rank() == 2)
  {
    std::string s = "Hello, moon!";
    world.send(0, 16, s);
  }
}
```

In Example 47.8, the process with rank 1 sends “Hello, world!” and the process with rank 2 “Hello, moon!” Since the order in which data is received doesn’t matter, the process with rank 0 calls `irecv()`. Since the program will only generate output when all asynchronous operations have been completed and all data has been received, the return values of type `boost::mpi::request` are passed to `boost::mpi::wait_all()`. Because `boost::mpi::wait_all()` expects two iterators, the objects of type `boost::mpi::request` are stored in an array. The begin and end iterators are passed to `boost::mpi::wait_all()`.

Boost.MPI provides additional functions you can use to wait for the completion of asynchronous operations. `boost::mpi::wait_any()` returns when exactly one asynchronous operation is complete, and `boost::mpi::wait_some()` returns when at least one asynchronous operation has been completed. Both functions return a `std::pair` that indicates which operation or operations are complete.

`boost::mpi::test_all()`, `boost::mpi::test_any()`, and `boost::mpi::test_some()` test the status of multiple asynchronous operations with a single call. These functions are non-blocking and return immediately.

47.4 Collective Data Exchange

The functions introduced so far share a one-to-one relationship: that is, one process sends and one process receives. The link is established through a tag. This section introduces functions that are called with the same parameters in multiple processes but execute different operations. For one process the function might send data, for another process it might receive data. These functions are called *collective operations*.

Example 47.9 Receiving data from multiple processes with `gather()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <vector>
```

```
#include <string>
#include <iterator>
#include <algorithm>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  if (world.rank() == 0)
  {
    std::vector<std::string> v;
    boost::mpi::gather<std::string>(world, "", v, 0);
    std::ostream_iterator<std::string> out{std::cout, "\n"};
    std::copy(v.begin(), v.end(), out);
  }
  else if (world.rank() == 1)
  {
    boost::mpi::gather(world, std::string{"Hello, world!"}, 0);
  }
  else if (world.rank() == 2)
  {
    boost::mpi::gather(world, std::string{"Hello, moon!"}, 0);
  }
}
```

Example 47.9 calls the function `boost::mpi::gather()` in multiple processes. Whether the function sends or receives depends on the parameters.
The processes with the ranks 1 and 2 use `boost::mpi::gather()` to send data. They pass, as parameters, the data being sent – the strings "Hello, world!" and "Hello, moon!" – and the rank of the process the data should be transmitted to. Since `boost::mpi::gather()` isn't a member function, the communicator **`world`** also has to be passed.
The process with rank 0 calls `boost::mpi::gather()` to receive data. Since the data has to be stored somewhere, an object of type std::vector<std::string> is passed. Please note that you have to use this type with `boost::mpi::gather()`. No other containers or string types are supported.
The process with rank 0 has to pass the same parameters as the processes with rank 1 and 2. That's why the process with rank 0 also passes **`world`**, a string to send, and 0 to `boost::mpi::gather()`.
If you start Example 47.9 with three processes, `Hello, world!` and `Hello, moon!` are displayed. If you look at the output carefully, you'll notice that an empty line is written first. The first line is the empty string the process with rank 0 passes to `boost::mpi::gather()`. There are three strings in **`v`** which were received from the processes with the ranks 0, 1 and 2. The indexes of the elements in the vector correspond to the ranks of the processes. If you run the example multiple times, you'll always get an empty string as a first element in the vector, "Hello, world!" as the second element and "Hello, moon!" as the third one.
Please note that you must not run Example 47.9 with more than three processes. If you start mpiexec with, for example, `-n 4`, no data is displayed. The program will hang and will have to be aborted with CTRL+C.
Collective operations must be executed for all processes. If your program calls a function such as `boost::mpi::gather()`, you have to make sure that the function is called in all processes. Otherwise it's a violation of the MPI standard. Because a function like `boost::mpi::gather()` has to be called by all processes, the call is usually not different per process, as in Example 47.9. Compare the previous example with Example 47.10, which does the same thing.

Example 47.10 Collecting data from all processes with `gather()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <vector>
#include <string>
#include <iterator>
#include <algorithm>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 1)
    s = "Hello, world!";
  else if (world.rank() == 2)
    s = "Hello, moon!";
  std::vector<std::string> v;
  boost::mpi::gather(world, s, v, 0);
  std::ostream_iterator<std::string> out{std::cout, "\n"};
  std::copy(v.begin(), v.end(), out);
}
```

You call functions for collective operations in all processes. Usually the functions are defined in a way that it's clear which operation has to be executed, even if all processes pass the same parameters.
Example 47.10 uses `boost::mpi::gather()`, which gathers data. The data is gathered in the process whose rank is passed as the last parameter to `boost::mpi::gather()`. This process gathers the data it receives from all processes. The vector to store data is used exclusively by the process that gathers data.
`boost::mpi::gather()` gathers data from all processes. This includes the process that gathers data. In Example 47.10, that is the process with rank 0. This process sends an empty string to itself in **s**. The empty string is stored in **v**. As you'll see in the following examples, collective operations always include all processes.
You can run Example 47.10 with as many processes as you like because every process calls `boost::mpi::gather()`. If you run the example with three processes, the result will be similar to the previous example.

Example 47.11 Scattering data with `scatter()` across all processes

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <vector>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::vector<std::string> v{"Hello, world!", "Hello, moon!",
    "Hello, sun!"};
  std::string s;
  boost::mpi::scatter(world, v, s, 0);
  std::cout << world.rank() << ": " << s << '\n';
}
```

Example 47.11 introduces the function `boost::mpi::scatter()`. It does the opposite of `boost::mpi::gather()`. While `boost::mpi::gather()` gathers data from multiple processes in one process, `boost::mpi::scatter()` scatters data from one process across multiple processes.
Example 47.11 scatters the data in **v** from the process with rank 0 across all processes, including itself. The process with rank 0 receives the string "Hello, world!" in **s**, the process with rank 1 receives "Hello, moon!" in **s**, and the process with rank 2 receives "Hello, sun!" in **s**.

Example 47.12 Sending data to all processes with `broadcast()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  boost::mpi::broadcast(world, s, 0);
  std::cout << s << '\n';
}
```

`boost::mpi::broadcast()` sends data from a process to all processes. The difference between this function and `boost::mpi::scatter()` is that the same data is sent to all processes. In Example 47.12, all processes receive the string “Hello, world!” in **`s`** and write `Hello, world!` to the standard output stream.

Example 47.13 Gathering and analyzing data with `reduce()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

std::string min(const std::string &lhs, const std::string &rhs)
{
  return lhs.size() < rhs.size() ? lhs : rhs;
}

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  else if (world.rank() == 1)
    s = "Hello, moon!";
  else if (world.rank() == 2)
    s = "Hello, sun!";
  std::string result;
  boost::mpi::reduce(world, s, result, min, 0);
  if (world.rank() == 0)
    std::cout << result << '\n';
}
```

`boost::mpi::reduce()` gathers data from multiple processes like `boost::mpi::gather()`. However, the data isn’t stored in a vector. `boost::mpi::reduce()` expects a function or function object, which it will use to analyze the data.

If you run Example 47.13 with three processes, the process with rank 0 receives the string “Hello, sun!” in **`result`**. The call to `boost::mpi::reduce()` gathers and analyzes the strings that all of the processes pass to it. They are analyzed using the function `min()`, which is passed as the fourth parameter to `boost::mpi::reduce()`. `min()` compares two strings and returns the shorter one.

If you run Example 47.13 with more than three processes, an empty string is displayed because all processes with a rank greater than 2 will pass an empty string to `boost::mpi::reduce()`. The empty string will be displayed because it is shorter than “Hello, sun!”

Example 47.14 Gathering and analyzing data with `all_reduce()`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

std::string min(const std::string &lhs, const std::string &rhs)
{
  return lhs.size() < rhs.size() ? lhs : rhs;
}

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  std::string s;
  if (world.rank() == 0)
```

```
    s = "Hello, world!";
  else if (world.rank() == 1)
    s = "Hello, moon!";
  else if (world.rank() == 2)
    s = "Hello, sun!";
  std::string result;
  boost::mpi::all_reduce(world, s, result, min);
  std::cout << world.rank() << ": " << result << '\n';
}
```

Example 47.14 uses the function `boost::mpi::all_reduce()`, which gathers and analyzes data like `boost::mpi::reduce()`. The difference between the two functions is that `boost::mpi::all_reduce()` sends the result of the analysis to all processes while `boost::mpi::reduce()` makes the result only available to the process whose rank is passed as the last parameter. Thus, no rank is passed to `boost::mpi::all_reduce()`. If you run Example 47.14 with three processes, every process writes `Hello, sun!` to the standard output stream.

47.5 Communicators

All of the examples in this chapter use only one communicator, which links all processes. However, it is possible to create more communicators to link subsets of processes. This is especially useful for collective operations that don't need to be executed by all processes.

Example 47.15 Working with multiple communicators

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  boost::mpi::communicator local = world.split(world.rank() < 2 ? 99 : 100);
  std::string s;
  if (world.rank() == 0)
    s = "Hello, world!";
  boost::mpi::broadcast(local, s, 0);
  std::cout << world.rank() << ": " << s << '\n';
}
```

Example 47.15 uses the function `boost::mpi::broadcast()`. This function sends the string “Hello, world!” from the process with rank 0 to all processes that are linked to the communicator **`local`**. The process with rank 0 has to be linked to that communicator, too.

The communicator **`local`** is created by a call to `split()`. `split()` is a member function called on the global communicator **`world`**. `split()` expects an integer to link processes together. All processes that pass the same integer to `split()` are linked to the same communicator. The value of the integer passed to `split()` doesn't matter. All that's important is that all processes that should be linked by a particular communicator pass the same value.

In Example 47.15, the two processes with the ranks 0 and 1 pass 99 to `split()`. If the program is started with more than two processes, the additional processes pass 100. This means the first two processes have one local communicator and all the other process have another local communicator. Every process is linked to the communicator that is returned by `split()`. Whether there are other processes linked to the same communicator depends on whether other processes passed the same integer to `split()`.

Please note that ranks are always relative to a communicator. The lowest rank is always 0. In Example 47.15, the process with the rank 0 relative to the global communicator has also the rank 0 relative to its local communicator. The process with the rank 2 relative to the global communicator has the rank 0 relative to its local communicator.

If you start Example 47.15 with two or more processes, `Hello, world!` will be displayed twice – once each by the processes with the ranks 0 and 1 relative to the global communicator. Because **`s`** is set to “Hello, world!” only in the process with the global rank 0, this string is sent through the communicator only to those processes that are linked to the same communicator. This is just the process with global rank 1, which is the only other process that passed 99 to `split()`.

Example 47.16 Grouping processes with `group`

```
#include <boost/mpi.hpp>
#include <boost/serialization/string.hpp>
#include <boost/range/irange.hpp>
#include <boost/optional.hpp>
#include <string>
#include <iostream>

int main(int argc, char *argv[])
{
  boost::mpi::environment env{argc, argv};
  boost::mpi::communicator world;
  boost::mpi::group local = world.group();
  boost::integer_range<int> r = boost::irange(0, 1);
  boost::mpi::group subgroup = local.exclude(r.begin(), r.end());
  boost::mpi::communicator others{world, subgroup};
  std::string s;
  boost::optional<int> rank = subgroup.rank();
  if (rank)
  {
    if (rank == 0)
      s = "Hello, world!";
    boost::mpi::broadcast(others, s, 0);
  }
  std::cout << world.rank() << ": " << s << '\n';
}
```

MPI supports grouping processes. This is done with the help of the class `boost::mpi::group`. If you call the member function `group()` on a communicator, all processes linked to the communicator are returned in an object of type `boost::mpi::group`. You can't use this object for communication. It can only be used to form a new group of processes from which a communicator can then be created.

`boost::mpi::group` provides member functions like `include()` and `exclude()`. You pass iterators to include or exclude processes. `include()` and `exclude()` return a new group of type `boost::mpi::group`. Example 47.16 passes two iterators to `exclude()` that refer to an object of type `boost::integer_range`. This object represents a range of integers. It is created with the help of the function `boost::irange()`, which expects a lower and upper bound. The upper bound is an integer that doesn't belong to the range. In this example, this means that **`r`** only includes the integer 0.

The call to `exclude()` results in **`subgroup`** being created, which contains all processes except the one with the rank 0. This group is then used to create a new communicator **`others`**. This is done by passing the global communicator **`world`** and **`subgroup`** to the constructor of `boost::mpi::communicator`.

Please note that **`others`** is a communicator that is empty in the process with rank 0. The process with rank 0 isn't linked to this communicator, but the variable **`others`** still exists in this process. You must be careful not to use **`others`** in this process. Example 47.16 prevents this by calling `rank()` on **`subgroup`**. The member function returns an empty object of type `boost::optional` in processes that don't belong to the group. Other processes receive their rank relative to the group.

If `rank()` returns a rank and no empty object of type `boost::optional`, `boost::mpi::broadcast()` is called. The process with rank 0 sends the string "Hello, world!" to all processes linked to the communicator **`others`**. Please note that the rank is relative to that communicator. The process with the rank 0 relative to **`others`** has the rank 1 relative to the global communicator **`world`**.

If you run Example 47.16 with more than two processes, all processes with a global rank greater than 0 will display `Hello, world!`.

Part XI

Generic Programming

The following libraries support generic programming. The libraries can be used without detailed knowledge of template meta programming.

- Boost.TypeTraits provides functions to check properties of types.
- Boost.EnableIf can be used together with Boost.TypeTraits to, for example, overload functions based on their return types.
- Boost.Fusion makes it possible to create heterogeneous containers – containers whose elements can have different types.

Chapter 48

Boost.TypeTraits

Types have different properties that generic programming takes advantage of. The Boost.TypeTraits library provides the tools needed to determine a type's properties and change them.
Since C++11, some functions provided by Boost.TypeTraits can be found in the standard library. You can access those functions through the header file `type_traits`. However, Boost.TypeTraits provides additional functions.

Example 48.1 Determining type categories

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_integral<int>::value << '\n';
  std::cout << is_floating_point<int>::value << '\n';
  std::cout << is_arithmetic<int>::value << '\n';
  std::cout << is_reference<int>::value << '\n';
}
```

Example 48.1 calls several functions to determine type categories. `boost::is_integral` checks whether a type is integral – whether it can store integers. `boost::is_floating_point` checks whether a type stores floating point numbers. `boost::is_arithmetic` checks whether a type supports arithmetic operators. And `boost::is_reference` can be used to determine whether a type is a reference.
`boost::is_integral` and `boost::is_floating_point` are mutually exclusive. A type either stores an integer or a floating point number. However, `boost::is_arithmetic` and `boost::is_reference` can apply to multiple categories. For example, both integer and floating point types support arithmetic operations.
All functions from Boost.TypeTraits provide a result in **`value`** that is either `true` or `false`. Example 48.1 outputs `true` for `is_integral<int>` and `is_arithmetic<int>` and outputs `false` for `is_floating_point<int>` and `is_reference<int>`. Because all of these functions are templates, nothing is processed at run time. The example behaves at run time as though the values `true` and `false` were directly used in the code.
In Example 48.1, the result of the various functions is a value of type bool, which can be written directly to standard output. If the result is to be processed by a function template, it should be forwarded as a type, not as a bool value.

Example 48.2 `boost::true_type` and `boost::false_type`

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_same<is_integral<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_floating_point<int>::type, false_type>::value <<
```

```
    '\n';
  std::cout << is_same<is_arithmetic<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_reference<int>::type, false_type>::value << '\n';
}
```

Besides **`value`**, functions from Boost.TypeTraits also provide the result in type. While **`value`** is a bool value, type is a type. Just like **`value`**, which can only be set to `true` or `false`, type can only be set to one of two types: `boost::true_type` or `boost::false_type`. type lets you pass the result of a function as a type to another function.
Example 48.2 uses another function from Boost.TypeTraits called `boost::is_same`. This function expects two types as parameters and checks whether they are the same. To pass the results of `boost::is_integral`, `boost::is_floating_point`, `boost::is_arithmetic`, and `boost::is_reference` to `boost::is_same`, type must be accessed. type is then compared with `boost::true_type` or `boost::false_type`. The results from `boost::is_same` are then read through **`value`** again. Because this is a bool value, it can be written to standard output.

Example 48.3 Checking type properties with Boost.TypeTraits

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << has_plus<int>::value << '\n';
  std::cout << has_pre_increment<int>::value << '\n';
  std::cout << has_trivial_copy<int>::value << '\n';
  std::cout << has_virtual_destructor<int>::value << '\n';
}
```

Example 48.3 introduces functions that check properties of types. `boost::has_plus` checks whether a type supports the operator `operator+` and whether two objects of the same type can be concatenated. `boost::has_pre_increment` checks whether a type supports the pre-increment operator `operator++`. `boost::has_trivial_copy` checks whether a type has a trivial copy constructor. And `boost::has_virtual_destructor` checks whether a type has a virtual destructor.
Example 48.3 displays `true` three times and `false` once.

Example 48.4 Changing type properties with Boost.TypeTraits

```
#include <boost/type_traits.hpp>
#include <iostream>

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_const<add_const<int>::type>::value << '\n';
  std::cout << is_same<remove_pointer<int*>::type, int>::value << '\n';
  std::cout << is_same<make_unsigned<int>::type, unsigned int>::value <<
    '\n';
  std::cout << is_same<add_rvalue_reference<int>::type, int&&>::value <<
    '\n';
}
```

Example 48.4 illustrates how type properties can be changed. `boost::add_const` adds `const` to a type. If the type is already constant, nothing changes. The code compiles without problems, and the type remains constant. `boost::remove_pointer` removes the asterisk from a pointer type and returns the type the pointer refers to. `boost::make_unsigned` turns a type with a sign into a type without a sign. And `boost::add_rvalue_reference` transforms a type into a rvalue reference.
Example 48.4 writes `true` four times to standard output.

Chapter 49

Boost.EnableIf

Boost.EnableIf makes it possible to disable overloaded function templates or specialized class templates. Disabling means that the compiler ignores the respective templates. This helps to prevent ambiguous scenarios in which the compiler doesn't know which overloaded function template to use. It also makes it easier to define templates that can be used not just for a certain type but for a group of types.
Since C++11, Boost.EnableIf has been part of the standard library. You can call the functions introduced in this chapter without using a Boost library; just include the header file `type_traits`.
Example 49.1 defines the function template `create()`, which returns an object of the type passed as a template parameter. The object is initialized in `create()`, which accepts no parameters. The signatures of the two `create()` functions don't differ. In that respect `create()` isn't an overloaded function. The compiler would report an error if Boost.EnableIf didn't enable one function and disable the other.
Boost.EnableIf provides the class `boost::enable_if`, which is a template that expects two parameters. The first parameter is a condition. The second parameter is the type of the `boost::enable_if` expression if the condition is true. The trick is that this type doesn't exist if the condition is false, in which case the `boost::enable_if` expression is invalid C++ code. However, when it comes to templates, the compiler doesn't complain about invalid code. Instead it ignores the template and searches for another one that might fit. This concept is known as SFINAE which stands for "Substitution Failure Is Not An Error."
In Example 49.1 both conditions in the `boost::enable_if` expressions use the class `std::is_same`. This class is defined in the C++11 standard library and allows you to compare two types. Because such a comparison is either true or false, it's sufficient to use `std::is_same` to define a condition.

Example 49.1 Overloading functions with `boost::enable_if` on their return value

```
#include <boost/utility/enable_if.hpp>
#include <type_traits>
#include <string>
#include <iostream>

template <typename T>
typename boost::enable_if<std::is_same<T, int>, T>::type create()
{
  return 1;
}

template <typename T>
typename boost::enable_if<std::is_same<T, std::string>, T>::type create()
{
  return "Boost";
}

int main()
{
  std::cout << create<std::string>() << '\n';
}
```

If a condition is true, the respective `create()` function should return an object of the type that was passed to `create()` as a template parameter. That's why `T` is passed as a second parameter to `boost::enable_if`. The entire `boost::enable_if` expression is replaced by `T` if the condition is true. In Example 49.1 the compiler

sees either a function that returns an int or a function that returns a `std::string`. If `create()` is called with any other type than int or `std::string`, the compiler will report an error.
Example 49.1 displays `Boost`.
Example 49.2 uses `boost::enable_if` to specialize a function for a group of types. The function is called `print()` and expects one parameter. It can be overloaded, although overloading requires you to use a concrete type. To do the same for a group of types like short, int or long, you can define an appropriate condition using `boost::enable_if`. Example 49.2 uses `std::is_integral` to do so. The second `print()` function is overloaded with `std::is_floating_point` for all floating point numbers.

Example 49.2 Specializing functions for a group of types with `boost::enable_if`

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

template <typename T>
void print(typename boost::enable_if<std::is_integral<T>, T>::type i)
{
  std::cout << "Integral: " << i << '\n';
}

template <typename T>
void print(typename boost::enable_if<std::is_floating_point<T>, T>::type f)
{
  std::cout << "Floating point: " << f << '\n';
}

int main()
{
  print<short>(1);
  print<long>(2);
  print<double>(3.14);
}
```

Chapter 50

Boost.Fusion

The standard library provides numerous containers that have one thing in common: They are homogeneous. That is, containers from the standard library can only store elements of one type. A vector of the type std::vector<int> can only store int values, and a vector of type std::vector<std::string> can only store strings.

Boost.Fusion makes it possible to create heterogeneous containers. For example, you can create a vector whose first element is an int and whose second element is a string. In addition, Boost.Fusion provides algorithms to process heterogeneous containers. You can think of Boost.Fusion as the standard library for heterogeneous containers.

Strictly speaking, since C++11, the standard library has provided a heterogeneous container, `std::tuple`. You can use different types for the values stored in a tuple. `boost:fusion::tuple` in Boost.Fusion is a similar type. While the standard library doesn't have much more to offer, tuples are just the starting place for Boost.Fusion.

Example 50.1 defines a tuple consisting of an int, a `std::string`, a bool, and a double. The tuple is based on `boost:fusion::tuple`. In Example 50.1, the tuple is then instantiated, initialized, and the various elements are retrieved with `boost::fusion::get()` and written to standard output. The function `boost::fusion::get()` is similar to `std::get()`, which accesses elements in `std::tuple`.

Fusion tuples don't differ from tuples from the standard library. Thus it's no surprise that Boost.Fusion provides a function `boost::fusion::make_tuple()`, which works like `std::make_tuple()`. However, Boost.Fusion provides additional functions that go beyond what is offered in the standard library.

Example 50.1 Processing Fusion tuples

```
#include <boost/fusion/tuple.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  std::cout << get<0>(t) << '\n';
  std::cout << get<1>(t) << '\n';
  std::cout << std::boolalpha << get<2>(t) << '\n';
  std::cout << get<3>(t) << '\n';
}
```

Example 50.2 introduces the algorithm `boost::fusion::for_each()`, which iterates over a Fusion container. The function is used here to write the values in the tuple **`t`** to standard output.

Example 50.2 Iterating over a tuple with `boost::fusion::for_each()`

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/algorithm.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;
```

```
struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  for_each(t, print{});
}
```

`boost::fusion::for_each()` is designed to work like `std::for_each()`. While `std::for_each()` only iterates over homogeneous containers, `boost::fusion::for_each()` works with heterogeneous containers. You pass a container, not an iterator, to `boost::fusion::for_each()`. If you don't want to iterate over all elements in a container, you can use a *view*.

Example 50.3 Filtering a Fusion container with `boost::fusion::filter_view`

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/view.hpp>
#include <boost/fusion/algorithm.hpp>
#include <boost/type_traits.hpp>
#include <boost/mpl/arg.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  filter_view<tuple_type, boost::is_integral<boost::mpl::arg<1>>> v{t};
  for_each(v, print{});
}
```

Boost.Fusion provides views, which act like containers but don't store data. With views, data in a container can be accessed differently. Views are similar to adaptors from Boost.Range. However, while adaptors from Boost.Range can be applied to only one container, views from Boost.Fusion can span data from multiple containers.

Example 50.3 uses the class `boost::fusion::filter_view` to filter the tuple **`t`**. The filter directs `boost::fusion::for_each()` to only write elements based on an integral type.

`boost::fusion::filter_view` expects as a first template parameter the type of the container to filter. The second template parameter must be a predicate to filter elements. The predicate must filter the elements based on their type.

The library is called Boost.Fusion because it combines two worlds: C++ programs process values at run time and types at compile time. For developers, values at run time are usually more important. Most tools from the standard library process values at run time. To process types at compile time, template meta programming is used. While values are processed depending on other values at run time, types are processed depending on other types

at compile time. Boost.Fusion lets you process values depending on types.
The second template parameter passed to `boost::fusion::filter_view` is a predicate, which will be applied to every type in the tuple. The predicate expects a type as a parameter and returns `true` if the type should be part of the view. If `false` is returned, the type is filtered out.
Example 50.3 uses the class `boost::is_integral` from Boost.TypeTraits. `boost::is_integral` is a template that checks whether a type is integral. Because the template parameter has to be passed to `boost::fusion::filter_view`, a placeholder from Boost.MPL, `boost::mpl::arg<1>`, is used to create a lambda function. `boost::mpl::arg<1>` is similar to **`boost::phoenix::place_holders::arg1`** from Boost.Phoenix.
In Example 50.3, the view **`v`** will contain only the int and bool elements from the tuple, and therefore, the example will write `10` and `true` to standard output.

Example 50.4 Accessing elements in Fusion containers with iterators

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/iterator.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  auto it = begin(t);
  std::cout << *it << '\n';
  auto it2 = advance<boost::mpl::int_<2>>(it);
  std::cout << std::boolalpha << *it2 << '\n';
}
```

After seeing `boost::fusion::tuple` and `boost::fusion::for_each()`, it shouldn't come as a surprise to find iterators in Example 50.4. Boost.Fusion provides several free-standing functions, such as `boost::fusion::begin()` and `boost::fusion::advance()`, that work like the identically named functions from the standard library.
The number of steps the iterator is to be incremented is passed to `boost::fusion::advance()` as a template parameter. The example again uses `boost::mpl::int_` from Boost.MPL.
`boost::fusion::advance()` returns an iterator of a different type from the one that was passed to the function. That's why Example 50.4 uses a second iterator **`it2`**. You can't assign the return value from `boost::fusion::advance()` to the first iterator **`it`**. Example 50.4 writes `10` and `true` to standard output.
In addition to the functions introduced in the example, Boost.Fusion provides other functions that work with iterators. These include the following: `boost::fusion::end()`, `boost::fusion::distance()`, `boost::fusion::next()` and `boost::fusion::prior()`.
So far we've only seen one heterogeneous container, `boost::fusion::tuple`. Example 50.5 introduces another container, `boost::fusion::vector`.
`boost::fusion::vector` is a vector: elements are accessed with indexes. Access is not implemented using the operator `operator[]`. Instead, it's implemented using `boost::fusion::at()`, a free-standing function. The index is passed as a template parameter wrapped with `boost::mpl::int_`.

Example 50.5 A heterogeneous vector with `boost::fusion::vector`

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef vector<int, std::string, bool, double> vector_type;
  vector_type v{10, "Boost", true, 3.14};
  std::cout << at<boost::mpl::int_<0>>(v) << '\n';
  auto v2 = push_back(v, 'X');
  std::cout << size(v) << '\n';
  std::cout << size(v2) << '\n';
  std::cout << back(v2) << '\n';
}
```

This example adds a new element of type char to the vector. This is done with the free-standing function `boost::fusion::push_back()`. Two parameters are passed to `boost::fusion::push_back()`: the vector to add the element to and the value to add.
`boost::fusion::push_back()` returns a new vector. The vector **`v`** isn't changed. The new vector is a copy of the original vector with the added element.
This example gets the number of elements in the vectors **`v`** and **`v2`** with `boost::fusion::size()` and writes both values to standard output. The program displays `4` and `5`. It then calls `boost::fusion::back()` to get and write the last element in **`v2`** to standard output, in this case the value is `X`.
If you look more closely at Example 50.5, you will notice that there is no difference between `boost::fusion::tuple` and `boost::fusion::vector`; they are the same. Thus, Example 50.5 will also work with `boost::fusion::tuple`.
Boost.Fusion provides additional heterogeneous containers, including: `boost::fusion::deque`, `boost::fusion::list` and `boost::fusion::set`. Example 50.6 introduces `boost::fusion::map`, a container for key/value pairs.
Example 50.6 creates a heterogeneous map with `boost::fusion::map()`. The map's type is `boost::fusion::map`, which isn't written out in the example thanks to the keyword `auto`.

Example 50.6 A heterogeneous map with `boost::fusion::map`

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/fusion/algorithm.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  auto m = make_map<int, std::string, bool, double>("Boost", 10, 3.14, true);
  if (has_key<std::string>(m))
    std::cout << at_key<std::string>(m) << '\n';
  auto m2 = erase_key<std::string>(m);
  auto m3 = push_back(m2, make_pair<float>('X'));
  std::cout << std::boolalpha << has_key<std::string>(m3) << '\n';
}
```

A map of type `boost::fusion::map` stores key/value pairs like `std::map` does. However, the keys in the Fusion map are types. A key/value pair consists of a type and a value mapped to that type. The value may be a different type than the key. In Example 50.6, the string "Boost" is mapped to the key int.
After the map has been created, `boost::fusion::has_key()` is called to check whether a key `std::string` exists. Then, `boost::fusion::at_key()` is called to get the value mapped to that key. Because the number 10 is mapped to `std::string`, it is written to standard output.
The key/value pair is then erased with `boost::fusion::erase_key()`. This doesn't change the map **`m`**. `boost::fusion::erase_key()` returns a new map which is missing the erased key/value pair.
The call to `boost::fusion::push_back()` adds a new key/value pair to the map. The key is float and the value is "X". `boost::fusion::make_pair()` is called to create the new key/value pair. This function is similar to `std::make_pair()`.
Finally, `boost::fusion::has_key()` is called again to check whether the map has a key `std::string`. Because it was erased, `false` is returned.
Please note that you don't need to call `boost::fusion::has_key()` to check whether a key exists before you call `boost::fusion::at_key()`. If a key is passed to `boost::fusion::at_key()` that doesn't exist in the map, you get a compiler error.
Boost.Fusion provides several macros that let you use structures as Fusion containers. This is possible because structures can act as heterogeneous containers. Example 50.7 defines a structure which can be used as a Fusion

container thanks to the macro `BOOST_FUSION_ADAPT_STRUCT`. This makes it possible to use the structure with functions like `boost::fusion::at()` or `boost::fusion::back()`.

Example 50.7 Fusion adaptors for structures

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <iostream>

struct strct
{
  int i;
  double d;
};

BOOST_FUSION_ADAPT_STRUCT(strct,
  (int, i)
  (double, d)
)

using namespace boost::fusion;

int main()
{
  strct s = {10, 3.14};
  std::cout << at<boost::mpl::int_<0>>(s) << '\n';
  std::cout << back(s) << '\n';
}
```

Boost.Fusion supports structures such as `std::pair` and `boost::tuple` without having to use macros. You just need to include the header file `boost/fusion/adapted.hpp` (see Example 50.8).

Example 50.8 Fusion support for `std::pair`

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <utility>
#include <iostream>

using namespace boost::fusion;

int main()
{
  auto p = std::make_pair(10, 3.14);
  std::cout << at<boost::mpl::int_<0>>(p) << '\n';
  std::cout << back(p) << '\n';
}
```

Part XII

Language Extensions

The following libraries extend the programming language C++.

- Boost.Coroutine makes it possible to use coroutines in C++ – something other programming languages usually support with the keyword `yield`.
- Boost.Foreach provides a range-based `for` loop, which was added to the language with C++11.
- Boost.Parameter lets you pass parameters as name/value pairs and in any order – as is allowed in Python, for example.
- Boost.Conversion provides two cast operators that replace `dynamic_cast` and allow you to differentiate between a downcast and a cross cast.

Chapter 51

Boost.Coroutine

With Boost.Coroutine it is possible to use *coroutines* in C++. Coroutines are a feature of other programming languages, which often use the keyword `yield` for coroutines. In these programming languages, `yield` can be used like `return`. However, when `yield` is used, the function remembers the location, and if the function is called again, execution continues from that location.

C++ doesn't define a keyword `yield`. However, with Boost.Coroutine it is possible to return from functions and continue later from the same location. The Boost.Asio library also uses Boost.Coroutine and benefits from coroutines.

There are two versions of Boost.Coroutine. This chapter introduces the second version, which is the current version. This version has been available since Boost 1.55.0 and replaces the first one.

Example 51.1 defines a function, `cooperative()`, which is called from `main()` as a coroutine. `cooperative()` returns to `main()` early and is called a second time. On the second call, it continues from where it left off.

To use `cooperative()` as a coroutine, the types pull_type and push_type are used. These types are provided by `boost::coroutines::coroutine`, which is a template that is instantiated with void in Example 51.1.

To use coroutines, you need pull_type and push_type. One of these types will be used to create an object that will be initialized with the function you want to use as a coroutine. The other type will be the first parameter of the coroutine function.

Example 51.1 creates an object named **`source`** of type pull_type in `main()`. `cooperative()` is passed to the constructor. push_type is used as the sole parameter in the signature of `cooperative()`.

Example 51.1 Using coroutines

```
#include <boost/coroutine/all.hpp>
#include <iostream>

using namespace boost::coroutines;

void cooperative(coroutine<void>::push_type &sink)
{
  std::cout << "Hello";
  sink();
  std::cout << "world";
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  std::cout << ", ";
  source();
  std::cout << "!\n";
}
```

When **`source`** is created, the function `cooperative()`, which is passed to the constructor, is immediately called as a coroutine. This happens because **`source`** is based on pull_type. If **`source`** was based on push_type, the constructor wouldn't call `cooperative()` as a coroutine.

`cooperative()` writes `Hello` to standard output. Afterwards, the function accesses **`sink`** as if it were a function. This is possible because push_type overloads `operator()`. While **`source`** in `main()` represents the corou-

tine cooperative(), **sink** in cooperative() represents the function main(). Calling **sink** makes cooperative() return, and main() continues from where cooperative() was called and writes a comma to standard output.
Then, main() calls **source** as if it were a function. Again, this is possible because of the overloaded operator(). This time, cooperative() continues from the point where it left off and writes world to standard output. Because there is no other code in cooperative(), the coroutine ends. It returns to main(), which writes an exclamation mark to standard output.
The result is that Example 51.1 displays Hello, world!
You can think of coroutines as cooperative threads. To a certain extent, the functions main() and cooperative() run concurrently. Code is executed in turns in main() and cooperative(). Instructions inside each function are executed sequentially. Thanks to coroutines, a function doesn't need to return before another function can be executed.

Example 51.2 Returning a value from a coroutine

```
#include <boost/coroutine/all.hpp>
#include <functional>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<int>::push_type &sink, int i)
{
  int j = i;
  sink(++j);
  sink(++j);
  std::cout << "end\n";
}

int main()
{
  using std::placeholders::_1;
  coroutine<int>::pull_type source{std::bind(cooperative, _1, 0)};
  std::cout << source.get() << '\n';
  source();
  std::cout << source.get() << '\n';
  source();
}
```

Example 51.2 is similar to the previous example. This time the template boost::coroutines::coroutine is instantiated with int. This makes it possible to return an int from the coroutine to the caller.
The direction the int value is passed depends on where pull_type and push_type are used. The example uses pull_type to instantiate an object in main(). cooperative() has access to an object of type push_type. push_type sends a value, and pull_type receives a value; thus, the direction of the data transfer is set.
cooperative() calls **sink**, with a parameter of type int. This parameter is required because the coroutine was instantiated with the data type int. The value passed to **sink** is received from **source** in main() by using the member function get(), which is provided by pull_type.
Example 51.2 also illustrates how a function with multiple parameters can be used as a coroutine. cooperative() has an additional parameter of type int, which can't be passed directly to the constructor of pull_type. The example uses std::bind() to link the function with pull_type.
The example writes 1 and 2 followed by end to standard output.

Example 51.3 Passing two values to a coroutine

```
#include <boost/coroutine/all.hpp>
#include <tuple>
#include <string>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<std::tuple<int, std::string>>::pull_type &source)
{
  auto args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
  source();
  args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
}

int main()
{
  coroutine<std::tuple<int, std::string>>::push_type sink{cooperative};
  sink(std::make_tuple(0, "aaa"));
  sink(std::make_tuple(1, "bbb"));
  std::cout << "end\n";
}
```

Example 51.3 uses push_type in `main()` and pull_type in `cooperative()`, which means data is transferred from the caller to the coroutine.
This example illustrates how multiple values can be passed. Boost.Coroutine doesn't support passing multiple values, so a tuple must be used. You need to pack multiple values into a tuple or another structure.
Example 51.3 displays `0 aaa`, `1 bbb`, and `end`.
A coroutine returns immediately when an exception is thrown. The exception is transported to the caller of the coroutine where it can be caught. Thus, exceptions are no different than with regular function calls.
Example 51.4 shows how this works. This example will write the string `error` to standard output.

Example 51.4 Coroutines and exceptions

```
#include <boost/coroutine/all.hpp>
#include <stdexcept>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<void>::push_type &sink)
{
  sink();
  throw std::runtime_error("error");
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  try
  {
    source();
  }
  catch (const std::runtime_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Chapter 52

Boost.Foreach

Boost.Foreach provides a macro that simulates the range-based `for` loop from C++11. You can use the macro `BOOST_FOREACH`, defined in `boost/foreach.hpp`, to iterate over a sequence without using iterators. If your development environment supports C++11, you can ignore Boost.Foreach.

Example 52.1 Using `BOOST_FOREACH` and `BOOST_REVERSE_FOREACH`

```
#include <boost/foreach.hpp>
#include <array>
#include <iostream>

int main()
{
  std::array<int, 4> a{{0, 1, 2, 3}};

  BOOST_FOREACH(int &i, a)
    i *= i;

  BOOST_REVERSE_FOREACH(int i, a)
  {
    std::cout << i << '\n';
  }
}
```

`BOOST_FOREACH` expects two parameters. The first parameter is a variable or reference, and the second is a sequence. The type of the first parameter needs to match the type of the elements in the sequence.
Anything offering iterators, such as containers from the standard library, classifies as a sequence. Boost.Foreach uses Boost.Range instead of directly accessing the member functions `begin()` and `end()`. However, because Boost.Range is based on iterators, anything providing iterators is compatible with `BOOST_FOREACH`.
Example 52.1 iterates over an array of type `std::array` with `BOOST_FOREACH`. The first parameter passed is a reference so that you can both read and modify the elements in the array. In Example 52.1, the first loop multiplies each number by itself.
The second loop uses the macro `BOOST_REVERSE_FOREACH`, which works the same as `BOOST_FOREACH`, but iterates backwards over a sequence. The loop writes the numbers 9, 4, 1, and 0 in that order to the standard output stream.
As usual, curly brackets can be omitted if the block only consists of one statement.
Please note that you should not use operations that invalidate the iterator inside the loop. For example, elements should not be added or removed while iterating over a vector. `BOOST_FOREACH` and `BOOST_REVERSE_FOREACH` require iterators to be valid throughout the whole iteration.

Chapter 53

Boost.Parameter

Boost.Parameter makes it possible to pass parameters as key/value pairs. In addition to supporting function parameters, the library also supports template parameters. Boost.Parameter is especially useful if you are using long parameter lists, and the order and meaning of parameters is difficult to remember. Key/value pairs make it possible to pass parameters in any order. Because every value is passed with a key, the meaning of the various values is also clearer.

Example 53.1 defines a function `complicated()`, which expects five parameters. The parameters may be passed in any order. Boost.Parameter provides the macro `BOOST_PARAMETER_FUNCTION` to define such a function.

Before `BOOST_PARAMETER_FUNCTION` can be used, the parameters for the key/value pairs must be defined. This is done with the macro `BOOST_PARAMETER_NAME`, which is just passed a parameter name. The example uses `BOOST_PARAMETER_NAME` five times to define the parameter names **a**, **b**, **c**, **d**, and **e**.

Please note that the parameter names are automatically defined in the namespace `tag`. This should avoid clashes with identically named definitions in a program.

After the parameter names have been defined, `BOOST_PARAMETER_FUNCTION` is used to define the function `complicated()`. The first parameter passed to `BOOST_PARAMETER_FUNCTION` is the type of the return value. This is void in the example. Please note that the type must be wrapped in parentheses – the first parameter is `(void)`. The second parameter is the name of the function being defined. The third parameter is the namespace containing the parameter names. In the fourth parameter, the parameter names are accessed to further specify them.

Example 53.1 Function parameters as key/value pairs

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char))
    (c, (double))
    (d, (std::string))
    (e, *)
  )
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_c = 3.14, _a = 1, _d = "Boost", _b = 'B', _e = true);
}
```

In Example 53.1 the fourth parameter starts with `required`, which is a keyword that makes the parameters that follow mandatory. `required` is followed by one or more pairs consisting of a parameter name and a type. It is important to wrap the type in parentheses.
Various types are used for the parameters **a**, **b**, **c**, and **d**. For example, **a** can be used to pass an int value to `complicated()`. No type is given for **e**. Instead, an asterisk is used, which means that the value passed may have any type. **e** is a template parameter.
After the various parameters have been passed to `BOOST_PARAMETER_FUNCTION`, the function body is defined. This is done, as usual, between a pair of curly brackets. Parameters can be accessed in the function body. They can be used like variables, with the types assigned within `BOOST_PARAMETER_FUNCTION`. Example 53.1 writes the parameters to standard output.
`complicated()` is called from `main()`. The parameters are passed to `complicated()` in an arbitrary order. Parameter names start with an underscore. Boost.Parameter uses the underscore to avoid name clashes with other variables.

Note

To pass function parameters as key/value pairs in C++, you can also use the named parameter idiom, which doesn't require a library like Boost.Parameter.

Example 53.2 Optional function parameters

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char)))
  (optional
    (c, (double), 3.14)
    (d, (std::string), "Boost")
    (e, *, true))
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_b = 'B', _a = 1);
}
```

`BOOST_PARAMETER_FUNCTION` also supports defining optional parameters.
In Example 53.2 the parameters **c**, **d**, and **e** are optional. These parameters are defined in `BOOST_PARAMETER_FUNCTION` using the `optional` keyword.
Optional parameters are defined like required parameters: a parameter name is given followed by a type. As usual, the type is wrapped in parentheses. However, optional parameters need to have a default value.
With the call to `complicated()`, only the parameters **a** and **b** are passed. These are the only required parameters. As the parameters **c**, **d**, and **e** aren't used, they are set to default values.
Boost.Parameter provides macros in addition to `BOOST_PARAMETER_FUNCTION`. For example, you can use `BOOST_PARAMETER_MEMBER_FUNCTION` to define a member function, and `BOOST_PARAMETER_CONST_MEMBER_FUNCTION` to define a constant member function.
You can define functions with Boost.Parameter that try to assign values to parameters automatically. In that case, you don't need to pass key/value pairs – it is sufficient to pass values only. If the types of all values are different, Boost.Parameter can detect which value belongs to which parameter. This might require you to have a deeper knowledge of template meta programming.
Example 53.3 uses Boost.Parameter to pass template parameters as key/value pairs. As with functions, it is possible to pass the template parameters in any order.
The example defines a class `complicated`, which expects three template parameters. Because the order of the parameters doesn't matter, they are called `A`, `B`, and `C`. `A`, `B`, and `C` aren't the names of the parameters that will be used when the class template is accessed. As with functions, the parameter names are defined using a macro. For template parameters, `BOOST_PARAMETER_TEMPLATE_KEYWORD` is used. Example 53.3 defines three parameter names integral_type, floating_point_type, and any_type.
After the parameter names have been defined, you must specify the types that may be passed. For example, the parameter integral_type can be used to pass types such as `int` or `long`, but not a type like `std::string`.
`boost::parameter::parameters` is used to create a signature that refers to the parameter names and defines which types may be passed with each of them.
`boost::parameter::parameters` is a tuple that describes parameters. Required parameters are marked with `boost::parameter::required`.
`boost::parameter::required` requires two parameters. The first is the name of the parameter defined with `BOOST_PARAMETER_TEMPLATE_KEYWORD`. The second identifies the type the parameter may be set to. For example, integral_type may be set to an integral type. This requirement is expressed with `std::is_integral<_>`. `std::is_integral<_>` is a lambda function based on Boost.MPL. `boost::mpl::placeholders::_` is a placeholder provided by this library. If the type to which integral_type is set is passed to `std::is_integral` instead of `boost::mpl::placeholders::_`, and the result is true, a valid type is used. The requirements for the other parameters floating_point_type and any_type are defined similarly.

Example 53.3 Template parameters as key/value pairs

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  required<tag::floating_point_type, std::is_floating_point<_>>,
  required<tag::any_type, std::is_object<_>>
```

```
> complicated_signature;

template <class A, class B, class C>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<int>,
    any_type<bool>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

After the signature has been created and defined as complicated_signature, it is used by the class `complicated`. First, the signature is bound with `complicated_signature::bind` to the template parameters `A`, `B`, and `C`. The new type, args, represents the connection between the template parameters passed and the requirements that must be met by the template parameters. Next, args is accessed to get the parameter values. This is done with `boost::parameter::value_type`. `boost::parameter::value_type` expects args and a parameter to be passed. The parameter determines the type created. In Example 53.3, the type definition integral_type in the class `complicated` is used to get the type that was passed with the parameter integral_type to `complicated`. `main()` accesses `complicated` to instantiate the class. The parameter integral_type is set to int, floating_point_type to double, and any_type to bool. The order of the parameters passed doesn't matter. The type definitions integral_type, floating_point_type, and any_type are then accessed by `typeid` to get their underlying types. Compiled with Visual C++ 2013, the example writes `int`, `double` and `bool` to standard output.

Example 53.4 Optional template parameters

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  optional<tag::floating_point_type, std::is_floating_point<_>>,
  optional<tag::any_type, std::is_object<_>>
> complicated_signature;

template <class A, class B = void_, class C = void_>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type, float>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type, bool>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<short>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Example 53.4 introduces optional template parameters. The signature uses `boost::parameter::optional` for the optional template parameters. The optional template parameters from `complicated` are set to `boost::parameter::void_`, and `boost::parameter::value_type` is given a default value. This default value is the type an optional parameter will be set to if the type isn't otherwise set.

`complicated` is instantiated in `main()`. This time only the parameters integral_type and floating_point_type are used. any_type is not used. Compiled with Visual C++ 2013, the example writes `short` for integral_type, `double` for floating_point_type, and `bool` for any_type to standard output.

Boost.Parameter can automatically detect template parameters. You can create signatures that allow types to be automatically assigned to parameters. As with function parameters, deeper knowledge in template meta programming is required to do this.

Chapter 54

Boost.Conversion

Boost.Conversion defines the cast operators `boost::polymorphic_cast` and `boost::polymorphic_downcast` in the header file `boost/cast.hpp`. They are designed to handle type casts – usually done with `dynamic_cast` – more precisely.

Example 54.1 Down and cross casts with `dynamic_cast`

```
struct base1 { virtual ~base1() = default; };
struct base2 { virtual ~base2() = default; };
struct derived : public base1, public base2 {};

void downcast(base1 *b1)
{
  derived *d = dynamic_cast<derived*>(b1);
}

void crosscast(base1 *b1)
{
  base2 *b2 = dynamic_cast<base2*>(b1);
}

int main()
{
  derived *d = new derived;
  downcast(d);

  base1 *b1 = new derived;
  crosscast(b1);
}
```

Example 54.1 uses the cast operator `dynamic_cast` twice: In `downcast()`, it transforms a pointer pointing to a base class to one pointing to a derived class. In `crosscast()`, it transforms a pointer pointing to a base class to one pointing to a different base class. The first transformation is a *downcast*, and the second is a *cross cast*. The cast operators from Boost.Conversion let you distinguish a downcast from a cross cast.

Example 54.2 Down and cross casts with `polymorphic_downcast` and `polymorphic_cast`

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

struct base1 { virtual ~base1() = default; };
struct base2 { virtual ~base2() = default; };
struct derived : public base1, public base2 {};

void downcast(base1 *b1)
{
  derived *d = boost::polymorphic_downcast<derived*>(b1);
}

void crosscast(base1 *b1)
{
  base2 *b2 = boost::polymorphic_cast<base2*>(b1);
}

int main()
{
  derived *d = new derived;
  downcast(d);

  base1 *b1 = new derived;
  crosscast(b1);
}
```

`boost::polymorphic_downcast` (see Example 54.2) can only be used for downcasts because it uses `static_cast` to perform the cast. Because `static_cast` does not dynamically check the cast for validity, `boost::polymorphic_downcast` must only be used if the cast is safe. In debug builds, `boost::polymorphic_downcast` uses `dynamic_cast` and `assert()` to make sure the type cast is valid. This test is only performed if the macro `NDEBUG` is not defined, which is usually the case for debug builds.

`boost::polymorphic_cast` is required for cross casts. `boost::polymorphic_cast` uses `dynamic_cast`, which is the only cast operator that can perform a cross cast. It is better to use `boost::polymorphic_cast` instead of `dynamic_cast` because the former throws an exception of type `std::bad_cast` in case of an error, while `dynamic_cast` returns a null pointer if the type cast fails.

Use `boost::polymorphic_downcast` and `boost::polymorphic_cast` only to convert pointers; otherwise, use `dynamic_cast`. Because `boost::polymorphic_downcast` is based on `static_cast`, it cannot convert objects of a base class to objects of a derived class. Also, it does not make sense to use `boost::polymorphic_cast` to convert types other than pointers because `dynamic_cast` will throw an exception of type `std::bad_cast` if a cast fails.

Part XIII

Error Handling

The following libraries support error handling.

- Boost.System provides classes to describe and identify errors. Since C++11, these classes have been part of the standard library.
- Boost.Exception makes it possible to attach data to exceptions after they have been thrown.

Chapter 55

Boost.System

Boost.System is a library that, in essence, defines four classes to identify errors. All four classes were added to the standard library with C++11. If your development environment supports C++11, you don't need to use Boost.System. However, since many Boost libraries use Boost.System, you might encounter Boost.System through those other libraries.

`boost::system::error_code` is the most basic class in Boost.System; it represents operating system-specific errors. Because operating systems typically enumerate errors, `boost::system::error_code` saves an error code in a variable of type int. Example 55.1 illustrates how to use this class.

Example 55.1 Using `boost::system::error_code`

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  std::cout << ec.value() << '\n';
}
```

Example 55.1 defines the function `fail()`, which is used to return an error. In order for the caller to detect whether `fail()` failed, an object of type `boost::system::error_code` is passed by reference. Many functions that are provided by Boost libraries use `boost::system::error_code` like this. For example, Boost.Asio provides the function `boost::asio::ip::host_name()`, to which you can pass an object of type `boost::system::error_code`.

Boost.System defines numerous error codes in the namespace `boost::system::errc`. Example 55.1 assigns the error code `boost::system::errc::not_supported` to **`ec`**. Because **`boost::system::errc::not_supported`** is a number and **`ec`** is an object of type `boost::system::error_code`, the function `boost::system::errc::make_error_code()` is called. This function creates an object of type `boost::system::error_code` with the respective error code.

In `main()`, `value()` is called on **`ec`**. This member function returns the error code stored in the object.

By default, 0 means no error. Every other number refers to an error. Error code values are operating system dependent. Refer to the documentation for your operating system for a description of error codes.

In addition to `value()`, `boost::system::error_code` provides the member function `category()`, which returns an object of type `boost::system::error_category`.

Error codes are simply numeric values. While operating system manufacturers such as Microsoft are able to guarantee the uniqueness of system error codes, keeping error codes unique across all existing applications is virtually impossible for application developers. It would require a central database filled with error codes from all software developers around the world to avoid reusing the same codes for different errors. Because this is impractical, error categories exist.

Example 55.2 Using `boost::system::error_category`

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  std::cout << ec.value() << '\n';
  std::cout << ec.category().name() << '\n';
}
```

Error codes of type `boost::system::error_code` belong to a category that can be retrieved with the member function `category()`. Errors created with `boost::system::errc::make_error_code()` automatically belong to the generic category. This is the category errors belong to if they aren't assigned to another category explicitly.

As shown in Example 55.2, `category()` returns an error's category. This is an object of type `boost::system::error_category`. There are only a few member functions. For example, `name()` retrieves the name of the category. Example 55.2 writes `generic` to standard output.

You can also use the free-standing function `boost::system::generic_category()` to access the generic category.

Boost.System provides a second category. If you call the free-standing function `boost::system::system_category()`, you get a reference to the system category. If you write the category's name to standard output, `system` is displayed.

Errors are uniquely identified by the error code and the error category. Because error codes are only required to be unique within a category, you should create a new category whenever you want to define error codes specific to your program. This makes it possible to use error codes that do not interfere with error codes from other developers.

Example 55.3 Creating error categories

```
#include <boost/system/error_code.hpp>
#include <string>
#include <iostream>

class application_category :
  public boost::system::error_category
{
public:
  const char *name() const noexcept { return "my app"; }
  std::string message(int ev) const { return "error message"; }
};

application_category cat;

int main()
{
  boost::system::error_code ec{129, cat};
  std::cout << ec.value() << '\n';
  std::cout << ec.category().name() << '\n';
}
```

A new error category is defined by creating a class derived from `boost::system::error_category`. This requires you to define various member functions. At a minimum, the member functions `name()` and `message()`

must be supplied because they are defined as pure virtual member functions in `boost::system::error_category`. For additional member functions, the default behavior can be overridden if required.
While `name()` returns the name of the error category, `message()` is used to retrieve the error description for a particular error code. Unlike Example 55.3, the parameter **ev** is usually evaluated to return a description based on the error code.
An object of the type of the newly created error category can be used to initialize an error code. Example 55.3 defines the error code **ec** using the new category `application_category`. Therefore, error code 129 is no longer a generic error; instead, its meaning is defined by the developer of the new error category.

> Note
>
> To compile Example 55.3 with Visual C++ 2013, remove the keyword `noexcept`. This version of the Microsoft compiler doesn't support `noexcept`.

`boost::system::error_code` provides a member function called `default_error_condition()`, that returns an object of type `boost::system::error_condition`. The interface of `boost::system::error_condition` is almost identical to the interface of `boost::system::error_code`. The only difference is the member function `default_error_condition()`, which is only provided by `boost::system::error_code`.

Example 55.4 Using `boost::system::error_condition`

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  boost::system::error_condition ecnd = ec.default_error_condition();
  std::cout << ecnd.value() << '\n';
  std::cout << ecnd.category().name() << '\n';
}
```

`boost::system::error_condition` is used just like `boost::system::error_code`. That's why it's possible, as shown in Example 55.4, to call the member functions `value()` and `category()` for an object of type `boost::system::error_condition`.
While the class `boost::system::error_code` is used for platform-dependent error codes, `boost::system::error_condition` is used to access platform-independent error codes. The member function `default_error_condition()` translates a platform-dependent error code into a platform-independent error code of type `boost::system::error_condition`.
You can use `boost::system::error_condition` to identify errors that are platform independent. Such an error could be, for example, a failed access to a non-existing file. While operating systems may provide different interfaces to access files and may return different error codes, trying to access a non-existing file is an error on all operating systems. The error code returned from operating system specific interfaces is stored in `boost::system::error_code`. The error code that describes the failed access to a non-existing file is stored in `boost::system::error_condition`.
The last class provided by Boost.System is `boost::system::system_error`, which is derived from `std::runtime_error`. It can be used to transport an error code of type `boost::system::error_code` in an exception.

Example 55.5 Using `boost::system::system_error`

```
#include <boost/system/error_code.hpp>
#include <boost/system/system_error.hpp>
```

```
#include <iostream>

using namespace boost::system;

void fail()
{
  throw system_error{errc::make_error_code(errc::not_supported)};
}

int main()
{
  try
  {
    fail();
  }
  catch (system_error &e)
  {
    error_code ec = e.code();
    std::cerr << ec.value() << '\n';
    std::cerr << ec.category().name() << '\n';
  }
}
```

In Example 55.5, the free-standing function `fail()` has been changed to throw an exception of type `boost::system::system_error` in case of an error. This exception can transport an error code of type `boost::system::error_code`. The exception is caught in `main()`, which writes the error code and the error category to standard error. There is a second variant of the function `boost::asio::ip::host_name()` that works just like this.

Chapter 56

Boost.Exception

The library Boost.Exception provides a new exception type, `boost::exception`, that lets you add data to an exception after it has been thrown. This type is defined in `boost/exception/exception.hpp`. Because Boost.Exception spreads its classes and functions over multiple header files, the following examples access the master header file `boost/exception/all.hpp` to avoid including header files one by one.
Boost.Exception supports the mechanism from the C++11 standard that transports an exception from one thread to another. `boost::exception_ptr` is similar to `std::exception_ptr`. However, Boost.Exception isn't a full replacement for the header file `exception` from the standard library. For example, Boost.Exception is missing support for nested exceptions of type `std::nested_exception`.

> Note
>
> To compile the examples in this chapter with Visual C++ 2013, remove the keyword `noexcept`. This version of the Microsoft compiler doesn't support `noexcept` yet.

Example 56.1 calls the function `write_lots_of_zeros()`, which in turn calls `allocate_memory()`. `allocate_memory()` allocates memory dynamically. The function passes `std::nothrow` to `new` and checks whether the return value is 0. If memory allocation fails, an exception of type `allocation_failed` is thrown. `allocation_failed` replaces the exception `std::bad_alloc` thrown by default if `new` fails to allocate memory.
`write_lots_of_zeros()` calls `allocate_memory()` to try and allocate a memory block with the greatest possible size. This is done with the help of `max()` from `std::numeric_limits`. The example intentionally tries to allocate that much memory to make the allocation fail.

Example 56.1 Using `boost::exception`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public boost::exception, public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    throw allocation_failed{};
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

`allocation_failed` is derived from `boost::exception` and `std::exception`. Deriving the class from `std::exception` is not necessary. `allocation_failed` could have also been derived from a class from a different class hierarchy in order to embed it in an existing framework. While Example 56.1 uses the class hierarchy defined by the standard, deriving `allocation_failed` solely from `boost::exception` would have been sufficient.

If an exception of type `allocation_failed` is caught, `allocate_memory()` must be the origin of the exception, since it is the only function that throws exceptions of this type. In programs that have many functions calling `allocate_memory()`, knowing the type of the exception is no longer sufficient to debug the program effectively. In those cases, it would help to know which function tried to allocate more memory than `allocate_memory()` could provide.

The challenge is that `allocate_memory()` does not have any additional information, such as the caller name, to add to the exception. `allocate_memory()` can't enrich the exception. This can only be done in the calling context.

With Boost.Exception, data can be added to an exception at any time. You just need to define a type based on `boost::error_info` for each bit of data you need to add.

`boost::error_info` is a template that expects two parameters. The first parameter is a *tag* that uniquely identifies the newly created type. This is typically a structure with a unique name. The second parameter refers to the type of the value stored inside the exception. Example 56.1 defines a new type, `errmsg_info` – uniquely identifiable via the structure `tag_errmsg` – that stores a string of type `std::string`.

In the `catch` handler of `write_lots_of_zeros()`, `errmsg_info` is used to create an object that is initialized with the string "writing lots of zeros failed." This object is then added to the exception of type `boost::exception` using `operator<<`. Then the exception is re-thrown.

Now, the exception doesn't just denote a failed memory allocation. It also says that the memory allocation failed when the program tried to write lots of zeros in the function `write_lots_of_zeros()`. Knowing which function called `allocate_memory()` makes debugging larger programs easier.

To retrieve all available data from an exception, the function `boost::diagnostic_information()` can be called in the `catch` handler of `main()`. `boost::diagnostic_information()` calls the member function `what()` for each exception passed to it and accesses all of the additional data stored inside the exception. `boost::diagnostic_information()` returns a string of type `std::string`, which, for example, can be written to standard error.

When compiled with Visual C++ 2013, Example 56.1 will display the following message:

```
Throw location unknown (consider using BOOST_THROW_EXCEPTION)
Dynamic exception type: struct allocation_failed
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

The message contains the type of the exception, the error message retrieved from `what()`, and the description, including the name of the structure.
`boost::diagnostic_information()` checks at run time whether or not a given exception is derived from `std::exception`. `what()` will only be called if that is the case.
The name of the function that threw the exception of type `allocation_failed` is unknown.
Boost.Exception provides a macro to throw an exception that contains not only the name of the function, but also additional data such as the file name and the line number.
Using the macro `BOOST_THROW_EXCEPTION` instead of `throw`, data such as function name, file name, and line number are automatically added to the exception. But this only works if the compiler supports macros for the additional data. While macros such as `__FILE__` and `__LINE__` have been standardized since C++98, the macro `__func__`, which gets the name of the current function, only became standard with C++11. Because many compilers provided such a macro before C++11, `BOOST_THROW_EXCEPTION` tries to identify the underlying compiler and use the corresponding macro if it exists.
Compiled with Visual C++ 2013, Example 56.2 displays the following message:

```
main.cpp(20): Throw in function char *__cdecl allocate_memory(unsigned int)
Dynamic exception type: class boost::exception_detail::clone_impl<struct
  boost::exception_detail::error_info_injector<struct allocation_failed> >
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

In Example 56.2, `allocation_failed` is no longer derived from `boost::exception`. `BOOST_THROW_EXCEPTION` accesses the function `boost::enable_error_info()`, which identifies whether or not an exception is derived from `boost::exception`. If not, it creates a new exception type derived from the specified type and `boost::exception`. This is why the message shown above contains a different exception type than `allocation_failed`.

Example 56.2 More data with `BOOST_THROW_EXCEPTION`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Example 56.3 does not use `boost::diagnostic_information()`, it uses `boost::get_error_info()` to directly access the error message of type `errmsg_info`. Because `boost::get_error_info()` returns a smart pointer of type `boost::shared_ptr`, `operator*` is used to fetch the error message. If the parameter passed to `boost::get_error_info()` is not of type `boost::exception`, a null pointer is returned. If the macro `BOOST_THROW_EXCEPTION` is always used to throw an exception, the exception will always be derived from `boost::exception` – there is no need to check the returned smart pointer for null in that case.

Example 56.3 Selectively accessing data with `boost::get_error_info()`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << *boost::get_error_info<errmsg_info>(e);
  }
}
```

Part XIV

Number Handling

The following libraries are all about working with numbers.

- Boost.Integer provides integral types to, for example, specify the exact number of bytes used by a variable.
- Boost.Accumulators provides accumulators that you can pass numbers to when you are calculating values like the mean or standard deviation.
- Boost.MinMax lets you get the smallest and largest number in a container with one function call.
- Boost.Random provides random number generators.
- Boost.NumericConversion provides a cast operator that protects against unintended overflows.

Chapter 57

Boost.Integer

Boost.Integer provides the header file `boost/cstdint.hpp`, which defines specialized types for integers. These definitions originate from the C99 standard. This is a version of the standard for the C programming language that was released in 1999. Because the first version of the C++ standard was released in 1998, it does not include the specialized integer types defined in C99.

C99 defines types in the header file `stdint.h`. This header file was taken into C++11. In C++, it is called `cstdint`. If your development environment supports C++11, you can access `cstdint`, and you don't have to use `boost/cstdint.hpp`.

The types from `boost/cstdint.hpp` are defined in the namespace `boost`. They can be divided into three categories:

- Types such as boost::int8_t and boost::uint64_t carry the exact memory size in their names. Thus, boost::int8_t contains exactly 8 bits, and boost::uint64_t contains exactly 64 bits.
- Types such as boost::int_least8_t and boost::uint_least32_t contain at least as many bits as their names say. It is possible that the memory size of boost::int_least8_t will be greater than 8 bits and that of boost::uint_least32_t will be greater than 32 bits.
- Types such as boost::int_fast8_t and boost::uint_fast16_t also have a minimum size. Their actual size is set to a value that guarantees the best performance. Example 57.1 compiled with Visual C++ 2013 and run on a 64-bit Windows 7 system displays 4 for `sizeof(uif16)`.

Please note that 64-bit types aren't available on all platforms. You can check with the macro `BOOST_NO_INT64_T` whether 64-bit types are available or not.

Example 57.1 Types for integers with number of bits

```
#include <boost/cstdint.hpp>
#include <iostream>

int main()
{
  boost::int8_t i8 = 1;
  std::cout << sizeof(i8) << '\n';

#ifndef BOOST_NO_INT64_T
  boost::uint64_t ui64 = 1;
  std::cout << sizeof(ui64) << '\n';
#endif

  boost::int_least8_t il8 = 1;
  std::cout << sizeof(il8) << '\n';

  boost::uint_least32_t uil32 = 1;
  std::cout << sizeof(uil32) << '\n';

  boost::int_fast8_t if8 = 1;
  std::cout << sizeof(if8) << '\n';

  boost::uint_fast16_t uif16 = 1;
  std::cout << sizeof(uif16) << '\n';
}
```

Boost.Integer defines two types, boost::intmax_t and boost::uintmax_t, for the maximum width integer types available on a platform. Example 57.2 compiled with Visual C++ 2013 and run on a 64-bit Windows 7 system displays `8` for `sizeof(imax)`. Thus, the biggest type for integers contains 64 bits.

Example 57.2 More specialized types for integers

```
#include <boost/cstdint.hpp>
#include <iostream>

int main()
{
  boost::intmax_t imax = 1;
  std::cout << sizeof(imax) << '\n';

  std::cout << sizeof(UINT8_C(1)) << '\n';

#ifndef BOOST_NO_INT64_T
  std::cout << sizeof(INT64_C(1)) << '\n';
#endif
}
```

Furthermore, Boost.Integer provides macros to use integers as literals with certain types. If an integer is written in C++ code, by default it uses the type int and allocates at least 4 bytes. Macros like `UINT8_C` and `INT64_C` make it possible to set a minimum size for integers as literals. Example 57.2 returns at least 1 for `sizeof(UINT8_C(1))` and at least 8 for `sizeof(INT64_C(1))`.
Boost.Integer provides additional header files that mainly define classes used for template meta programming.

Chapter 58

Boost.Accumulators

Boost.Accumulators provides classes to process samples. For example, you can find the largest or smallest sample, or calculate the total of all samples. While the standard library supports some of these operations, Boost.Accumulators also supports statistical calculations, such as mean and standard deviation.

The library is called Boost.Accumulators because the *accumulator* is an essential concept. An accumulator is a container that calculates a new result every time a value is inserted. The value isn't necessarily stored in the accumulator. Instead the accumulator continuously updates intermediary results as it is fed new values.

Boost.Accumulators contains three parts:

- The framework provides the overall structure of the library. It provides the class `boost::accumulators::accumulator_set`, which is always used with Boost.Accumulators. While you need to know about this and a few other classes from the framework, the details don't matter unless you want to develop your own accumulators. The header file `boost/accumulators/accumulators.hpp` gives you access to `boost::accumulators::accumulator_set` and other classes from the framework.
- Boost.Accumulators provides numerous accumulators that perform calculations. You can access and use all of these accumulators once you include `boost/accumulators/statistics.hpp`.
- Boost.Accumulators provides operators to, for example, multiply a complex number of type `std::complex` with an int value or add two vectors. The header file `boost/accumulators/numeric/functional.hpp` defines operators for `std::complex`, `std::valarray`, and `std::vector`. You don't need to include the header file yourself because it is included in the header files for the accumulators. However, you have to define the macros `BOOST_NUMERIC_FUNCTIONAL_STD_COMPLEX_SUPPORT`, `BOOST_NUMERIC_FUNCTIONAL_STD_VALARRAY_SUPPORT`, and `BOOST_NUMERIC_FUNCTIONAL_STD_VECTOR_SUPPORT` to make the operators available.

All classes and functions provided by Boost.Accumulators are defined in `boost::accumulators` or nested namespaces. For example, all accumulators are defined in `boost::accumulators::tag`.

Example 58.1 Counting with `boost::accumulators::tag::count`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<int, features<tag::count>> acc;
  acc(4);
  acc(-6);
  acc(9);
  std::cout << count(acc) << '\n';
}
```

Example 58.1 uses `boost::accumulators::tag::count`, a simple accumulator that counts the number of values passed to it. Thus, since three values are passed, this example writes `3` to standard output. To use an accumulator, you access the class `boost::accumulators::accumulator_set`, which is a template that expects as its first parameter the type of the values that will be processed. Example 58.1 passes int as the first parameter.

The second parameter specifies the accumulators you want to use. You can use multiple accumulators. The class name `boost::accumulators::accumulator_set` indicates that any number of accumulators can be managed.
Strictly speaking, you specify *features*, not accumulators. Features define what should be calculated. You determine the what, not the how. There can be different implementations for features. The implementations are the accumulators.
Example 58.1 uses `boost::accumulators::tag::count` to select an accumulator that counts values. If several accumulators exist that can count values, Boost.Accumulators selects the default accumulator.
Please note that you can't pass features directly to `boost::accumulators::accumulator_set`. You need to use `boost::accumulators::features`.
An object of type `boost::accumulators::accumulator_set` can be used like a function. Values can be passed by calling `operator()`. They are immediately processed. The values passed must have the same type as was passed as the first template parameter to `boost::accumulators::accumulator_set`.
For every feature, there is an identically named *extractor*. An extractor receives the current result of an accumulator. Example 58.1 uses the extractor `boost::accumulators::count()`. The only parameter passed is **`acc`**. `boost::accumulators::count()` returns 3.

Example 58.2 Using `mean` and `variance`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>> acc;
  acc(8);
  acc(9);
  acc(10);
  acc(11);
  acc(12);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Example 58.2 uses the two features `boost::accumulators::tag::mean` and `boost::accumulators::tag::variance` to calculate the mean and the variance of five values. The example writes `10` and `2` to standard output.
The variance is 2 because Boost.Accumulators assigns a weight of 0.2 to each of the five values. The accumulator selected with `boost::accumulators::tag::variance` uses weights. If weights are not set explicitly, all values are given the same weight.

Example 58.3 Calculating the weighted variance

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>, int> acc;
  acc(8, weight = 1);
  acc(9, weight = 1);
  acc(10, weight = 4);
  acc(11, weight = 1);
  acc(12, weight = 1);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Example 58.3 passes int as a third template parameter to `boost::accumulators::accumulator_set`. This parameter specifies the data type of the weights. In this example, weights are assigned to every value.
Boost.Accumulators uses Boost.Parameter to pass additional parameters, such as weights, as name/value pairs. The parameter name for weights is **`weight`**. You can treat the parameter like a variable and assign a value. The name/value pair is passed as an additional parameter after every value to the accumulator.
In Example 58.3, the value 10 has a weight of 4 while all other values have a weight of 1. The mean is still 10 since weights don't matter for means. However, the variance is now 1.25. It has decreased compared to the previous example because the middle value has a higher weight than the other values.
Boost.Accumulators provides many more accumulators. They are used like the accumulators introduced in this chapter. The documentation of the library contains an overview on all available accumulators.

Chapter 59

Boost.MinMax

Boost.MinMax provides an algorithm to find the minimum and the maximum of two values using only one function call, which is more efficient than calling `std::min()` and `std::max()`.
Boost.MinMax is part of C++11. You find the algorithms from this Boost library in the header file `algorithm` if your development environment supports C++11.

Example 59.1 Using `boost::minmax()`

```
#include <boost/algorithm/minmax.hpp>
#include <boost/tuple/tuple.hpp>
#include <iostream>

int main()
{
  int i = 2;
  int j = 1;

  boost::tuples::tuple<const int&, const int&> t = boost::minmax(i, j);

  std::cout << t.get<0>() << '\n';
  std::cout << t.get<1>() << '\n';
}
```

`boost::minmax()` computes the minimum and maximum of two objects. While both `std::min()` and `std::max()` return only one value, `boost::minmax()` returns two values as a tuple. The first reference in the tuple points to the minimum and the second to the maximum. Example 59.1 writes `1` and `2` to the standard output stream.
`boost::minmax()` is defined in `boost/algorithm/minmax.hpp`.

Example 59.2 Using `boost::minmax_element()`

```
#include <boost/algorithm/minmax_element.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  typedef std::array<int, 4> array;
  array a{{2, 3, 0, 1}};

  std::pair<array::iterator, array::iterator> p =
    boost::minmax_element(a.begin(), a.end());

  std::cout << *p.first << '\n';
  std::cout << *p.second << '\n';
}
```

Just as the standard library offers algorithms to find the minimum and maximum values in a container, Boost.MinMax offers the same functionality with only one call to the function `boost::minmax_element()`.

Unlike `boost::minmax()`, `boost::minmax_element()` returns a `std::pair` containing two iterators. The first iterator points to the minimum and the second points to the maximum. Thus, Example 59.2 writes `0` and `3` to the standard output stream.
`boost::minmax_element()` is defined in `boost/algorithm/minmax_element.hpp`.
Both `boost::minmax()` and `boost::minmax_element()` can be called with a third parameter that specifies how objects should be compared. Thus, these functions can be used like the algorithms from the standard library.

Chapter 60

Boost.Random

The library Boost.Random provides numerous random number generators that allow you to decide how random numbers should be generated. It was always possible in C++ to generate random numbers with `std::rand()` from `cstdlib`. However, with `std::rand()` the way random numbers are generated depends on how the standard library was implemented.

You can use all of the random number generators and other classes and functions from Boost.Random when you include the header file `boost/random.hpp`.

Large parts of this library were added to the standard library with C++11. If your development environment supports C++11, you can rewrite the Boost.Random examples in this chapter by including the header file `random` and accessing the namespace `std`.

Example 60.1 Pseudo-random numbers with `boost::random::mt19937`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  std::cout << gen() << '\n';
}
```

Example 60.1 accesses the random number generator `boost::random::mt19937`. The operator `operator()` generates a random number, which is written to standard output.

The random numbers generated by `boost::random::mt19937` are integers. Whether integers or floating point numbers are generated depends on the particular generator you use. All random number generators define the type `result_type` to determine the type of the random numbers. The `result_type` for `boost::random::mt19937` is boost::uint32_t.

All random number generators provide two member functions: `min()` and `max()`. These functions return the smallest and largest number that can be generated by that random number generator.

Nearly all of the random number generators provided by Boost.Random are *pseudo-random number generators*. Pseudo-random number generators don't generate real random numbers. They are based on algorithms that generate seemingly random numbers. `boost::random::mt19937` is one of these pseudo-random number generators.

Pseudo-random number generators typically have to be initialized. If they are initialized with the same values, they return the same random numbers. That's why in Example 60.1 the return value of `std::time()` is passed to the constructor of `boost::random::mt19937`. This should ensure that when the program is run at different times, different random numbers will be generated.

Pseudo-random numbers are good enough for most use cases. `std::rand()` is also based on a pseudo-random number generator, which must be initialized with `std::srand()`. However, Boost.Random provides a random number generator that can generate real random numbers, as long as the operating system has a source to generate real random numbers.

Example 60.2 Real random numbers with `boost::random::random_device`

```
#include <boost/random/random_device.hpp>
#include <iostream>

int main()
{
  boost::random::random_device gen;
  std::cout << gen() << '\n';
}
```

`boost::random::random_device` is a *non-deterministic random number generator*, which is a random number generator that can generate real random numbers. There is no algorithm that needs to be initialized. Thus, predicting the random numbers is impossible. Non-deterministic random number generators are often used in security-related applications.
`boost::random::random_device` calls operating system functions to generate random numbers. If, as in Example 60.2, the default constructor is called, `boost::random::random_device` uses the cryptographic service provider MS_DEF_PROV on Windows and `/dev/urandom` on Linux as a source.
If you want to use another source, call the constructor of `boost::random::random_device`, which expects a parameter of type `std::string`. How this parameter is interpreted depends on the operating system. On Windows, it must be the name of a cryptographic service provider, on Linux a path to a device.
Please note that `boost/random/random_device.hpp` must be included if you want to use the class `boost::random::random_device`. This class is not made available by `boost/random.hpp`.

Example 60.3 The random numbers 0 and 1 with `bernoulli_distribution`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  boost::random::bernoulli_distribution<> dist;
  std::cout << dist(gen) << '\n';
}
```

Example 60.3 uses the pseudo-random number generator `boost::random::mt19937`. In addition, a *distribution* is used. Distributions are Boost.Random classes that map the range of random numbers from a random number generator to another range. While random number generators like `boost::random::mt19937` have a built-in lower and upper limit for random numbers that can be seen using `min()` and `max()`, you may need random numbers in a different range.
Example 60.3 simulates throwing a coin. Because a coin has only two sides, the random number generator should return 0 or 1. `boost::random::bernoulli_distribution` is a distribution that returns one of two possible results.
Distributions are used like random number generators: you call the operator `operator()` to receive a random number. However, you must pass a random number generator as a parameter to a distribution. In Example 60.3, **`dist`** uses the random number generator **`gen`** to return either 0 or 1.

Example 60.4 Random numbers between 1 and 100 with `uniform_int_distribution`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  boost::random::uniform_int_distribution<> dist{1, 100};
  std::cout << dist(gen) << '\n';
}
```

Boost.Random provides numerous distributions. Example 60.4 uses a distribution that is often needed: `boost::random::uniform_int_distribution`. This distribution lets you define the range of random numbers you need. In Example 60.4, **`dist`** returns a number between 1 and 100.
Please note that the values 1 and 100 can be returned by **`dist`**. The lower and upper limits of distributions are inclusive.
There are many distributions in Boost.Random besides `boost::random::bernoulli_distribution` and `boost::random::uniform_int_distribution`. For example, there are distributions like `boost::random::normal_distribution` and `boost::random::chi_squared_distribution`, which are used in statistics.

Chapter 61

Boost.NumericConversion

The library Boost.NumericConversion can be used to convert numbers of one numeric type to a different numeric type. In C++, such a conversion can also take place implicitly, as shown in Example 61.1.

Example 61.1 Implicit conversion from int to short

```
#include <iostream>

int main()
{
  int i = 0x10000;
  short s = i;
  std::cout << s << '\n';
}
```

Example 61.1 will compile cleanly because the type conversion from int to short takes place automatically. However, even though the program will run, the result of the conversion depends on the compiler used. The number `0x10000` in the variable **`i`** is too big to be stored in a variable of type short. According to the standard, the result of this operation is implementation specific. Compiled with Visual C++ 2013, the program displays `0`, which clearly differs from the value in **`i`**.

To avoid these kind of problems, you can use the cast operator `boost::numeric_cast` (see Example 61.2). `boost::numeric_cast` is used exactly like the existing C++ cast operators. The correct header file must be included; in this case, the header file `boost/numeric/conversion/cast.hpp`.

`boost::numeric_cast` does the same conversion as C++, but it verifies whether the conversion can take place without changing the value being converted. In Example 61.2, this verification fails, and an exception of type `boost::numeric::bad_numeric_cast` is thrown because `0x10000` is too big to be placed in a variable of type short.

Example 61.2 Overflow detection with `boost::numeric_cast`

```
#include <boost/numeric/conversion/cast.hpp>
#include <iostream>

int main()
{
  try
  {
    int i = 0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << '\n';
  }
  catch (boost::numeric::bad_numeric_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Strictly speaking, an exception of type `boost::numeric::positive_overflow` will be thrown. This type specifies an overflow – in this case for positive numbers. There is also `boost::numeric::negative_overflow`, which specifies an overflow for negative numbers (see Example 61.3).

Example 61.3 Overflow detection for negative numbers

```
#include <boost/numeric/conversion/cast.hpp>
#include <iostream>

int main()
{
  try
  {
    int i = -0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << '\n';
  }
  catch (boost::numeric::negative_overflow &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Boost.NumericConversion defines additional exception types, all derived from `boost::numeric::bad_numeric_cast`. Because `boost::numeric::bad_numeric_cast` is derived from `std::bad_cast`, a `catch` handler can also catch exceptions of this type.

Part XV

Application Libraries

Application libraries refers to libraries that are typically used exclusively in the development of stand-alone applications and not in the development of libraries.

- Boost.Log is a logging library.
- Boost.ProgramOptions is a library to define and parse command line options.
- Boost.Serialization lets you serialize objects to, for example, save them to and load them from files.
- Boost.Uuid supports working with UUIDs.

Chapter 62

Boost.Log

Boost.Log is the logging library in Boost. It supports numerous back-ends to log data in various formats. Back-ends are accessed through front-ends that bundle services and forward log entries in different ways. For example, there is a front-end that uses a thread to forward log entries asynchronously. Front-ends can have filters to ignore certain log entries. And they define how log entries are formatted as strings. All these functions are extensible, which makes Boost.Log a powerful library.
Example 62.1 introduces the essential components of Boost.Log. Boost.Log gives you access to back-ends, front-ends, the core, and loggers:

- Back-ends decide where data is written. `boost::log::sinks::text_ostream_backend` is initialized with a stream of type `std::ostream` and writes log entries to it.
- Front-ends are the connection between the core and a back-end. They implement various functions that don't need to be implemented by each individual back-end. For example, filters can be added to a front-end to choose which log entries get forwarded to the back-end and which don't.

  Example 62.1 uses the front-end `boost::log::sinks::asynchronous_sink`. You must use a front-end even if you don't use filters. `boost::log::sinks::asynchronous_sink` uses a thread that forwards log entries to a back-end asynchronously. This can improve the performance but defers write operations.
- The core is the central component that all log entries are routed through. It is implemented as a singleton. To get a pointer to the core, call `boost::log::core::get()`.

  Front-ends must be added to the core to receive log entries. Whether log entries are forwarded to front-ends depends on the filter in the core. Filters can be registered either in front-ends or in the core. Filters registered in the core are global, and filters registered in front-ends are local. If a log entry is filtered out by the core, it isn't forwarded to any front-end. If it is filtered by a front-end, it can still be processed by other front-ends and forwarded to their back-ends.
- The logger is the component in Boost.Log you will use most often. While you access back-ends, front-ends, and the core only when you initialize the logging library, you use a logger every time you write a log entry. The logger forwards the entry to the core.

  The logger in Example 62.1 is of the type `boost::log::sources::logger`. This is the simplest logger. When you want to write a log entry, use the macro `BOOST_LOG` and pass the logger as a parameter. The log entry is created by writing data into the macro as if it is a stream of type `std::ostream`.

Example 62.1 Back-end, front-end, core, and logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
```

```
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  sources::logger lg;

  BOOST_LOG(lg) << "note";
  sink->flush();
}
```

Back-end, front-end, core, and logger work together. `boost::log::sinks::asynchronous_sink`, a front-end, is a template that receives the back-end `boost::log::sinks::text_ostream_backend` as a parameter. Afterwards, the front-end is instantiated with `boost::shared_ptr`. The smart pointer is required to register the front-end in the core: the call to `boost::log::core::add_sink()` expects a `boost::shared_ptr`.
Because the back-end is a template parameter of the front-end, it can only be configured after the front-end has been instantiated. The back-end determines how this is done. The member function `add_stream()` is provided by the back-end `boost::log::sinks::text_ostream_backend` to add streams. You can add more than one stream to `boost::log::sinks::text_ostream_backend`. Other back-ends provide different member functions for configuration. Consult the documentation for details.
To get access to a back-end, all front-ends provide the member function `locked_backend()`. This member function is called `locked_backend()` because it returns a pointer that provides synchronized access to the back-end as long as the pointer exists. You can access a back-end through pointers returned by `locked_backend()` from multiple threads without having to synchronize access yourself.
You can instantiate a logger like `boost::log::sources::logger` with the default constructor. The logger automatically calls `boost::log::core::get()` to forward log entries to the core.
You can access loggers without macros. Loggers are objects with member functions you can call. However, macros like `BOOST_LOG` make it easier to write log entries. Without macros it wouldn't be possible to write a log entry in one line of code.
Example 62.1 calls `boost::log::sinks::asynchronous_sink::flush()` at the end of `main()`. This call is required because the front-end is asynchronous and uses a thread to forward log entries. The call makes sure that all buffered log entries are passed to the back-end and are written. Without the call to `flush()`, the example could terminate without displaying `note`.
Example 62.2 is based on Example 62.1, but it replaces `boost::sources::logger` with the logger `boost::sources::severity_logger`. This logger adds an attribute for a log level to every log entry. You can use the macro `BOOST_LOG_SEV` to set the log level.
The type of the log level depends on a template parameter passed to `boost::sources::severity_logger`. Example 62.2 uses int. That's why numbers like 0 and 1 are passed to `BOOST_LOG_SEV`. If `BOOST_LOG` is used, the log level is set to 0.
Example 62.2 also calls `set_filter()` to register a filter at the front-end. The filter function is called for every log entry. If the function returns `true`, the log entry is forwarded to the back-end. Example 62.2 defines the function `only_warnings()` with a return value of type bool.
`only_warnings()` expects a parameter of type `boost::log::attribute_value_set`. This type represents log entries while they are being passed around in the logging framework. `boost::log::record` is another type for log entries that is like a wrapper for `boost::log::attribute_value_set`. This type provides the member function `attribute_values()`, which retrieves a reference to the `boost::log::attribute_value_set`. Filter functions receive a `boost::log::attribute_value_set` directly and no `boost::log::record`. `boost::log::attribute_value_set` stores key/value pairs. Think of it as a `std::unordered_map`.
Log entries consist of attributes. Attributes have a name and a value. You can create attributes yourself. They can also be created automatically – for example by loggers. In fact, that's why Boost.Log provides multiple loggers. `boost::log::sources::severity_logger` adds an attribute called Severity to every log entry. This attribute stores the log level. That way a filter can check whether the log level of a log entry is greater than 0.

Example 62.2 `boost::sources::severity_logger` with a filter

```
#include <boost/log/common.hpp>
```

```
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

bool only_warnings(const attribute_value_set &set)
{
  return set["Severity"].extract<int>() > 0;
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(&only_warnings);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG(lg) << "note";
  BOOST_LOG_SEV(lg, 0) << "another note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

`boost::log::attribute_value_set` provides several member functions to access attributes. The member functions are similar to the ones provided by `std::unordered_map`. For example, `boost::log::attribute_value_set` overloads the operator `operator[]`. This operator returns the value of an attribute whose name is passed as a parameter. If the attribute doesn't exist, it is created.
The type of attribute names is `boost::log::attribute_name`. This class provides a constructor that accepts a string, so you can pass a string directly to `operator[]`, as in Example 62.2.
The type of attribute values is `boost::log::attribute_value`. This class provides member functions to receive the value in the attribute's original type. Because the log level is an int value, int is passed as a template parameter to `extract()`.
`boost::log::attribute_value` also defines the member functions `extract_or_default()` and `extract_or_throw()`. `extract()` returns a value created with the default constructor if a type conversion fails – for example 0 in case of an int. `extract_or_default()` returns a default value which is passed as another parameter to that member function. `extract_or_throw()` throws an exception of type `boost::log::runtime_error` in the event of an error.
For type-safe conversions, Boost.Log provides the visitor function `boost::log::visit()`, which you can use instead of `extract()`.
Example 62.2 displays `warning`. This log entry has a log level greater than 0 and thus isn't filtered.

Example 62.3 Changing the format of a log entry with `set_formatter()`

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

void severity_and_message(const record_view &view, formatting_ostream &os)
{
  os << view.attribute_values()["Severity"].extract<int>() << ": " <<
    view.attribute_values()["Message"].extract<std::string>();
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_formatter(&severity_and_message);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Example 62.3 is based on Example 62.2. This time the log level is displayed.
Front-ends provide the member function `set_formatter()`, which can be passed a format function. If a log entry isn't filtered by a front-end, it is forwarded to the format function. This function formats the log entry as a string that is then passed from the front-end to the back-end. If you don't call `set_formatter()`, by default the back-end only receives what is on the right side of a macro like `BOOST_LOG`.
Example 62.3 passes the function `severity_and_message()` to `set_formatter()`. `severity_and_message()` expects parameters of type `boost::log::record_view` and `boost::log::formatting_ostream`. `boost::log::record_view` is a view on a log entry. It's similar to `boost::log::record`. However, `boost::log::record_view` is an immutable log entry.
`boost::log::record_view` provides the member function `attribute_values()`, which returns a constant reference to `boost::log::attribute_value_set`. `boost::log::formatting_ostream` is the stream used to create the string that is passed to the back-end.
`severity_and_message()` accesses the attributes Severity and Message. `extract()` is called to get the attribute values, which are then written to the stream. Severity returns the log level as an int value. Message provides access to what is on the right side of a macro like `BOOST_LOG`. Consult the documentation for a complete list of available attribute names.
Example 62.3 uses no filter. The example writes two log entries: `0:note` and `1:warning`.
Example 62.4 uses both a filter and a format function. This time the functions are implemented as lambda functions – not as C++11 lambda functions but as Boost.Phoenix lambda functions.
Boost.Log provides helpers for lambda functions in the namespace `boost::log::expressions`. For example, **`boost::log::expressions::stream`** represents the stream. **`boost::log::expressions::smessage`** provides access to everything on the right side of a macro like `BOOST_LOG`. You can use `boost::log::expressions::attr()` to access any attribute. Instead of **`smessage`** Example 62.4 could use `attr<std::string>("Message")`.
Example 62.4 displays `1:warning` and `2:error`.

Example 62.4 Filtering log entries and formatting them with lambda functions

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::attr<int>("Severity") > 0);
  sink->set_formatter(expressions::stream <<
    expressions::attr<int>("Severity") << ": " << expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Boost.Log supports user-defined keywords. You can use the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define keywords to access attributes without having to repeatedly pass attribute names as strings to `boost::log::expressions::attr()`.
Example 62.5 uses the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define a keyword **`severity`**. The macro expects three parameters: the name of the keyword, the attribute name as a string, and the type of the attribute. The new keyword can be used in filter and format lambda functions. This means you are not restricted to using keywords, such as **`boost::log::expressions::smessage`**, that are provided by Boost.Log – you can also define new keywords.

Example 62.5 Defining keywords for attributes

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << severity << ": " <<
    expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

In all of the examples so far, the attributes used are the ones defined in Boost.Log. Example 62.6 shows how to create user-defined attributes.

You create a global attribute by calling `add_global_attribute()` on the core. The attribute is global because it is added to every log entry automatically.

`add_global_attribute()` expects two parameters: the name and the type of the new attribute. The name is passed as a string. For the type you use a class from the namespace `boost::log::attributes`, which provides classes to define different attributes. Example 62.6 uses `boost::log::attributes::counter` to define the attribute LineCounter, which adds a line number to every log entry. This attribute will number log entries starting at 1.

`add_global_attribute()` is not a function template. `boost::log::attributes::counter` isn't passed as a template parameter. The attribute type must be instantiated and passed as an object.

Example 62.6 Defining attributes

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << counter << " - " << severity <<
    ": " << expressions::smessage << " (" << timestamp << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

Example 62.6 uses a second attribute called Timestamp. This is a scoped attribute that is created with `BOOST_LOG_SCOPED_LOGGER_ATTR`. This macro adds an attribute to a logger. The first parameter is the logger, the second is the attribute name, and the third is the attribute object. The type of the attribute object is `boost::log::attribute::local_clock`. The attribute is set to the current time for each log entry.

The attribute Timestamp is added to the log entry "error" only. Timestamp exists only in the scope where `BOOST_LOG_SCOPED_LOGGER_ATTR` is used. When the scope ends, the attribute is removed. `BOOST_LOG_SCOPED_LOGGER_ATTR` is similar to a call to `add_attribute()` and `remove_attribute()`.

Example 62.7 Helper functions for filters and formats

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <iomanip>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::is_in_range(severity, 1, 3));
  sink->set_formatter(expressions::stream << std::setw(5) << counter <<
    " - " << severity << ": " << expressions::smessage << " (" <<
    expressions::format_date_time(timestamp, "%H:%M:%S") << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

As in Example 62.5, Example 62.6 uses the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define keywords for the new attributes. The format function accesses the keywords to write the line number and current time. The value of **`timestamp`** will be an empty string for those log entries where the attribute Timestamp is undefined. Boost.Log provides numerous helper functions for filters and formats. Example 62.7 calls the helper `boost::log::expressions::is_in_range()` to filter log entries whose log level is outside a range. `boost::log::expressions::is_in_range()` expects the attribute as its first parameter and lower and upper bounds as its second and third parameters. As with iterators, the upper bound is exclusive and doesn't belong to the range. `boost::log::expressions::format_date_time()` is called in the format function. It is used to format a timepoint. Example 62.7 uses `boost::log::expressions::format_date_time()` to write the time without a date. You can also use manipulators from the standard library in format functions. Example 62.7 uses `std::setw()` to set the width for the counter.

Example 62.8 uses several loggers, front-ends, and back-ends. In addition to using the classes `boost::log::sinks::asynchronous_sink`, `boost::log::sinks::text_ostream_backend` and `boost::log::sources::severity_logger`, the example also uses the front-end `boost::log::sinks::synchronous_sink`, the back-end `boost::log::sinks::text_multifile_backend`, and the logger `boost::log::sources::channel_logger`.
The front-end `boost::log::sinks::synchronous_sink` provides synchronous access to a back-end, which lets you use a back-end in a multithreaded application even if the back-end isn't thread safe.
The difference between the two front-ends `boost::log::sinks::asynchronous_sink` and `boost::log::sinks::synchronous_sink` is that the latter isn't based on a thread. Log entries are passed to the back-end in the same thread.
Example 62.8 uses the front-end `boost::log::sinks::synchronous_sink` with the back-end `boost::log::sinks::text_multifile_backend`. This back-end writes log entries to one or more files. File names are created according to a rule passed by `set_file_name_composer()` to the back-end. If you use the free-standing function `boost::log::sinks::file::as_file_name_composer()`, as in the example, the rule can be created as a lambda function with the same building blocks used for format functions. However, the attributes aren't used to create the string that is written to a back-end. Instead, the string will be the name of the file that log entries will be written to.
Example 62.8 uses the keywords **`channel`** and **`severity`**, which are defined with the macro `BOOST_LOG_ATTRIBUTE_KEYWORD`. They refer to the attributes Channel and Severity. The member function `or_default()` is called on the keywords to pass a default value if an attribute isn't set. If a log entry is written and Channel and Severity are not set, the entry is written to the file `None-0.log`. If a log entry is written with the log level 1, it is stored in the file `None-1.log`. If the log level is 1 and the channel is called Main, the log entry is saved in the file `Main-1.log`.
The attribute Channel is defined by the logger `boost::log::sources::channel_logger`. The constructor expects a channel name. The name can't be passed directly as a string. Instead, it must be passed as a named parameter. That's why the example uses `keywords::channel ="Main"` even though `boost::log::sources::channel_logger` doesn't accept any other parameters.

Example 62.8 Several loggers, front-ends, and back-ends

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/channel_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/utility/string_literal.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <string>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(channel, "Channel", std::string)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend>
    ostream_sink;
  boost::shared_ptr<ostream_sink> ostream =
    boost::make_shared<ostream_sink>();
  boost::shared_ptr<std::ostream> clog{&std::clog,
    boost::empty_deleter{}};
  ostream->locked_backend()->add_stream(clog);
  core::get()->add_sink(ostream);

  typedef sinks::synchronous_sink<sinks::text_multifile_backend>
    multifile_sink;
  boost::shared_ptr<multifile_sink> multifile =
    boost::make_shared<multifile_sink>();
  multifile->locked_backend()->set_file_name_composer(
    sinks::file::as_file_name_composer(expressions::stream <<
    channel.or_default<std::string>("None") << "-" <<
    severity.or_default(0) << ".log"));
  core::get()->add_sink(multifile);

  sources::severity_logger<int> severity_lg;
  sources::channel_logger<> channel_lg{keywords::channel = "Main"};

  BOOST_LOG_SEV(severity_lg, 1) << "severity message";
  BOOST_LOG(channel_lg) << "channel message";
  ostream->flush();
}
```

Please note that the named parameter `boost::log::keywords::channel` has nothing to do with the keywords you create with the macro `BOOST_LOG_ATTRIBUTE_KEYWORD`.
`boost::log::sources::channel_logger` identifies log entries from different components of a program. Components can use their own objects of type `boost::log::sources::channel_logger`, giving them unique names. If components only access their own loggers, it's clear which component a particular log entry came from.

Example 62.9 Handling exceptions centrally

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/log/utility/exception_handler.hpp>
#include <boost/log/exceptions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

struct handler
{
  void operator()(const runtime_error &ex) const
  {
    std::cerr << "boost::log::runtime_error: " << ex.what() << '\n';
  }

  void operator()(const std::exception &ex) const
  {
    std::cerr << "std::exception: " << ex.what() << '\n';
  }
};

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);
  core::get()->set_exception_handler(
    make_exception_handler<runtime_error, std::exception>(handler{}));

  sources::logger lg;

  BOOST_LOG(lg) << "note";
}
```

Boost.Log provides the option to handle exceptions in the logging framework centrally. This means you don't need to wrap every `BOOST_LOG` in a `try` block to handle exceptions in `catch`.
Example 62.9 calls the member function `set_exception_handler()`. The core provides this member function to register a handler. All exceptions in the logging framework will be passed to that handler. The handler is implemented as a function object. It has to overload `operator()` for every exception type expected. An instance of that function object is passed to `set_exception_handler()` through the function template `boost::log::make_exception_handler()`. All exception types you want to handle must be passed as template parameters to `boost::log::make_exception_handler()`.
The function `boost::log::make_exception_suppressor()` let's you discard all exceptions in the logging framework. You call this function instead of `boost::log::make_exception_handler()`.

Example 62.10 A macro to define a global logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(lg, sources::wlogger_mt)

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  BOOST_LOG(lg::get()) << L"note";
}
```

All of the examples in this chapter use local loggers. If you want to define a global logger, use the macro `BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT` as in Example 62.10. You pass the name of the logger as the first parameter and the type as the second. You don't access the logger through its name. Instead, you call `get()`, which returns a pointer to a singleton.
Boost.Log provides additional macros such as `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS`. They let you initialize global loggers. `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS` lets you pass parameters to the constructor of a global logger. All of these macros guarantee that global loggers will be correctly initialized.
Boost.Log provides many more functions that are worth a look. For example, you can configure the logging framework through a container with key/value pairs as strings. Then, you don't need to instantiate classes and call member functions. For example, a key Destination can be set to Console, which will automatically make the logging framework use the back-end `boost::log::sinks::text_ostream_backend`. The back-end can be configured through additional key/value pairs. Because the container can also be serialized in an INI-file, it is possible to store the configuration in a text file and initialize the logging framework with that file.

Chapter 63

Boost.ProgramOptions

Boost.ProgramOptions is a library that makes it easy to parse command-line options, for example, for console applications. If you develop applications with a graphical user interface, command-line options are usually not important.
To parse command-line options with Boost.ProgramOptions, the following three steps are required:

1. Define command-line options. You give them names and specify which ones can be set to a value. If a command-line option is parsed as a key/value pair, you also set the type of the value – for example, whether it is a string or a number.

2. Use a parser to evaluate the command line. You get the command line from the two parameters of `main()`, which are usually called **`argc`** and **`argv`**.

3. Store the command-line options evaluated by the parser. Boost.ProgramOptions offers a class derived from `std::map` that saves command-line options as name/value pairs. Afterwards, you can check which options have been stored and what their values are.

Example 63.1 shows the basic approach for parsing command-line options with Boost.ProgramOptions.

Example 63.1 Basic approach with Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <iostream>

using namespace boost::program_options;

void on_age(int age)
{
  std::cout << "On age: " << age << '\n';
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->default_value(3.14f), "Pi")
      ("age", value<int>()->notifier(on_age), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, desc), vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << vm["age"].as<int>() << '\n';
    else if (vm.count("pi"))
```

```
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

To use Boost.ProgramOptions, include the header file `boost/program_options.hpp`. You can access all classes and functions from this library in the namespace `boost::program_options`.
Use the class `boost::program_options::options_description` to describe command-line options. An object of this type can be written to a stream such as **`std::cout`** to display an overview of available command-line options. The string passed to the constructor gives the overview a name that acts as a title for the command-line options.
`boost::program_options::options_description` defines a member function `add()` that expects a parameter of type `boost::program_options::option_description`. You call this function to describe each command-line option. Instead of calling this function for every command-line option, Example 63.1 calls the member function `add_options()`, which makes that task easier.
`add_options()` returns a proxy object representing an object of type `boost::program_options::options_description`. The type of the proxy object doesn't matter. It's more interesting that the proxy object simplifies defining many command-line options. It uses the overloaded operator `operator()`, which you can call to pass the required data to define a command-line option. This operator returns a reference to the same proxy object, which allows you to call `operator()` multiple times.
Example 63.1 defines three command-line options with the help of the proxy object. The first command-line option is `--help`. The description of this option is set to "Help screen". The option is a switch, not a name/value pair. You set `--help` on the command line or omit it. It's not possible to set `--help` to a value.
Please note that the first string passed to `operator()` is "help,h". You can specify short names for command-line options. A short name must consist of just one letter and is set after a comma. Now the help can be displayed with either `--help` or `-h`.
Besides `--help`, two more command-line options are defined: `--pi` and `--age`. These options aren't switches, they're name/value pairs. Both `--pi` and `--age` expect to be set to a value.
You pass a pointer to an object of type `boost::program_options::value_semantic` as the second parameter to `operator()` to define an option as a name/value pair. You don't need to access `boost::program_options::value_semantic` directly. You can use the helper function `boost::program_options::value()`, which creates an object of type `boost::program_options::value_semantic`. `boost::program_options::value()` returns the object's address, which you then can pass to the proxy object using `operator()`.
`boost::program_options::value()` is a function template that takes the type of the command-line option value as a template parameter. Thus, the command-line option `--age` expects an integer and `--pi` expects a floating point number.
The object returned from `boost::program_options::value()` provides some useful member functions. For example, you can call `default_value()` to provide a default value. Example 63.1 sets `--pi` to 3.14 if that option isn't used on the command line.
`notifier()` links a function to a command-line option's value. That function is then called with the value of the command-line option. In Example 63.1, the function `on_age()` is linked to `--age`. If the command-line option `--age` is used to set an age, the age is passed to `on_age()` which writes it to standard output.
Processing values with functions like `on_age()` is optional. You don't have to use `notifier()` because it's possible to access values in other ways.
After all command-line options have been defined, you use a parser. In Example 63.1, the helper function `boost::program_options::parse_command_line()` is called to parse the command line. This function takes **`argc`** and **`argv`**, which define the command line, and **`desc`**, which contains the option descriptions. `boost::program_options::parse_command_line()` returns the parsed options in an object of type `boost::program_options::parsed_options`. You usually don't access this object directly. Instead you pass it to `boost::program_options::store()`, which stores the parsed options in a container.
Example 63.1 passes **`vm`** as a second parameter to `boost::program_options::store()`. **`vm`** is an object of type `boost::program_options::variables_map`. This class is derived from the class std::map<std::string, boost::program_options::variable_value> and, thus, provides the same member functions as `std::map`. For example, you can call `count()` to check whether a certain command-line option has been used and is stored in the container.

In Example 63.1, before **`vm`** is accessed and `count()` is called, `boost::program_options::notify()` is called. This function triggers functions, such as `on_age()`, that are linked to a value using `notifier()`. Without `boost::program_options::notify()`, `on_age()` would not be called.
`vm` lets you check whether a certain command-line option exists, and it also lets you access the value the command-line option is set to. The value's type is `boost::program_options::variable_value`, a class that uses `boost::any` internally. You can get the object of type `boost::any` from the member function `value()`.
Example 63.1 calls `as()`, not `value()`. This member function converts the value of a command-line option to the type passed as a template parameter. `as()` uses `boost::any_cast()` for the type conversion.
Be sure the type you pass to `as()` matches the type of the command-line option. For example, Example 63.1 expects the command-line option `--age` to be set to a number of type int, so int must be passed as a template parameter to `as()`.
You can start Example 63.1 in many ways. Here is one example:

```
test
```

In this case `Pi:3.14` is displayed. Because `--pi` isn't set on the command line, the default value is displayed.
This example sets a value using `--pi`:

```
test --pi 3.1415
```

The program now displays `Pi:3.1415`.
This example also passes an age:

```
test --pi 3.1415 --age 29
```

The output is now `On age:29` and `Age:29`. The first line is written when `boost::program_options::notify()` is called; this triggers the execution of `on_age()`. There is no output for `--pi` because the program uses `else if` statements that only display the value set with `--pi` if `--age` is not set.
This example shows the help:

```
test -h
```

You get a complete overview on all command-line options:

```
Options:
  -h [ --help ]          Help screen
  --pi arg (=3.1400001) Pi
  --age arg              Age
```

As you can see, the help can be shown in two different ways because a short name for that command-line option was defined. For `--pi` the default value is displayed. The command-line options and their descriptions are formatted automatically. You only need to write the object of type `boost::program_options::options_description` to standard output as in Example 63.1.
Now, start the example like this:

```
test --age
```

The output is the following:

```
the required argument for option '--age' is missing.
```

Because `--age` isn't set, the parser used in `boost::program_options::parse_command_line()` throws an exception of type `boost::program_options::error`. The exception is caught, and an error message is written to standard output.
`boost::program_options::error` is derived from `std::logic_error`. Boost.ProgramOptions defines additional exceptions, which are all derived from `boost::program_options::error`. One of those exceptions is `boost::program_options::invalid_syntax`, which is the exact exception thrown in Example 63.1 if you don't supply a value for `--age`.
Example 63.2 introduces more configuration settings available with Boost.ProgramOptions.

Example 63.2 Special configuration settings with Boost.ProgramOptions

```cpp
#include <boost/program_options.hpp>
#include <string>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(), std::ostream_iterator<std::string>{
    std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    int age;

    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->implicit_value(3.14f), "Pi")
      ("age", value<int>(&age), "Age")
      ("phone", value<std::vector<std::string>>()->multitoken()->
        zero_tokens()->composing(), "Phone")
      ("unreg", "Unrecognized options");

    command_line_parser parser{argc, argv};
    parser.options(desc).allow_unregistered().style(
      command_line_style::default_style |
      command_line_style::allow_slash_for_short);
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << age << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
    else if (vm.count("unreg"))
      to_cout(collect_unrecognized(parsed_options.options,
        exclude_positional));
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.2 parses command-line options like the previous example does. However, there are some notable differences. For example, `implicit_value()` is called, rather than `default_value()`, when defining the `--pi` command-line option. This means that pi isn't set to 3.14 by default. `--pi` must be set on the command line

for pi to be available. However, you don't need to supply a value to the `--pi` command-line option if you use `implicit_value()`. It's sufficient to pass `--pi` without setting a value. In that case, pi is set to 3.14 implicitly.
For the command-line option `--age`, a pointer to the variable **`age`** is passed to `boost::program_options::value()`. This stores the value of a command-line option in a variable. Of course, the value is still available in the container **`vm`**.
Please note that a value is only stored in **`age`** if `boost::program_options::notify()` is called. Even though `notifier()` isn't used in this example, `boost::program_options::notify()` still must be used. To avoid problems, it's a good idea to always call `boost::program_options::notify()` after parsed command-line options have been stored with `boost::program_options::store()`.
Example 63.2 supports a new command-line option `--phone` to pass a phone number to the program. In fact, you can pass multiple phone numbers on the command line. For example, the following command line starts the program with the phone numbers 123 and 456:

```
test --phone 123 456
```

Example 63.2 supports multiple phone numbers because `multitoken()` is called on this command-line option's value. And, since `zero_tokens()` is called, `--phone` can also be used without passing a phone number.
You can also pass multiple phone numbers by repeating the `--phone` option, as shown in the following command line:

```
test --phone 123 --phone 456
```

In this case, both phone numbers, 123 and 456, are parsed. The call to `composing()` makes it possible to use a command-line option multiple times – the values are composed.
The value of the argument to `--phone` is of type std::vector<std::string>. You need to use a container to store multiple phone numbers.
Example 63.2 defines another command-line option, `--unreg`. This is a switch that can't be set to a value. It is used later in the example to decide whether command-line options that aren't defined in **`desc`** should be displayed.
While Example 63.1 calls the function `boost::program_options::parse_command_line()` to parse command-line options, Example 63.2 uses a parser of type `boost::program_options::command_line_parser`. **`argc`** and **`argv`** are passed to the constructor.
`boost::program_options::command_line_parser` provides several member functions. You must call `options()` to pass the definition of command-line options to the parser.
Like other member functions, `options()` returns a reference to the same parser. That way, member functions can be easily called one after another. Example 63.2 calls `allow_unregistered()` after `options()` to tell the parser not to throw an exception if unknown command-line options are detected. Finally, `style()` is called to tell the parser that short names can be used with a slash. Thus, the short name for the `--help` option can be either `-h` or `/h`.
Please note that `boost::program_options::parse_command_line()` supports a fourth parameter, which is forwarded to `style()`. If you want to use an option like **`boost::program_options::command_line_style::allow_slash_for_short`**, you can still use the function `boost::program_options::parse_command_line()`.
After the configuration has been set, call `run()` on the parser. This member function returns the parsed command-line options in an object of type `boost::program_options::parsed_options`, which you can pass to `boost::program_options::store()` to store the options in **`vm`**.
Later in the code, Example 63.2 accesses **`vm`** again to evaluate command-line options. Only the call to `boost::program_options::collect_unrecognized()` is new. This function is called for the command-line option `--unreg`. The function expects an object of type `boost::program_options::parsed_options`, which is returned by `run()`. It returns all unknown command-line options in a std::vector<std::string>. For example, if you start the program with **test --unreg --abc**, `--abc` will be written to standard output.
When **`boost::program_options::exclude_positional`** is passed as the second parameter to `boost::program_options::collect_unrecognized()`, positional options are ignored. For Example 63.2, this doesn't matter because no positional options are defined. However, `boost::program_options::collect_unrecognized()` requires this parameter.
Example 63.3 illustrates positional options.

Example 63.3 Positional options with Boost.ProgramOptions

```
#include <boost/program_options.hpp>
#include <string>
#include <vector>
```

```
#include <algorithm>
#include <iterator>
#include <iostream>

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(),
    std::ostream_iterator<std::string>{std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("phone", value<std::vector<std::string>>()->
        multitoken()->zero_tokens()->composing(), "Phone");

    positional_options_description pos_desc;
    pos_desc.add("phone", -1);

    command_line_parser parser{argc, argv};
    parser.options(desc).positional(pos_desc).allow_unregistered();
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.3 defines `--phone` as a positional option using the class `boost::program_options::positional_options_description`. This class provides the member function `add()`, which expects the name of the command-line option and a position to be passed. The example passes “phone” and -1.
With positional options, values can be set on the command line without using command-line options. You can start Example 63.3 like this:

```
test 123 456
```

Even though `--phone` isn’t used, 123 and 456 are recognized as phone numbers.
Calling `add()` on an object of type `boost::program_options::positional_options_description` assigns values on the command line to command-line options using position numbers. When Example 63.3 is called using the command line **test 123 456**, 123 has the position number 0 and 456 has the position number 1. Example 63.3 passes -1 to `add()`, which assigns all of the values – 123 and 456 – to `--phone`. If you changed Example 63.3 to pass the value 0 to `add()`, only 123 would be recognized as a phone number. And if 1 was passed to `add()`, only 456 would be recognized.
`pos_desc` is passed with `positional()` to the parser. That’s how the parser knows which command-line options are positional.
Please note that you have to make sure that positional options are defined. In Example 63.3, for example, “phone” could only be passed to `add()` because a definition for `--phone` already existed in **`desc`**.

In all previous examples, Boost.ProgramOptions was used to parse command-line options. However, the library supports loading configuration options from a file, too. This can be useful if the same command-line options have to be set repeatedly.
Example 63.4 uses two objects of type `boost::program_options::options_description`. **`generalOptions`** defines options that must be set on the command line. **`fileOptions`** defines options that can be loaded from a configuration file.
It's not mandatory to define options with two different objects of type `boost::program_options::options_description`. You can use just one if the set of options is the same for both command line and file. In Example 63.4, separating options makes sense because you don't want to allow `--help` to be set in the configuration file. If that was allowed and the user put that option in the configuration file, the program would display the help screen every time.
Example 63.4 loads `--age` from a configuration file. You can pass the name of the configuration file as a command-line option. In this example, `--config` is defined in **`generalOptions`** for that reason.
After the command-line options have been parsed with `boost::program_options::parse_command_line()` and stored in **`vm`**, the example checks whether `--config` is set. If it is, the configuration file is opened with `std::ifstream`. The `std::ifstream` object is passed to the function `boost::program_options::parse_config_file()` along with **`fileOptions`**, which describes the options. `boost::program_options::parse_config_file()` does the same thing as `boost::program_options::parse_command_line()` and returns parsed options in an object of type `boost::program_options::parsed_options`. This object is passed to `boost::program_options::store()` to store the parsed options in **`vm`**.

Example 63.4 Loading options from a configuration file

```
#include <boost/program_options.hpp>
#include <string>
#include <fstream>
#include <iostream>

using namespace boost::program_options;

int main(int argc, const char *argv[])
{
  try
  {
    options_description generalOptions{"General"};
    generalOptions.add_options()
      ("help,h", "Help screen")
      ("config", value<std::string>(), "Config file");

    options_description fileOptions{"File"};
    fileOptions.add_options()
      ("age", value<int>(), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, generalOptions), vm);
    if (vm.count("config"))
    {
      std::ifstream ifs{vm["config"].as<std::string>().c_str()};
      if (ifs)
        store(parse_config_file(ifs, fileOptions), vm);
    }
    notify(vm);

    if (vm.count("help"))
      std::cout << generalOptions << '\n';
    else if (vm.count("age"))
      std::cout << "Your age is: " << vm["age"].as<int>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

If you create a file called `config.txt`, put age=29 in that file, and execute the command line below, you will get the result shown.

```
test --config config.txt
```

The output is the following:

```
Your age is: 29
```

If you support the same options on the command line and in a configuration file, your program may parse the same option twice – once with `boost::program_options::parse_command_line()` and once with `boost::program_options::parse_config_file()`. The order of the function calls determines which value you will find in **`vm`**. Once a command-line option's value has been stored in **`vm`**, that value will not be overwritten. Whether the value is set by an option on the command line or in a configuration file depends only on the order in which you call the `store()` function.
Boost.ProgramOptions also defines the function `boost::program_options::parse_environment()`, which can be used to load options from environment variables. The class `boost::environment_iterator` lets you iterate over environment variables.

Chapter 64

Boost.Serialization

The library Boost.Serialization makes it possible to convert objects in a C++ program to a sequence of bytes that can be saved and loaded to restore the objects. There are different data formats available to define the rules for generating sequences of bytes. All of the formats supported by Boost.Serialization are only intended for use with this library. For example, the XML format developed for Boost.Serialization should not be used to exchange data with programs that do not use Boost.Serialization. The only advantage of the XML format is that it can make debugging easier since C++ objects are saved in a readable format.

Note

As outlined in the release notes of version 1.55.0 of the Boost libraries, a missing include causes a compiler error with Visual C++ 2013. This bug has been fixed in Boost 1.56.0.

64.1 Archive

The main concept of Boost.Serialization is the *archive*. An archive is a sequence of bytes that represent serialized C++ objects. Objects can be added to an archive to serialize them and then later loaded from the archive. In order to restore previously saved C++ objects, the same types are presumed.

Boost.Serialization provides archive classes such as `boost::archive::text_oarchive`, which is defined in `boost/archive/text_oarchive.hpp`. This class makes it possible to serialize objects as a text stream. With Boost 1.56.0, Example 64.1 writes `22 serialization::archive 11 1` to the standard output stream.

Example 64.1 Using `boost::archive::text_oarchive`

```
#include <boost/archive/text_oarchive.hpp>
#include <iostream>

using namespace boost::archive;

int main()
{
  text_oarchive oa{std::cout};
  int i = 1;
  oa << i;
}
```

As can be seen, the object **oa** of type `boost::archive::text_oarchive` can be used like a stream to serialize a variable using `operator<<`. However, archives should not be considered as regular streams that store arbitrary data. To restore data, you must access it as you stored it, using the same data types in the same order. Example 64.2 serializes and restores a variable of type int.

Example 64.2 Using `boost::archive::text_iarchive`

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
```

```
#include <iostream>
#include <fstream>

using namespace boost::archive;

void save()
{
  std::ofstream file{"archive.txt"};
  text_oarchive oa{file};
  int i = 1;
  oa << i;
}

void load()
{
  std::ifstream file{"archive.txt"};
  text_iarchive ia{file};
  int i = 0;
  ia >> i;
  std::cout << i << '\n';
}

int main()
{
  save();
  load();
}
```

The class `boost::archive::text_oarchive` serializes data as a text stream, and the class `boost::archive::text_iarchive` restores data from such a text stream. To use these classes, include the header files `boost/archive/text_iarchive.hpp` and `boost/archive/text_oarchive.hpp`.
Constructors of archives expect an input or output stream as a parameter. The stream is used to serialize or restore data. While Example 64.2 accesses a file, other streams, such as a stringstream, can also be used.

Example 64.3 Serializing with a stringstream

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

void save()
{
  text_oarchive oa{ss};
  int i = 1;
  oa << i;
}

void load()
{
  text_iarchive ia{ss};
  int i = 0;
  ia >> i;
  std::cout << i << '\n';
}

int main()
{
  save();
  load();
}
```

Example 64.3 writes 1 to standard output using a stringstream to serialize data.
So far, only primitive types have been serialized. Example 64.4 shows how to serialize objects of user-defined types. In order to serialize objects of user-defined types, you must define the member function `serialize()`. This function is called when the object is serialized to or restored from a byte stream. Because `serialize()` is used for both serializing and restoring, Boost.Serialization supports the operator `operator&` in addition to `operator<<` and `operator>>`. With `operator&` there is no need to distinguish between serializing and restoring within `serialize()`.

Example 64.4 Serializing user-defined types with a member function

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  text_oarchive oa{ss};
  animal a{4};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
}

int main()
{
  save();
  load();
}
```

`serialize()` is automatically called any time an object is serialized or restored. It should never be called explicitly and, thus, should be declared as private. If it is declared as private, the class `boost::serialization::access` must be declared as a friend to allow Boost.Serialization to access the member function.

Example 64.5 Serializing with a free-standing function

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
```

```
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

struct animal
{
  int legs_;

  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }
};

template <typename Archive>
void serialize(Archive &ar, animal &a, const unsigned int version)
{
  ar & a.legs_;
}

void save()
{
  text_oarchive oa{ss};
  animal a{4};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
}

int main()
{
  save();
  load();
}
```

There may be situations that do not you allow to modify an existing class in order to add `serialize()`. For example, this is true for classes from the standard library.
In order to serialize types that cannot be modified, the free-standing function `serialize()` can be defined as shown in Example 64.5. This function expects a reference to an object of the corresponding type as its second parameter.
Implementing `serialize()` as a free-standing function requires that essential member variables of a class can be accessed from outside. In Example 64.5, `serialize()` can only be implemented as a free-standing function since **`legs_`** is no longer a private member variable of the class `animal`.
Boost.Serialization provides `serialize()` functions for many classes from the standard library. To serialize objects based on standard classes, additional header files need to be included.

Example 64.6 Serializing strings

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>

using namespace boost::archive;
```

```
std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs, std::string name) :
    legs_{legs}, name_{std::move(name)} {}
  int legs() const { return legs_; }
  const std::string &name() const { return name_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  friend void serialize(Archive &ar, animal &a, const unsigned int version);

  int legs_;
  std::string name_;
};

template <typename Archive>
void serialize(Archive &ar, animal &a, const unsigned int version)
{
  ar & a.legs_;
  ar & a.name_;
}

void save()
{
  text_oarchive oa{ss};
  animal a{4, "cat"};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
  std::cout << a.name() << '\n';
}

int main()
{
  save();
  load();
}
```

Example 64.6 extends the class `animal` by adding **`name_`**, a member variable of type `std::string`. In order to serialize this member variable, the header file `boost/serialization/string.hpp` must be included to provide the appropriate free-standing function `serialize()`.
As mentioned before, Boost.Serialization defines `serialize()` functions for many classes from the standard library. These functions are defined in header files that carry the same name as the corresponding header files from the standard. So, to serialize objects of type `std::string`, include the header file `boost/serialization/string.hpp` and to serialize objects of type `std::vector`, include the header file `boost/serialization/vector.hpp`. It is fairly obvious which header file to include.
One parameter of `serialize()` that has been ignored so far is **`version`**. This parameter helps make archives backward compatible. Example 64.7 can load an archive that was created by Example 64.5. The version of the class `animal` in Example 64.5 did not contain a name. Example 64.7 checks the version number when loading an archive and only accesses the name if the version is greater than 0. This allows it to handle an older archive

that was created without name.
The macro BOOST_CLASS_VERSION assigns a version number to a class. The version number for the class animal in Example 64.7 is 1. If BOOST_CLASS_VERSION is not used, the version number is 0 by default.

Example 64.7 Backward compatibility with version numbers

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/string.hpp>
#include <iostream>
#include <sstream>
#include <string>
#include <utility>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs, std::string name) :
    legs_{legs}, name_{std::move(name)} {}
  int legs() const { return legs_; }
  const std::string &name() const { return name_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  friend void serialize(Archive &ar, animal &a, const unsigned int version);

  int legs_;
  std::string name_;
};

template <typename Archive>
void serialize(Archive &ar, animal &a, const unsigned int version)
{
  ar & a.legs_;
  if (version > 0)
    ar & a.name_;
}

BOOST_CLASS_VERSION(animal, 1)

void save()
{
  text_oarchive oa{ss};
  animal a{4, "cat"};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  ia >> a;
  std::cout << a.legs() << '\n';
  std::cout << a.name() << '\n';
}

int main()
{
  save();
  load();
}
```

The version number is stored in the archive and is part of it. While the version number specified for a particular class via the `BOOST_CLASS_VERSION` macro is used during serialization, the parameter **version** of `serialize()` is set to the value stored in the archive when restoring. If the new version of `animal` accesses an archive containing an object serialized with the old version, the member variable **name_** would not be restored because the old version did not have such a member variable.

64.2 Pointers and References

Boost.Serialization can also serialize pointers and references. Because a pointer stores the address of an object, serializing the address does not make much sense. When serializing pointers and references, the referenced object is serialized.
Example 64.8 creates a new object of type `animal` with `new` and assigns it to the pointer **a**. The pointer – not `*a` – is then serialized. Boost.Serialization automatically serializes the object referenced by **a** and not the address of the object.
If the archive is restored, **a** will not necessarily contain the same address. A new object is created and its address is assigned to **a** instead. Boost.Serialization only guarantees that the object is the same as the one serialized, not that its address is the same.
Because smart pointers are used in connection with dynamically allocated memory, Boost.Serialization provides also support for them.

Example 64.8 Serializing pointers

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  boost::archive::text_oarchive oa{ss};
  animal *a = new animal{4};
  oa << a;
  std::cout << std::hex << a << '\n';
  delete a;
}

void load()
{
  boost::archive::text_iarchive ia{ss};
  animal *a;
  ia >> a;
  std::cout << std::hex << a << '\n';
  std::cout << std::dec << a->legs() << '\n';
  delete a;
}

int main()
{
  save();
  load();
}
```

Example 64.9 Serializing smart pointers

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/scoped_ptr.hpp>
#include <boost/scoped_ptr.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  text_oarchive oa{ss};
  boost::scoped_ptr<animal> a{new animal{4}};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  boost::scoped_ptr<animal> a;
  ia >> a;
  std::cout << a->legs() << '\n';
}

int main()
{
  save();
  load();
}
```

Example 64.9 uses the smart pointer `boost::scoped_ptr` to manage a dynamically allocated object of type `animal`. Include the header file `boost/serialization/scoped_ptr.hpp` to serialize such a pointer.

To serialize a smart pointer of type `boost::shared_ptr`, use the header file `boost/serialization/shared_ptr.hpp`.

Example 64.10 Serializing references

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

void save()
{
  text_oarchive oa{ss};
  animal a{4};
  animal &r = a;
  oa << r;
}

void load()
{
  text_iarchive ia{ss};
  animal a;
  animal &r = a;
  ia >> r;
  std::cout << r.legs() << '\n';
}

int main()
{
  save();
  load();
}
```

Please note that Boost.Serialization hasn't been updated for C++11, yet. Smart pointers from the C++11 standard library like `std::shared_ptr` and `std::unique_ptr` are not currently supported by Boost.Serialization.

Boost.Serialization can also serialize references without any issues (see Example 64.10). Just as with pointers, the referenced object is serialized automatically.

64.3 Serialization of Class Hierarchy Objects

Derived classes must access the function `boost::serialization::base_object()` inside the member function `serialize()` to serialize objects based on class hierarchies. This function guarantees that inherited member variables of base classes are correctly serialized.

Example 64.11 uses a class called `bird`, which is derived from `animal`. Both `animal` and `bird` define a private member function `serialize()` that makes it possible to serialize objects based on either class. Because `bird` is derived from `animal`, `serialize()` must ensure that member variables inherited from `animal` are serialized, too.
Inherited member variables are serialized by accessing the base class inside the member function `serialize()` of the derived class and calling `boost::serialization::base_object()`. You must use this function rather than, for example, `static_cast` because only `boost::serialization::base_object()` ensures correct serialization.

Example 64.11 Serializing derived classes correctly

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;
std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  int legs() const { return legs_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

class bird : public animal
{
public:
  bird() = default;
  bird(int legs, bool can_fly) :
    animal{legs}, can_fly_{can_fly} {}
  bool can_fly() const { return can_fly_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<animal>(*this);
    ar & can_fly_;
  }

  bool can_fly_;
};

void save()
{
  text_oarchive oa{ss};
  bird penguin{2, false};
  oa << penguin;
}

void load()
{
  text_iarchive ia{ss};
  bird penguin;
  ia >> penguin;
  std::cout << penguin.legs() << '\n';
  std::cout << std::boolalpha << penguin.can_fly() << '\n';
}

int main()
{
  save();
  load();
}
```

Addresses of dynamically created objects can be assigned to pointers of the corresponding base class type. Example 64.12 shows that Boost.Serialization can serialize them correctly as well.

Example 64.12 Registering derived classes statically with `BOOST_CLASS_EXPORT`

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/export.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  virtual int legs() const { return legs_; }
  virtual ~animal() = default;

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

class bird : public animal
{
public:
  bird() = default;
  bird(int legs, bool can_fly) :
    animal{legs}, can_fly_{can_fly} {}
  bool can_fly() const { return can_fly_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<animal>(*this);
    ar & can_fly_;
  }

  bool can_fly_;
};
```

```
BOOST_CLASS_EXPORT(bird)

void save()
{
  text_oarchive oa{ss};
  animal *a = new bird{2, false};
  oa << a;
  delete a;
}

void load()
{
  text_iarchive ia{ss};
  animal *a;
  ia >> a;
  std::cout << a->legs() << '\n';
  delete a;
}

int main()
{
  save();
  load();
}
```

The program creates an object of type `bird` inside the function `save()` and assigns it to a pointer of type animal*, which in turn is serialized via `operator<<`.

As mentioned in the previous section, the referenced object is serialized, not the pointer. To have Boost.Serialization recognize that an object of type `bird` must be serialized, even though the pointer is of type animal*, the class `bird` needs to be declared. This is done using the macro `BOOST_CLASS_EXPORT`, which is defined in `boost/serialization/export.hpp`. Because the type `bird` does not appear in the pointer definition, Boost.Serialization cannot serialize an object of type `bird` correctly without the macro.

The macro `BOOST_CLASS_EXPORT` must be used if objects of derived classes are to be serialized using a pointer to their corresponding base class. A disadvantage of `BOOST_CLASS_EXPORT` is that, because of static registration, classes can be registered that may not be used for serialization at all. Boost.Serialization offers a solution for this scenario.

Instead of using the macro `BOOST_CLASS_EXPORT`, Example 64.13 calls the member function template `register_type()`. The type to be registered is passed as a template parameter. Note that `register_type()` must be called both in `save()` and `load()`.

The advantage of `register_type()` is that only classes used for serialization must be registered. For example, when developing a library, one does not know which classes a developer may use for serialization later. While the macro `BOOST_CLASS_EXPORT` makes this easy, it may register types that are not going to be used for serialization.

Example 64.13 Registering derived classes dynamically with `register_type()`

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/export.hpp>
#include <iostream>
#include <sstream>

std::stringstream ss;

class animal
{
public:
  animal() = default;
  animal(int legs) : legs_{legs} {}
  virtual int legs() const { return legs_; }
  virtual ~animal() = default;

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version) { ar & legs_; }

  int legs_;
};

class bird : public animal
{
public:
  bird() = default;
  bird(int legs, bool can_fly) :
    animal{legs}, can_fly_{can_fly} {}
  bool can_fly() const { return can_fly_; }

private:
  friend class boost::serialization::access;

  template <typename Archive>
  void serialize(Archive &ar, const unsigned int version)
  {
    ar & boost::serialization::base_object<animal>(*this);
    ar & can_fly_;
  }

  bool can_fly_;
};

void save()
{
  boost::archive::text_oarchive oa{ss};
  oa.register_type<bird>();
  animal *a = new bird{2, false};
  oa << a;
  delete a;
}

void load()
{
  boost::archive::text_iarchive ia{ss};
  ia.register_type<bird>();
  animal *a;
  ia >> a;
  std::cout << a->legs() << '\n';
  delete a;
}

int main()
{
  save();
  load();
}
```

64.4 Wrapper Functions for Optimization

This section introduces wrapper functions to optimize the serialization process. These functions mark objects to allow Boost.Serialization to apply certain optimization techniques.

Example 64.14 Serializing an array without a wrapper function

```
#include <boost/archive/text_oarchive.hpp>
```

```
#include <boost/archive/text_iarchive.hpp>
#include <boost/array.hpp>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

void save()
{
  text_oarchive oa{ss};
  boost::array<int, 3> a{{0, 1, 2}};
  oa << a;
}

void load()
{
  text_iarchive ia{ss};
  boost::array<int, 3> a;
  ia >> a;
  std::cout << a[0] << ", " << a[1] << ", " << a[2] << '\n';
}

int main()
{
  save();
  load();
}
```

Example 64.14 uses Boost.Serialization without any wrapper function. The example creates and writes the value `22 serialization::archive 11 0 0 3 0 1 2` to the string. Using the wrapper function `boost::serialization::make_array()`, the value written can be shortened to the following string: `22 serialization::archive 11 0 1 2`.

Example 64.15 Serializing an array with the wrapper function `make_array()`

```
#include <boost/archive/text_oarchive.hpp>
#include <boost/archive/text_iarchive.hpp>
#include <boost/serialization/array.hpp>
#include <array>
#include <iostream>
#include <sstream>

using namespace boost::archive;

std::stringstream ss;

void save()
{
  text_oarchive oa{ss};
  std::array<int, 3> a{{0, 1, 2}};
  oa << boost::serialization::make_array(a.data(), a.size());
}

void load()
{
  text_iarchive ia{ss};
  std::array<int, 3> a;
  ia >> boost::serialization::make_array(a.data(), a.size());
  std::cout << a[0] << ", " << a[1] << ", " << a[2] << '\n';
}

int main()
{
  save();
  load();
}
```

boost::serialization::make_array() expects the address and the length of an array. However, because it is known in advance, the length does not need to be serialized as part of the array.

boost::serialization::make_array() can be used whenever classes such as std::array or std::vector contain an array that can be serialized directly. Additional member variables, which would normally also be serialized, are skipped (see Example 64.15).

Boost.Serialization also provides the wrapper boost::serialization::make_binary_object(). Similar to boost::serialization::make_array(), this function expects an address and a length. boost::serialization::make_binary_object() is used solely for binary data that has no underlying structure, while boost::serialization::make_array() is used for arrays.

Chapter 65

Boost.Uuid

Boost.Uuid provides generators for *UUIDs*. UUIDs are universally unique identifiers that don't depend on a central coordinating instance. There is, for example, no database storing all generated UUIDs that can be checked to see whether a new UUID has been used.

UUIDs are used by distributed systems that have to uniquely identify components. For example, Microsoft uses UUIDs to identify interfaces in the COM world. For new interfaces developed for COM, unique identifiers can be easily assigned.

UUIDs are 128-bit numbers. Various methods exist to generate UUIDs. For example, a computer's network address can be used to generate a UUID. The generators provided by Boost.Uuid are based on a random number generator to avoid generating UUIDs that can be traced back to the computer generating them.

All classes and functions from Boost.Uuid are defined in the namespace `boost::uuids`. There is no master header file to get access to all of them.

Example 65.1 generates a random UUID. It uses the class `boost::uuids::random_generator`, which is defined in `boost/uuid/uuid_generators.hpp`. This header file provides access to all generators provided by Boost.Uuid.

`boost::uuids::random_generator` is used like the generators from the C++11 standard library or from Boost.Random. This class overloads `operator()` to generate random UUIDs.

The type of a UUID is `boost::uuids::uuid`. `boost::uuids::uuid` is a *POD* – plain old data. You can't create objects of type `boost::uuids::uuid` without a generator. But then, it's a lean type that allocates exactly 128 bits. The class is defined in `boost/uuid/uuid.hpp`.

An object of type `boost::uuids::uuid` can be written to the standard output stream. However, you must include `boost/uuid/uuid_io.hpp`. This header file provides the overloaded operator to write objects of type `boost::uuids::uuid` to an output stream.

Example 65.1 Generating random UUIDs with `boost::uuids::random_generator`

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();
  std::cout << id << '\n';
}
```

Example 65.1 displays output that looks like the following: `0cb6f61f-be68-5afc-8686-c52e3fc7a50d`. Using dashes is the preferred way of displaying UUIDs.

Example 65.2 Member functions of `boost::uuids::uuid`

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <iostream>
```

```
using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();
  std::cout << id.size() << '\n';
  std::cout << std::boolalpha << id.is_nil() << '\n';
  std::cout << id.variant() << '\n';
  std::cout << id.version() << '\n';
}
```

`boost::uuids::uuid` provides only a few member functions, some of which are introduced in Example 65.2. `size()` returns the size of a UUID in bytes. Because a UUID is always 128 bits, `size()` always returns 16. `is_nil()` returns `true` if the UUID is a nil UUID. The nil UUID is 00000000-0000-0000-0000-000000000000. `variant()` and `version()` specify the kind of UUID and how it was generated. In Example 65.2, `variant()` returns 1, which means the UUID conforms to RFC 4122. `version()` returns 4, which means that the UUID was created by a random number generator.
`boost::uuids::uuid` also provides member functions like `begin()`, `end()`, and `swap()` .

Example 65.3 Generators from Boost.Uuid

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  nil_generator nil_gen;
  uuid id = nil_gen();
  std::cout << std::boolalpha << id.is_nil() << '\n';

  string_generator string_gen;
  id = string_gen("CF77C981-F61B-7817-10FF-D916FCC3EAA4");
  std::cout << id.variant() << '\n';

  name_generator name_gen(id);
  std::cout << name_gen("theboostcpplibraries.com") << '\n';
}
```

Example 65.3 contains more generators from Boost.Uuid. `nil_generator` generates a nil UUID. `is_nil()` returns `true` only if the UUID is nil.
You use `string_generator` if you want to use an existing UUID. You can generate UUIDs at sites such as `http://www.uuidgenerator.net/`. For the UUID in Example 65.3, `variant()` returns 0, which means that the UUID conforms to the backwards compatible NCS standard. `name_generator` is used to generate UUIDs in namespaces.
Please note the spelling of UUIDs when using `string_generator`. You can pass a UUID without dashes, but if you use dashes, they must be in the right places. Case (upper or lower) is ignored.

Example 65.4 Conversion to strings

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <boost/lexical_cast.hpp>
#include <string>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();

  std::string s = to_string(id);
  std::cout << s << '\n';

  std::cout << boost::lexical_cast<std::string>(id) << '\n';
}
```

Boost.Uuid provides the functions `boost::uuids::to_string()` and `boost::uuids::to_wstring()` to convert a UUID to a string (see Example 65.4). It is also possible to use `boost::lexical_cast()` for the conversion.

Part XVI

Design Patterns

The following libraries are for design patterns.

- Boost.Flyweight helps in situations where many identical objects are used in a program and memory consumption needs to be reduced.
- Boost.Signals2 makes it easy to use the observer design pattern. This library is called Boost.Signals2 because it implements the signal/slot concept.
- Boost.MetaStateMachine makes it possible to transfer state machines from UML to C++.

Chapter 66

Boost.Flyweight

Boost.Flyweight is a library that makes it easy to use the design pattern of the same name. Flyweight helps save memory when many objects share data. With this design pattern, instead of storing the same data multiple times in objects, shared data is kept in just one place, and all objects refer to that data. While you can implement this design pattern with, for example, pointers, it is easier to use Boost.Flyweight.

Example 66.1 A hundred thousand identical strings without Boost.Flyweight

```
#include <string>
#include <vector>

struct person
{
  int id_;
  std::string city_;
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Example 66.1 creates a hundred thousand objects of type `person`. `person` defines two member variables: **`id_`** identifies persons, and **`city_`** stores the city people live in. In this example, all people live in Berlin. That's why **`city_`** is set to “Berlin” in all hundred thousand objects. Thus, the example uses a hundred thousand strings all set to the same value. With Boost.Flyweight, one string – instead of thousands – can be used and memory consumption reduced.

Example 66.2 One string instead of a hundred thousand strings with Boost.Flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
```

```
    persons.push_back({i, "Berlin"});
}
```

To use Boost.Flyweight, include `boost/flyweight.hpp`, as in Example 66.2. Boost.Flyweight provides additional header files that only need to be included if you need to change the detailed library settings.

Example 66.3 Using `boost::flyweights::flyweight` multiple times

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  flyweight<std::string> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

All classes and functions are in the namespace `boost::flyweights`. Example 66.2 only uses the class `boost::flyweights::flyweight`, which is the most important class in this library. The member variable **`city_`** uses the type flyweight<std::string> rather than `std::string`. This is all you need to change to use this design pattern and reduce the memory requirements of the program.

Example 66.3 adds a second member variable, **`country_`**, to the class `person`. This member variable contains the names of the countries people live in. Since, in this example, all people live in Berlin, they all live in the same country. That's why `boost::flyweights::flyweight` is used in the definition of the member variable **`country_`**, too.

Boost.Flyweight uses an internal container to store objects. It makes sure there can't be multiple objects with same values. By default, Boost.Flyweight uses a hash container such as `std::unordered_set`. For different types, different hash containers are used. As in Example 66.3, both member variables **`city_`** and **`country_`** are strings; therefore, only one container is used. In this example, this is not a problem because the container only stores two strings: "Berlin" and "Germany." If many different cities and countries must be stored, it would be better to store cities in one container and countries in another.

Example 66.4 Using `boost::flyweights::flyweight` multiple times with tags

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct city {};
struct country {};

struct person
{
  int id_;
  flyweight<std::string, tag<city>> city_;
  flyweight<std::string, tag<country>> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Example 66.4 passes a second template parameter to `boost::flyweights::flyweight`. This is a *tag*. Tags are arbitrary types only used to differentiate the types on which **`city_`** and **`country_`** are based. Example 66.4 defines two empty structures `city` and `country`, which are used as tags. However, the example could have instead used int, bool, or any type.
The tags make **`city_`** and **`country_`** use different types. Now two hash containers are used by Boost.Flyweight – one stores cities, the other stores countries.

Example 66.5 Template parameters of `boost::flyweights::flyweight`

```
#include <boost/flyweight.hpp>
#include <boost/flyweight/set_factory.hpp>
#include <boost/flyweight/no_locking.hpp>
#include <boost/flyweight/no_tracking.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string, set_factory<>, no_locking, no_tracking> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Template parameters other than tags can be passed to `boost::flyweights::flyweight`. Example 66.5 passes `boost::flyweights::set_factory`, `boost::flyweights::no_locking`, and `boost::flyweights::no_tracking`. Additional header files are included to make use of these classes.
`boost::flyweights::set_factory` tells Boost.Flyweight to use a sorted container, such as `std::set`, rather than a hash container. With `boost::flyweights::no_locking`, support for multithreading, which is normally activated by default, is deactivated. `boost::flyweights::no_tracking` tells Boost.Flyweight to not track objects stored in internal containers. By default, when objects are no longer used, Boost.Flyweight detects this and removes them from the containers. When `boost::flyweights::no_tracking` is set, the detection mechanism is disabled. This improves performance. However, containers can only grow and will never shrink.
Boost.Flyweight supports additional settings. Check the official documentation if you are interested in more details on tuning.

Chapter 67

Boost.Signals2

Boost.Signals2 implements the signal/slot concept. One or multiple functions – called *slots* – are linked with an object that can emit a signal. Every time the signal is emitted, the linked functions are called.
The signal/slot concept can be useful when, for example, developing applications with graphical user interfaces. Buttons can be modelled so they emit a signal when a user clicks on them. They can support links to many functions to handle user input. That way it is possible to process events flexibly.
`std::function` can also be used for event handling. One crucial difference between `std::function` and Boost.Signals2, is that Boost.Signals2 can associate more than one event handler with a single event. Therefore, Boost.Signals2 is better for supporting event-driven development and should be the first choice whenever events need to be handled.
Boost.Signals2 succeeds the library Boost.Signals, which is deprecated and not discussed in this book.

67.1 Signals

Boost.Signals2 offers the class `boost::signals2::signal`, which can be used to create a signal. This class is defined in `boost/signals2/signal.hpp`. Alternatively, you can use the header file `boost/signals2.hpp`, which is a master header file that defines all of the classes and functions available in Boost.Signals2.
Boost.Signals2 defines `boost::signals2::signal` and other classes, as well as all functions in the namespace `boost::signals2`.

Example 67.1 "Hello, world!" with `boost::signals2::signal`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
}
```

`boost::signals2::signal` is a class template that expects as a template parameter the signature of the function that will be used as an event handler. In Example 67.1, only functions with a signature of void() can be associated with the signal **`s`**.
A lambda function is associated with the signal **`s`** through `connect()`. Because the lambda function conforms to the required signature, void(), the association is successfully established. The lambda function is called whenever the signal **`s`** is triggered.
The signal is triggered by calling **`s`** like a regular function. The signature of this function matches the one passed as the template parameter. The brackets are empty because void() does not expect any parameters. Calling **`s`** results in a trigger that, in turn, executes the lambda function that was previously associated with `connect()`.
Example 67.1 can also be implemented with `std::function`, as shown in Example 67.2.

Example 67.2 “Hello, world!” with `std::function`

```
#include <functional>
#include <iostream>

int main()
{
  std::function<void()> f;
  f = []{ std::cout << "Hello, world!\n"; };
  f();
}
```

In Example 67.2, the lambda function is also executed when `f` is called. While `std::function` can only be used in a scenario like Example 67.2, Boost.Signals2 provides far more variety. For example, it can associate multiple functions with a particular signal (see Example 67.3).

Example 67.3 Multiple event handlers with `boost::signals2::signal`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello"; });
  s.connect([]{ std::cout << ", world!\n"; });
  s();
}
```

`boost::signals2::signal` allows you to assign multiple functions to a particular signal by calling `conn ect()` repeatedly. Whenever the signal is triggered, the functions are executed in the order in which they were associated with `connect()`.
The order can also be explicitly defined with the help of an overloaded version of `connect()`, which expects a value of type int as an additional parameter (Example 67.4).
Like the previous example, Example 67.4 displays `Hello, world!`.

Example 67.4 Event handlers with an explicit order

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect(1, []{ std::cout << ", world!\n"; });
  s.connect(0, []{ std::cout << "Hello"; });
  s();
}
```

To release an associated function from a signal, call `disconnect()`. Example 67.5 only prints `Hello` because the association with `world()` was released before the signal was triggered.

Example 67.5 Disconnecting event handlers from `boost::signals2::signal`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

void hello() { std::cout << "Hello"; }
void world() { std::cout << ", world!\n"; }
```

```
int main()
{
  signal<void()> s;
  s.connect(hello);
  s.connect(world);
  s.disconnect(world);
  s();
}
```

Besides `connect()` and `disconnect()`, `boost::signals2::signal` provides a few more member functions (see Example 67.6). `num_slots()` returns the number of associated functions. If no function is associated, `num_slots()` returns 0. `empty()` tells you whether event handlers are connected or not. And `disconnect_all_slots()` does exactly what its name says: it releases all existing associations.

Example 67.6 Additional member functions of `boost::signals2::signal`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  s.connect([]{ std::cout << "Hello"; });
  s.connect([]{ std::cout << ", world!"; });
  std::cout << s.num_slots() << '\n';
  if (!s.empty())
    s();
  s.disconnect_all_slots();
}
```

In Example 67.7 two lambda functions are associated with the signal **`s`**. The first lambda function returns 1, the second returns 2.
Example 67.7 writes `2` to standard output. Both return values were correctly accepted by **`s`**, but all except the last one were ignored. By default, only the last return value of all associated functions is returned.

Example 67.7 Processing return values from event handlers

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<int()> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::cout << *s() << '\n';
}
```

Please note that `s()` does not directly return the result of the last function called. An object of type `boost::optional` is returned, which when de-referenced returns the number 2. Triggering a signal that is not associated with any function does not yield any return value. Thus, in this case, `boost::optional` allows Boost.Signals2 to return an empty object. `boost::optional` is introduced in Chapter 21.
It is possible to customize a signal so that the individual return values are processed accordingly. To do this, a *combiner* must be passed to `boost::signals2::signal` as a second template parameter.
A combiner is a class with an overloaded `operator()`. This operator is automatically called with two iterators, which are used to access the functions associated with the particular signal. When the iterators are de-referenced, the functions are called and their return values become available in the combiner. A common algorithm from the standard library, such as `std::min_element()`, can then be used to calculate and return the smallest value (see Example 67.8).

`boost::signals2::signal` uses `boost::signals2::optional_last_value` as the default combiner. This combiner returns objects of type `boost::optional`. A user can define a combiner with a return value of any type. For instance, the combiner `min_element` in Example 67.8 returns the type passed as a template parameter to `min_element`.
It is not possible to pass an algorithm such as `std::min_element()` as a template parameter directly to `boost::signals2::signal`. `boost::signals2::signal` expects that the combiner defines a type called result_type, which denotes the type of the value returned by `operator()`. Since this type is not defined by standard algorithms, the compiler will report an error.

Example 67.8 Finding the smallest return value with a user-defined combiner

```
#include <boost/signals2.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::signals2;

template <typename T>
struct min_element
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    std::vector<T> v(first, last);
    return *std::min_element(v.begin(), v.end());
  }
};

int main()
{
  signal<int(), min_element<int>> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::cout << s() << '\n';
}
```

Please note that it is not possible to pass the iterators **`first`** and **`last`** directly to `std::min_element()` because this algorithm expects forward iterators, while combiners work with input iterators. That's why a vector is used to store all return values before determining the smallest value with `std::min_element()`.
Example 67.9 modifies the combiner to store all return values in a container, rather than evaluating them. It stores all the return values in a vector that is then returned by `s()`.

Example 67.9 Receiving all return values with a user-defined combiner

```
#include <boost/signals2.hpp>
#include <vector>
#include <algorithm>
#include <iostream>

using namespace boost::signals2;

template <typename T>
struct min_element
{
  typedef T result_type;

  template <typename InputIterator>
  T operator()(InputIterator first, InputIterator last) const
  {
    return T(first, last);
  }
};
```

```
int main()
{
  signal<int(), min_element<std::vector<int>>> s;
  s.connect([]{ return 1; });
  s.connect([]{ return 2; });
  std::vector<int> v = s();
  std::cout << *std::min_element(v.begin(), v.end()) << '\n';
}
```

67.2 Connections

Functions can be managed with the aid of the `connect()` and `disconnect()` member functions provided by `boost::signals2::signal`. Because `connect()` returns an object of type `boost::signals2::connection`, associations can also be managed differently (see Example 67.10).

Example 67.10 Managing connections with `boost::signals2::connection`

```
#include <boost/signals2.hpp>
#include <iostream>

int main()
{
  boost::signals2::signal<void()> s;
  boost::signals2::connection c = s.connect(
    []{ std::cout << "Hello, world!\n"; });
  s();
  c.disconnect();
}
```

The `disconnect()` member function of `boost::signals2::signal` requires a function pointer to be passed in. Avoid this by calling `disconnect()` on the `boost::signals2::connection` object.
To block a function for a short time without removing the association from the signal, `boost::signals2::shared_connection_block` can be used.

Example 67.11 Blocking connections with `shared_connection_block`

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

int main()
{
  signal<void()> s;
  connection c = s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
  shared_connection_block b{c};
  s();
  b.unblock();
  s();
}
```

Example 67.11 executes the lambda function twice. The signal **s** is triggered three times, but the lambda function is not called the second time because an object of type `boost::signals2::shared_connection_block` was created to block the call. Once the object goes out of scope, the block is automatically removed. A block can also be removed explicitly by calling `unblock()`. Because it is called before the last trigger, the final call to the lambda function is executed again.

Example 67.12 Blocking a connection multiple times

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;
```

```
int main()
{
  signal<void()> s;
  connection c = s.connect([]{ std::cout << "Hello, world!\n"; });
  shared_connection_block b1{c, false};
  {
    shared_connection_block b2{c};
    std::cout << std::boolalpha << b1.blocking() << '\n';
    s();
  }
  s();
}
```

In addition to `unblock()`, `boost::signals2::shared_connection_block` provides the member functions `block()` and `blocking()`. The former is used to block a connection after a call to `unblock()`, while the latter makes it possible to check whether or not a connection is currently blocked.
Please note that `boost::signals2::shared_connection_block` carries the word "shared" for a reason: multiple objects of type `boost::signals2::shared_connection_block` can be initialized with the same connection.
Example 67.12 accesses **`s`** twice, but the lambda function is only called the second time. The program writes `Hello, world!` to the standard output stream only once.
Because `false` is passed to the constructor as the second parameter, the first object of type `boost::signals2::shared_connection_block` does not block the connection to the signal **`s`**. Hence, calling `blocking()` on the object **`b1`** returns `false`.
Nevertheless, the lambda function is not executed when **`s`** is first accessed because the access happens only after a second object of type `boost::signals2::shared_connection_block` has been instantiated. By not passing a second parameter to the constructor, the connection is blocked by the object. When **`s`** is accessed for the second time, the lambda function is executed because the block was automatically removed once **`b2`** went out of scope.

Example 67.13 Member functions as event handlers

```
#include <boost/signals2.hpp>
#include <memory>
#include <functional>
#include <iostream>

using namespace boost::signals2;

struct world
{
  void hello() const
  {
    std::cout << "Hello, world!\n";
  }
};

int main()
{
  signal<void()> s;
  {
    std::unique_ptr<world> w(new world());
    s.connect(std::bind(&world::hello, w.get()));
  }
  std::cout << s.num_slots() << '\n';
  s();
}
```

Boost.Signals2 can release a connection once the object whose member function is associated with a signal is destroyed.
Example 67.13 associates the member function of an object with a signal, with the help of `std::bind()`. The object is destroyed before the signal is triggered, which is a problem because, instead of passing an object of type

world, only an address was passed to `std::bind()`. By the time `s()` is called, the object referenced no longer exists.
It is possible to modify the program so that the connection is automatically released once the object is destroyed. Example 67.14 does this.

Example 67.14 Releasing associated member functions automatically

```
#include <boost/signals2.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::signals2;

struct world
{
  void hello() const
  {
    std::cout << "Hello, world!\n";
  }
};

int main()
{
  signal<void()> s;
  {
    boost::shared_ptr<world> w(new world());
    s.connect(signal<void()>::slot_type(&world::hello, w.get()).track(w));
  }
  std::cout << s.num_slots() << '\n';
  s();
}
```

Now `num_slots()` returns `0`. Example 67.14 does not try to call a member function on an object that doesn't exist when the signal is triggered. The change was to tie the object of type `world` to a smart pointer of type `boost::shared_ptr`, which is passed to `track()`. This member function is called on the slot that was passed to `connect()` to request tracking on the corresponding object.
A function or member function associated with a signal is called a *slot*. The type to specify a slot was not used in the previous examples because passing a pointer to a function or member function to `connect()` was sufficient. The corresponding slot was created and associated with the signal automatically.
In Example 67.14, however, the smart pointer is associated with the slot by calling `track()`. Because the type of the slot depends on the signal, `boost::signals2::signal` provides a type slot_type to access the required type. slot_type behaves just like `std::bind`, making it possible to pass both parameters to describe the slot directly. `track()` can then be called to associate the slot with a smart pointer of type `boost::shared_ptr`. The object is then tracked, which causes the slot to be automatically removed once the tracked object is destroyed.
To manage objects with different smart pointers, slots provide a member function called `track_foreign()`. While `track()` expects a smart pointer of type `boost::shared_ptr`, `track_foreign()` allows you to, for example, use a smart pointer of type `std::shared_ptr`. Smart pointers other than `std::shared_ptr` and `std::weak_ptr` must be introduced to Boost.Signals2 before they can be passed to `track_foreign()`.
The consumer of a particular event can access an object of type `boost::signals2::signal` to create new associations or release existing ones.

Example 67.15 Creating new connections in an event handler

```
#include <boost/signals2.hpp>
#include <iostream>

boost::signals2::signal<void()> s;

void connect()
{
  s.connect([]{ std::cout << "Hello, world!\n"; });
}

int main()
{
  s.connect(connect);
  s();
}
```

Example 67.15 accesses **s** inside the `connect()` function to associate a lambda function with the signal. Since `connect()` is called when the signal is triggered, the question is whether the lambda function will also be called. The program does not output anything, which means the lambda function is never called. While Boost.Signals2 supports associating functions to signals when a signal is triggered, the new associations will only be used when the signal is triggered again.

Example 67.16 Releasing connections in an event handler

```
#include <boost/signals2.hpp>
#include <iostream>

boost::signals2::signal<void()> s;

void hello()
{
  std::cout << "Hello, world!\n";
}

void disconnect()
{
  s.disconnect(hello);
}

int main()
{
  s.connect(disconnect);
  s.connect(hello);
  s();
}
```

Example 67.16 does not create a new association, instead it releases an existing one. As in Example 67.15, this example writes nothing to the standard output stream.

This behavior can be explained quite simply. Imagine that a temporary copy of all slots is created whenever a signal is triggered. Newly created associations are not added to the temporary copy and therefore can only be called the next time the signal is triggered. Released associations, on the other hand, are still part of the temporary copy, but will be checked by the combiner when de-referenced to avoid calling a member function on an object that has already been destroyed.

67.3 Multithreading

Almost all classes provided by Boost.Signals2 are thread safe and can be used in multithreaded applications. For example, objects of type `boost::signals2::signal` and `boost::signals2::connection` can be accessed from different threads.

On the other hand, `boost::signals2::shared_connection_block` is not thread safe. This limitation is not important because multiple objects of type `boost::signals2::shared_connection_block` can be created in different threads and can use the same connection object.

Example 67.17 `boost::signals2::signal` is thread safe

```
#include <boost/signals2.hpp>
#include <thread>
#include <mutex>
#include <iostream>

boost::signals2::signal<void(int)> s;
std::mutex m;

void loop()
{
  for (int i = 0; i < 100; ++i)
    s(i);
}

int main()
{
  s.connect([](int i){
    std::lock_guard<std::mutex> lock{m};
    std::cout << i << '\n';
  });
  std::thread t1{loop};
  std::thread t2{loop};
  t1.join();
  t2.join();
}
```

Example 67.17 creates two threads that execute the `loop()` function, which accesses **`s`** one hundred times to call the associated lambda function. Boost.Signals2 explicitly supports simultaneous access from different threads to objects of type `boost::signals2::signal`.
Example 67.17 displays numbers from 0 to 99. Because **`i`** is incremented in two threads and written to the standard output stream in the lambda function, the numbers will not only be displayed twice, they will also overlap. However, because `boost::signals2::signal` can be accessed from different threads, the program will not crash.
However, Example 67.17 still requires synchronization. Because two threads access **`s`**, the associated lambda function runs in parallel in two threads. To avoid having the two threads interrupt each other while writing to standard output, a mutex is used to synchronize access to **`std::cout`**.
For single-threaded applications, support for multithreading can be disabled in Boost.Signals2.

Example 67.18 `boost::signals2::signal` without thread safety

```
#include <boost/signals2.hpp>
#include <iostream>

using namespace boost::signals2;

signal<void()> s;

int main()
{
  typedef keywords::mutex_type<dummy_mutex> dummy_mutex;
  signal_type<void(), dummy_mutex>::type s;
  s.connect([]{ std::cout << "Hello, world!\n"; });
  s();
}
```

Out of the many template parameters supported by `boost::signals2::signal`, the last one defines the type of mutex used for synchronization. Fortunately, Boost.Signals2 offers a simpler way to disable synchronization than passing the complete list of parameters.
The `boost::signals2::keywords` namespace contains classes that make it possible to pass template parameters by name. `boost::signals2::keywords::mutex_type` can be used to pass the mutex type as the second template parameter to `boost::signals2::signal_type`. Please note that `boost::signals2::signal_type`, not `boost::signals2::signal`, must be used in this case. The type equivalent to `boost::signals2::signal`, which is required to define the signal **`s`**, is retrieved via `boost::signals2::signal_type::type`.
Boost.Signals2 provides an empty mutex implementation called `boost::signals2::dummy_mutex`. If a signal is defined with this class, it will no longer support multithreading (see Example 67.18).

Chapter 68

Boost.MetaStateMachine

Boost.MetaStateMachine is used to define state machines. State machines describe objects through their states. They describe what states exist and what transitions between states are possible.
Boost.MetaStateMachine provides three different ways to define state machines. The code you need to write to create a state machine depends on the front-end.
If you go with the basic front-end or the function front-end, you define state machines in the conventional way: you create classes, derive them from other classes provided by Boost.MetaStateMachine, define required member variables, and write the required C++ code yourself. The fundamental difference between the basic front-end and the function front-end is that the basic front-end expects function pointers, while the function front-end lets you use function objects.
The third front-end is called eUML and is based on a domain-specific language. This front-end makes it possible to define state machines by reusing definitions of a UML state machine. Developers familiar with UML can copy definitions from a UML behavior diagram to C++ code. You don't need to translate UML definitions to C++ code.
eUML is based on a set of macros that you must use with this front-end. The advantage of the macros is that you don't need to work directly with many of the classes provided by Boost.MetaStateMachine. You just need to know which macros to use. This means you can't forget to derive your state machine from a class, which can happen with the basic front-end or the function front-end. This chapter introduces Boost.MetaStateMachine with eUML.
Example 68.1 uses the simplest state machine possible: A lamp has exactly two states. It is either on or off. If it is off, it can be switched on. If it is on, it can be switched off. It is possible to switch from every state to every other state.

Example 68.1 Simple state machine with eUML

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
```

```
{
  msm::back::state_machine<light_state_machine> light;
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
  std::cout << *light.current_state() << '\n';
}
```

Example 68.1 uses the eUML front-end to describe the state machine of a lamp. Boost.MetaStateMachine doesn't have a master header file. Therefore, the required header files have to be included one by one. `boost/msm/front/euml/euml.hpp` and `boost/msm/front/euml/state_grammar.hpp` provide access to eUML macros. `boost/msm/back/state_machine.hpp` is required to link a state machine from the front-end to a state-machine from the back-end. While front-ends provide various possibilities to define state machines, the actual implementation of a state machine is found in the back-end. Since Boost.MetaStateMachine contains only one back-end, you don't need to select an implementation.

All of the definitions from Boost.MetaStateMachine are in the namespace `boost::msm`. Unfortunately, many eUML macros don't refer explicitly to classes in this namespace. They use either the namespace `msm` or no namespace at all. That's why Example 68.1 creates an alias for the namespace `boost::msm` and makes the definitions in `boost::msm::front::euml` available with a `using` directive. Otherwise the eUML macros lead to compiler errors.

To use the state machine of a lamp, first define the states for off and on. States are defined with the macro `BOOST_MSM_EUML_STATE`, which expects the name of the state as its second parameter. The first parameter describes the state. You'll see later how these descriptions look like. The two states defined in Example 68.1 are called **`Off`** and **`On`**.

To switch between states, events are required. Events are defined with the macro `BOOST_MSM_EUML_EVENT`, which expects the name of the event as its sole parameter. Example 68.1 defines an event called **`press`**, which represents the action of pressing the light switch. Since the same event switches a light on and off, only one event is defined.

When the required states and events are defined, the macro `BOOST_MSM_EUML_TRANSITION_TABLE` is used to create a *transition table*. This tables defines valid transitions between states and which events trigger which state transitions.

`BOOST_MSM_EUML_TRANSITION_TABLE` expects two parameters. The first parameter defines the transition table, and the second is the name of the transition table. The syntax of the first parameter is designed to make it easy to recognize how states and events relate to each other. For example, `Off + press ==On` means that the machine in the state **`Off`** switches to the state **`On`** with the event **`press`**. The intuitive and self-explanatory syntax of a transition table definition is one of the strengths of the eUML front-end.

After the transition table has been created, the state machine is defined with the macro `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`. The second parameter is again the simpler one: it sets the name of the state machine. The state machine in Example 68.1 is named `light_state_machine`.

The first parameter of `BOOST_MSM_EUML_DECLARE_STATE_MACHINE` is a tuple. The first value is the name of the transition table. The second value is an expression using **`init_`**, which is an attribute provided by Boost.MetaStateMachine. You'll learn more about attributes later. The expression `init_ << Off` is required to set the initial state of the state machine to **`Off`**.

The state machine `light_state_machine`, defined with `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`, is a class. You use this class to instantiate a state machine from the back-end. In Example 68.1 this is done by passing `light_state_machine` to the class template `boost::msm::back::state_machine` as a parameter. This creates a state machine called **`light`**.

State machines provide a member function `process_event()` to process events. If you pass an event to `process_event()`, the state machines changes its state depending on its transition table.

To make it easier to see what happens in Example 68.1 when `process_event()` is called multiple times, `current_state()` is called. This member function should only be used for debugging purposes. It returns a pointer to an int. Every state is an int value assigned in the order the states have been accessed in `BOOST_MSM_EUML_TRANSITION_TABLE`. In Example 68.1 **`Off`** is assigned the value 0 and **`On`** is assigned the value 1. The example writes `0`, `1`, and `0` to standard output. The light switch is pressed two times, which switches the light on and off.

Example 68.2 State machine with state, event, and action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
```

```
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    BOOST_MSM_EUML_STATE_NAME(Off) &sourceState,
    BOOST_MSM_EUML_STATE_NAME(On) &targetState) const
  {
    std::cout << "Switching on\n";
  }

  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    decltype(On) &sourceState,
    decltype(Off) &targetState) const
  {
    std::cout << "Switching off\n";
  }
};

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.2 extends the state machine for the lamp by an action. An action is executed by an event triggering a state transition. Because actions are optional, a state machine could be defined without them.
Actions are defined with `BOOST_MSM_EUML_ACTION`. Strictly speaking, a function object is defined. You must overload the operator `operator()`. The operator must accept four parameters. The parameters reference an event, a state machine and two states. You are free to define a template or use concrete types for all of the parameters. In Example 68.2, concrete types are only set for the last two parameters. Because these parameters describe the beginning and ending states, you can overload `operator()` so that different member functions are executed for different switches.
Please note that the states **`On`** and **`Off`** are objects. Boost.MetaStateMachine provides a macro `BOOST_MSM_EUML_STATE_NAME` to get the type of a state. If you use C++11, you can use the operator `decltype` instead of the macro.
The action **`switch_light`**, which has been defined with `BOOST_MSM_EUML_ACTION`, is executed when the light switch is pressed. The transition table has been changed accordingly. The first transition is now `Off + press /switch_light ==On`. You pass actions after a slash after the event. This transition means that the operator `operator()` of **`switch_light`** is called if the current state is **`Off`** and the event **`press`** happens. After the action has been executed, the new state is **`On`**.
Example 68.2 writes `Switching on` and then `Switching off` to standard output.

Example 68.3 uses a guard in the transition table. The definition of the first transition is `Off + press [!is_broken] /switch_light ==On`. Passing **`is_broken`** in brackets means that the state machine checks before the action **`switch_light`** is called whether the transition may occur. This is called a guard. A guard must return a result of type bool.
A guard like **`is_broken`** is defined with `BOOST_MSM_EUML_ACTION` in the same way as actions. Thus, the operator `operator()` has to be overloaded for the same four parameters. `operator()` must have a return value of type bool to be used as a guard.
Please note that you can use logical operators like `operator!` on guards inside brackets.
If you run the example, you'll notice that nothing is written to standard output. The action **`switch_light`** is not executed – the light stays off. The guard **`is_broken`** returns `true`. However, because the operator `operator!` is used, the expression in brackets evaluates to `false`.
You can use guards to check whether a state transition can occur. Example 68.3 uses **`is_broken`** to check whether the lamp is broken. While a transition from off to on is usually possible and the transition table describes lamps correctly, in this example, the lamp cannot be switched on. Despite two calls to `process_event()`, the state of **`light`** is **`Off`**.

Example 68.3 State machine with state, event, guard, and action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return true;
  }
};

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm, class Source, class Target>
  void operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    std::cout << "Switching\n";
  }
};

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.4 State machine with state, event, entry action, and exit action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Entering\n";
  }
};

BOOST_MSM_EUML_ACTION(state_exit)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Exiting\n";
  }
};

BOOST_MSM_EUML_STATE((state_entry, state_exit), Off)
BOOST_MSM_EUML_STATE((state_entry, state_exit), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

In Example 68.4, the first parameter passed to `BOOST_MSM_EUML_STATE` is a tuple consisting of `state_entry` and `state_exit`. `state_entry` is an entry action, and `state_exit` is an exit action. These actions are executed when a state is entered or exited.

Like actions, entry and exit actions are defined with `BOOST_MSM_EUML_ACTION`. However, the overloaded operator `operator()` expects only three parameters: references to an event, a state machine, and a state. Transitions between states don't matter for entry and exit actions, so only one state needs to be passed to `operator()`. For entry actions, this state is entered. For exit actions, this state is exited.

In Example 68.4, both states **`Off`** and **`On`** have entry and exit actions. Because the event **`press`** occurs twice, `Entering` and `Exiting` is displayed twice. Please note that `Exiting` is displayed first and `Entering` afterwards because the first action executed is an exit action.

The first event **`press`** triggers a transition from **`Off`** to **`On`**, and `Exiting` and `Entering` are each displayed once. The second event **`press`** switches the state to **`Off`**. Again `Exiting` and `Entering` are each displayed once. Thus, state transitions execute the exit action first, then the entry action of the new state.

Example 68.5 uses the guard **is_broken** to check whether a state transition from **Off** to **On** is possible. This time the return value of **is_broken** depends on how often the light switch has been pressed. It is possible to switch the light on two times before the lamp is broken. To count how often the light has been switched on, an attribute is used.
Attributes are variables that can be attached to objects. They let you adapt the behavior of state machines at run time. Because data such as how often the light has been switched on has to be stored somewhere, it makes sense to store it directly in the state machine, in a state, or in an event.
Before an attribute can be used, it has to be defined. This is done with the macro BOOST_MSM_EUML_DECLARE_ATTRIBUTE. The first parameter passed to BOOST_MSM_EUML_DECLARE_ATTRIBUTE is the type, and the second is the name of the attribute. Example 68.5 defines the attribute **switched_on** of type int.
After the attribute has been defined, it must be attached to an object. The example attaches the attribute **switched_on** to the state machine. This is done via the fifth value in the tuple, which is passed as the first parameter to BOOST_MSM_EUML_DECLARE_STATE_MACHINE. With **attributes_**, a keyword from Boost.MetaStateMachine is used to create a lambda function. To attach the attribute **switched_on** to the state machine, write **switched_on** to **attributes_** as though it were a stream, using operator<<.
The third and fourth values in the tuples are both set to **no_action**. The attribute is passed as the fifth value in the tuple. The third and fourth values can be used to define entry and exit actions for the state machine. If no entry and exit actions are defined, use **no_action**.
After the attribute has been attached to the state machine, it can be accessed with get_attribute(). In Example 68.5, this member function is called in the entry action state_entry to increment the value of the attribute. Because state_entry is only linked to the state **On**, **switched_on** is only incremented when the light is switched on.
switched_on is also accessed from the guard **is_broken**, which checks whether the value of the attribute is greater than 1. If it is, the guard returns true. Because attributes are initialized with the default constructor and **switched_on** is set to 0, **is_broken** returns true if the light has been switched on two times.

Example 68.5 Attributes in a state machine

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Switched on\n";
    ++fsm.get_attribute(switched_on);
  }
};
BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return fsm.get_attribute(switched_on) > 1;
  }
};

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((state_entry), On)
BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

In Example 68.5, the event **press** occurs five times. The light is switched on and off two times and then switched on again. The first two times the light is switched on, `Switched on` is displayed. However, the third time the light is switched on there is no output. This happens because **is_broken** returns `true` after the light has been switched on two times, and therefore, there is no state transition from **Off** to **On**. This means the entry action for the state **On** is not executed, and the example does not write to standard output.

Example 68.6 does the same thing as Example 68.5: after switching the light on two times, the light is broken and can't be switched on anymore. While the previous example accessed the attribute **switched_on** in actions, this example uses attributes in the transition table.

Boost.MetaStateMachine provides the function `fsm_()` to access an attribute in a state machine. That way a guard is defined that checks whether **switched_on** is smaller than 2. And an action is defined that increments **switched_on** every time the state switches from **Off** to **On**.

Please note that the smaller-than comparison in the guard is done with `Int_<2>()`. The number 2 must be passed as a template parameter to `Int_` to create an instance of this class. That creates a function object that has the type needed by Boost.MetaStateMachine.

Example 68.6 also uses the macro `BOOST_MSM_EUML_FUNCTION` to make a function an action. The first parameter passed to `BOOST_MSM_EUML_FUNCTION` is the name of the action that can be used in the function front-end. The second parameter is the name of the function. The third parameter is the name of the action as it is used in eUML. The fourth and fifth parameters are the return values for the function – one for the case where the action is used for a state transition, and the other for the case where the action describes an entry or exit action. After `write_message()` has been turned into an action this way, an object of type `write_message_` is created and used following `++fsm_(switched_on)` in the transition table. In a state transition from **Off** to **On**, the attribute **switched_on** is incremented and then `write_message()` is called.

Example 68.6 displays `Switched on` twice, as in Example 68.5.

Boost.MetaStateMachine provides additional functions, such as `state_()` and `event_()`, to access attributes attached to other objects. Other classes, such as `Char_` and `String_`, can also be used like `Int_`.

Tip

As you can see in the examples, the front-end eUML requires you to use many macros. The header file `boost/msm/front/euml/common.hpp` contains definitions for all of the eUML macros, which makes it a useful reference.

Example 68.6 Accessing attributes in transition tables

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)
```

```
void write_message()
{
  std::cout << "Switched on\n";
}

BOOST_MSM_EUML_FUNCTION(WriteMessage_, write_message, write_message_,
  void, void)

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [fsm_(switched_on) < Int_<2>()] / (++fsm_(switched_on),
    write_message_()) == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Part XVII

Other Libraries

The following libraries provide small but helpful utilities.

- Boost.Utility collects everything that doesn’t fit somewhere else in the Boost libraries.
- Boost.Assign provides helper functions that make it easier to perform operations such as adding multiple values to a container without having to call `push_back()` repeatedly.
- Boost.Swap provides a variant of `std::swap()` that is optimized for the Boost libraries.
- Boost.Operators makes it easy to define operators based on other operators.

Chapter 69

Boost.Utility

The library Boost.Utility is a conglomeration of miscellaneous, useful classes and functions that are too small to justify being maintained in stand-alone libraries. While the utilities are small and can be learned quickly, they are completely unrelated. Unlike the examples in other chapters, the code samples here do not build on each other, since they are independent utilities.

While most utilities are defined in `boost/utility.hpp`, some have their own header files. The following examples include the appropriate header file for the utility being introduced.

Example 69.1 passes the function `boost::checked_delete()` as a parameter to the member function `pop_back_and_dispose()`, which is provided by the class `boost::intrusive::list` from Boost.Intrusive. `boost::intrusive::list` and `pop_back_and_dispose()` are introduced in Chapter 18, while `boost::checked_delete()` is provided by Boost.Utility and defined in `boost/checked_delete.hpp`.

`boost::checked_delete()` expects as its sole parameter a pointer to the object that will be deleted by `delete`. Because `pop_back_and_dispose()` expects a function that takes a pointer to destroy the corresponding object, it makes sense to pass in `boost::checked_delete()` – that way, you don't need to define a similar function.

Unlike `delete`, `boost::checked_delete()` ensures that the type of the object to be destroyed is complete. `delete` will accept a pointer to an object with an incomplete type. While this concerns a detail of the C++ standard that you can usually ignore, you should note that `boost::checked_delete()` is not completely identical to a call to `delete` because it puts higher demands on its parameter.

Example 69.1 Using `boost::checked_delete()`

```
#include <boost/checked_delete.hpp>
#include <boost/intrusive/list.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : public boost::intrusive::list_base_hook<>
{
  std::string name_;
  int legs_;

  animal(std::string name, int legs) : name_{std::move(name)},
    legs_{legs} {}
};

int main()
{
  animal *a = new animal{"cat", 4};

  typedef boost::intrusive::list<animal> animal_list;
  animal_list al;

  al.push_back(*a);

  al.pop_back_and_dispose(boost::checked_delete<animal>);
  std::cout << al.size() << '\n';
}
```

Boost.Utility also provides `boost::checked_array_delete()`, which can be used to destroy arrays. It calls `delete[]` rather than `delete`.
Additionally, two classes, `boost::checked_deleter` and `boost::checked_array_deleter`, are available to create function objects that behave like `boost::checked_delete()` and `boost::checked_array_delete()`, respectively.

Example 69.2 Using `BOOST_CURRENT_FUNCTION`

```
#include <boost/current_function.hpp>
#include <iostream>

int main()
{
  const char *funcname = BOOST_CURRENT_FUNCTION;
  std::cout << funcname << '\n';
}
```

Example 69.2 uses the macro `BOOST_CURRENT_FUNCTION`, defined in `boost/current_function.hpp`, to return the name of the surrounding function as a string.
`BOOST_CURRENT_FUNCTION` provides a platform-independent way to retrieve the name of a function. Starting with C++11, you can do the same thing with the standardized macro `__func__`. Before C++11, compilers like Visual C++ and GCC supported the macro `__FUNCTION__` as an extension. `BOOST_CURRENT_FUNCTION` uses whatever macro is supported by the compiler.
If compiled with Visual C++ 2013, Example 69.2 displays `int __cdecl main(void)`.

Example 69.3 Using `boost::prior()` and `boost::next()`

```
#include <boost/next_prior.hpp>
#include <array>
#include <algorithm>
#include <iostream>

int main()
{
  std::array<char, 4> a{{'a', 'c', 'b', 'd'}};

  auto it = std::find(a.begin(), a.end(), 'b');
  auto prior = boost::prior(it, 2);
  auto next = boost::next(it);

  std::cout << *prior << '\n';
  std::cout << *it << '\n';
  std::cout << *next << '\n';
}
```

Boost.Utility provides two functions, `boost::prior()` and `boost::next()`, that return an iterator relative to another iterator. In Example 69.3, **`it`** points to “b” in the array, **`prior`** points to “a”, and **`next`** to “d”.
Unlike `std::advance()`, `boost::prior()` and `boost::next()` return a new iterator and do not modify the iterator that was passed in.
In addition to the iterator, both functions accept a second parameter that indicates the number of steps to move forward or backward. In Example 69.3, the iterator is moved two steps backward in the call to `boost::prior()` and one step forward in the call to `boost::next()`.
The number of steps is always a positive number, even for `boost::prior()`, which moves backwards.
To use `boost::prior()` and `boost::next()`, include the header file `boost/next_prior.hpp`.
Both functions were added to the standard library in C++11, where they are called `std::prev()` and `std::next()`. They are defined in the header file `iterator`.

Example 69.4 Using `boost::noncopyable`

```
#include <boost/noncopyable.hpp>
#include <string>
#include <utility>
#include <iostream>
```

```
struct animal : boost::noncopyable
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

void print(const animal &a)
{
  std::cout << a.name << '\n';
  std::cout << a.legs << '\n';
}

int main()
{
  animal a{"cat", 4};
  print(a);
}
```

Boost.Utility provides the class `boost::noncopyable`, which is defined in `boost/noncopyable.hpp`. This class makes it impossible to copy (and move) objects.
The same effect can be achieved by defining the copy constructor and assignment operator as private member functions or – since C++11 – by removing the copy constructor and assignment operator with `delete`. However, deriving from `boost::noncopyable` explicitly states the intention that objects of a class should be non-copyable.

> **Note**
>
> Some developers prefer `boost::noncopyable` while others prefer to remove member functions explicitly with `delete`. You will find arguments for both approaches at Stack Overflow, among other places.

Example 69.4 can be compiled and executed. However, if the signature of the `print()` function is modified to take an object of type `animal` by value rather than by reference, the resulting code will no longer compile.

Example 69.5 Using `boost::addressof()`

```
#include <boost/utility/addressof.hpp>
#include <string>
#include <iostream>

struct animal
{
  std::string name;
  int legs;

  int operator&() const { return legs; }
};

int main()
{
  animal a{"cat", 4};
  std::cout << &a << '\n';
  std::cout << boost::addressof(a) << '\n';
}
```

To retrieve the address of a particular object, even if `operator&` has been overloaded, Boost.Utility provides the function `boost::addressof()`, which is defined in `boost/utility/addressof.hpp` (see Example 69.5). With C++11, this function became part of the standard library and is available as `std::addressof()`

in the header file memory.

Example 69.6 Using BOOST_BINARY

```
#include <boost/utility/binary.hpp>
#include <iostream>

int main()
{
  int i = BOOST_BINARY(1001 0001);
  std::cout << i << '\n';

  short s = BOOST_BINARY(1000 0000 0000 0000);
  std::cout << s << '\n';
}
```

The macro BOOST_BINARY lets you create numbers in binary form. Standard C++ only supports hexadecimal and octal forms, using the prefixes 0x and 0. C++11 introduced user-defined literals, which allows you to define custom suffixes, but there still is no standard way of using numbers in binary form in C++11.

Example 69.6 displays 145 and -32768. The bit sequence stored in **s** represents a negative number because the 16-bit type short uses the 16th bit – the most significant bit in short – as the sign bit.

BOOST_BINARY simply offers another option to write numbers. Because, in C++, the default type for numbers is int, BOOST_BINARY also uses int. To define a number of type long, use the macro BOOST_BINARY_L, which generates the equivalent of a number suffixed with the letter L.

Boost.Utility includes additional macros such as BOOST_BINARY_U, which initializes a variable without a sign bit. All of these macros are defined in the header file boost/utility/binary.hpp.

Example 69.7 Using boost::string_ref

```
#include <boost/utility/string_ref.hpp>
#include <iostream>

boost::string_ref start_at_boost(boost::string_ref s)
{
  auto idx = s.find("Boost");
  return (idx != boost::string_ref::npos) ? s.substr(idx) : "";
}

int main()
{
  boost::string_ref s = "The Boost C++ Libraries";
  std::cout << start_at_boost(s) << '\n';
}
```

Example 69.7 introduces the class boost::string_ref, which is a reference to a string that only supports read access. To a certain extent, the reference is comparable with const std::string&. However, const std::string& requires the existence of an object of type std::string. boost::string_ref can also be used without std::string. The benefit of boost::string_ref is that, unlike std::string, it requires no memory to be allocated.

Example 69.7 looks for the word “Boost” in a string. If found, a string starting with that word is displayed. If the word “Boost” isn’t found, an empty string is displayed. The type of the string **s** in main() isn’t std::string, it’s boost::string_ref. Thus no memory is allocated with new and no copy is created. **s** points to the literal string “The Boost C++ Libraries” directly.

The type of the return value of start_at_boost() is boost::string_ref, not std::string. The function doesn’t return a new string, it returns a reference. This reference is to either a substring of the parameter or an empty string. start_at_boost() requires that the original string remains valid as long as references of type boost::string_ref are in use. If this is guaranteed, as in Example 69.7, memory allocations can be avoided.

Additional utilities are also available, but they are beyond the scope of this book because they are mostly used by the developers of Boost libraries or for template meta programming. The documentation of Boost.Utility provides a fairly comprehensive overview of these additional utilities and can serve as a starting point if you are interested.

Chapter 70

Boost.Assign

The library Boost.Assign provides helper functions to initialize containers or add elements to containers. These functions are especially helpful if many elements need to be stored in a container. Thanks to the functions offered by Boost.Assign, you don't need to call a member function like `push_back()` repeatedly to insert elements one by one into a container.

If you work with a development environment that supports C++11, you can benefit from initializer lists. Usually you can pass any number of values to the constructor to initialize containers. Thanks to initializer lists, you don't have to depend on Boost.Assign with C++11. However, Boost.Assign provides helper functions to add multiple values to an existing container. These helper functions can be useful in C++11 development environments.

Example 70.1 introduces a few functions that containers can be initialized with. To use the functions defined by Boost.Assign, include the header file `boost/assign.hpp`.

Boost.Assign provides three functions to initialize containers. The most important, and the one you usually work with, is `boost::assign::list_of()`. You use `boost::assign::map_list_of()` with `std::map` and `boost::assign::tuple_list_of()` to initialize tuples in a container.

You don't have to use `boost::assign::map_list_of()` or `boost::assign::tuple_list_of()`. You can initialize any container with `boost::assign::list_of()`. However, if you use `std::map` or a container with tuples, you must pass a template parameter to `boost::assign::list_of()` that tells the function how elements are stored in the container. This template parameter is not required for `boost::assign::map_list_of()` and `boost::assign::tuple_list_of()`.

Example 70.1 Initializing containers

```
#include <boost/assign.hpp>
#include <boost/tuple/tuple.hpp>
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <utility>

using namespace boost::assign;

int main()
{
  std::vector<int> v = list_of(1)(2)(3);

  std::stack<int> s = list_of(1)(2)(3).to_adapter();

  std::vector<std::pair<std::string, int>> v2 =
    list_of<std::pair<std::string, int>>("a", 1)("b", 2)("c", 3);

  std::map<std::string, int> m =
    map_list_of("a", 1)("b", 2)("c", 3);

  std::vector<boost::tuple<std::string, int, double>> v3 =
    tuple_list_of("a", 1, 9.9)("b", 2, 8.8)("c", 3, 7.7);
}
```

All three functions return a proxy object. This object overloads the operator `operator()`. You can call this operator multiple times to save values in the container. Even though you access another object, and not the container, the container is changed through this proxy object.
If you want to initialize adapters like `std::stack`, call the member function `to_adapter()` on the proxy. The proxy then calls the member function `push()`, which is provided by all adapters.
`boost::assign::tuple_list_of()` supports tuples of type `boost::tuple` only. You cannot use this function to initialize containers with tuples from the standard library.
Example 70.2 illustrates how values can be added to existing containers.
The `boost::assign::push_back()`, `boost::assign::push_front()`, `boost::assign::insert()`, and `boost::assign::push()` functions of Boost.Assign return a proxy. You pass these functions the container you want to add new elements to. Then, you call the operator `operator()` on the proxy and pass the values you want to store in the container.

Example 70.2 Adding values to containers

```
#include <boost/assign.hpp>
#include <vector>
#include <deque>
#include <set>
#include <queue>

int main()
{
  std::vector<int> v;
  boost::assign::push_back(v)(1)(2)(3);

  std::deque<int> d;
  boost::assign::push_front(d)(1)(2)(3);

  std::set<int> s;
  boost::assign::insert(s)(1)(2)(3);

  std::queue<int> q;
  boost::assign::push(q)(1)(2)(3);
}
```

The four functions `boost::assign::push_back()`, `boost::assign::push_front()`, `boost::assign::insert()`, and `boost::assign::push()` are called in this manner because the proxies returned call the identically named member functions on the container. Example 70.2 adds the three numbers 1, 2, and 3 to the vector **v** with three calls to `push_back()`.
Boost.Assign provides additional helper functions you can use to add values to a container, including `boost::assign::add_edge()`, which you can use to get a proxy for a graph from Boost.Graph.

Chapter 71

Boost.Swap

If you use many Boost libraries and also use `std::swap()` to swap data, consider using `boost::swap()` as an alternative. `boost::swap()` is provided by Boost.Swap and is defined in `boost/swap.hpp`.

Example 71.1 Using `boost::swap()`

```
#include <boost/swap.hpp>
#include <boost/array.hpp>
#include <iostream>

int main()
{
  char c1 = 'a';
  char c2 = 'b';

  boost::swap(c1, c2);

  std::cout << c1 << c2 << '\n';

  boost::array<int, 1> a1{{1}};
  boost::array<int, 1> a2{{2}};

  boost::swap(a1, a2);

  std::cout << a1[0] << a2[0] << '\n';
}
```

`boost::swap()` does nothing different from `std::swap()`. However, because many Boost libraries offer specializations for swapping data that are defined in the namespace `boost`, `boost::swap()` can take advantage of them. In Example 71.1, `boost::swap()` accesses `std::swap()` to swap the values of the two `char` variables and uses the optimized function `boost::swap()` from Boost.Array to swap data in the arrays.
Example 71.1 writes `ba` and `21` to the standard output stream.

Chapter 72

Boost.Operators

Boost.Operators provides numerous classes to automatically overload operators. In Example 72.1, a greater-than operator is automatically added, even though there is no declaration, because the greater-than operator can be implemented using the already defined less-than operator.

Example 72.1 Greater-than operator with `boost::less_than_comparable`

```
#include <boost/operators.hpp>
#include <string>
#include <utility>
#include <iostream>

struct animal : public boost::less_than_comparable<animal>
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}

  bool operator<(const animal &a) const { return legs < a.legs; }
};

int main()
{
  animal a1{"cat", 4};
  animal a2{"spider", 8};

  std::cout << std::boolalpha << (a2 > a1) << '\n';
}
```

To automatically add operators, derive a class from classes defined by Boost.Operators in `boost/operators.hpp`. If a class is derived from `boost::less_than_comparable`, then `operator>`, `operator<=`, and `operator>=` are automatically defined.

Because many operators can be expressed in terms of other operators, automatic overloading is possible. For example, `boost::less_than_comparable` implements the greater-than operator as the opposite of the less-than operator; if an object isn't less than another, it must be greater, assuming they aren't equal.

If two objects can be equal, use `boost::partially_ordered` as the base class. By defining `operator==`, `boost::partially_ordered` can determine whether less than really means greater than or equal.

In addition to `boost::less_than_comparable` and `boost::partially_ordered`, classes are provided that allow you to overload arithmetic and logical operators. Classes are also available to overload operators usually provided by iterators, pointers, or arrays. Because automatic overloading is only possible once other operators have been defined, the particular operators that must be provided will vary depending on the situation. Consult the documentation for more information.

Index