Antony Polukhin

Boost C++ Application Development Cookbook

Second Edition

Recipies to simplify your Application Development

Packt

Boost C++ Application Development Cookbook

Second Edition

Recipes to simplify your application development

Antony Polukhin

BIRMINGHAM - MUMBAI

Boost C++ Application Development Cookbook

Second Edition

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Second edition: August 2017

Production reference: 1280817

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78728-224-7

www.packtpub.com

Credits

Author Antony Polukhin	**Copy Editor** Charlotte Carneiro
Reviewer Glen Fernandes	**Project Coordinator** Sheejal Shah
Commissioning Editor Merint Mathew	**Proofreader** Safis Editing
Acquisition Editor	**Indexer**

Nitin Dasan	Rekha Nair
Content Development Editor Sreeja Nair	**Graphics** Jason Monteiro
Technical Editor Surabhi Kulkarni	**Production Coordinator** Melwyn D’sa

About the Author

If you are wondering who is **Antony Polukhin** and could he be trusted to teach about C++ and Boost libraries, then here are some facts:

- Antony Polukhin currently represents Russia in international C++ standardization committee
- He is the author of multiple Boost libraries and maintains (keeps an eye on) some of the old Boost libraries
- He is a perfectionist: all the source codes from the book are auto tested on multiple platforms using different C++ standards.

But let's start from the beginning.

Antony Polukhin was born in Russia. As a child, he could speak the Russian and Hungarian languages and learned English at school. Since his school days, he was participating in different mathematics, physics, and chemistry competitions and winning them.

He was accepted into University twice: once for taking part in a city mathematics competition and again for gaining high score in an University's mathematics and physics challenge. In his university life, there was a year when he did not participate in exams at all: he gained A's in all disciplines by writing highly difficult programs for each teacher. He met his future wife in university and graduated with honors.

For more than three years, he worked in a VoIP company developing business logic for a commercial alternative to Asterisc. During those days he started contributing to Boost and became a maintainer of the `Boost.LexicalCast` library. He also started making translations to Russian for Ubuntu Linux at that time.

Today, he works for Yandex Ltd., helps Russian speaking people with C++ standardization proposals, continues to contribute to the open source and to the C++ language in general. You may find his code in Boost libraries such as Any, Conversion, DLL, LexicalCast, Stacktrace, TypeTraits, Variant, and others.

He has been happily married for more than five years.

I would like to thank my family, especially my wife, Irina Polukhina, for drawing sketches of pictures and diagrams all through the book.
Great thanks to Paul Anthony Bristow for reviewing the first edition of this book and getting through the insane number of commas that I used in the first drafts.
Separate thanks to Glen Joseph Fernandes for providing a lot of useful information and comments for the second edition.
I would also like to thank all of the people from the Boost community for writing those great libraries and for opening an amazing word of C++ for me.

About the Reviewer

Glen Joseph Fernandes has worked at both Intel and Microsoft as a Software Engineer. He is the author of the *Boost Align library*, a major contributor to the *Boost Smart Pointers* and *Boost Core libraries*, and has also contributed to several other Boost C++ libraries. He is a contributor to the *ISO C++ Standard* by authoring proposal papers and defect reports, and even has at least one feature accepted for the upcoming *C++20 standard* (P0674r1: *Extending make_shared to Support Arrays*). Glen lives with his wife, Caroline, and daughter, Aeryn, in the US, graduated from the University of Sydney in Australia, and, before all that, lived in New Zealand.

www.PacktPub.com

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at https://www.amazon.com/dp/1787282244.

If you'd like to join our team of regular reviewers, you can e-mail us at `customerreviews@packtpub.com`. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

If you want to take advantage of the real power of Boost and C++ and avoid the confusion about which library to use in which situation, then this book is for you.

Beginning with the basics of Boost C++, you will move on to learn how the Boost libraries simplify application development. You will learn to convert data, such as string to numbers, numbers to string, numbers to numbers, and more. Managing resources will become a piece of cake. You’ll see what kind of work can be done at compile time and what Boost containers can do. You will learn everything for the development of high-quality, fast, and portable applications. Write a program once, and then you can use it on Linux, Windows, macOS, and Android operating systems. From manipulating images to graphs, directories, timers, files, and networking, everyone will find an interesting topic. Note that the knowledge from this book won’t get outdated, as more and more Boost libraries become part of the C++ Standard.

What this book covers

Chapter 1, *Starting to Write Your Application*, tells you about libraries for everyday use. We'll see how to get configuration options from different sources and what can be cooked up using some of the data types introduced by Boost library authors.

Chapter 2, *Managing Resources*, deals with data types, introduced by the Boost libraries, mostly focusing on working with pointers. We'll see how to easily manage resources, and how to use a data type capable of storing any functional objects, functions, and lambda expressions. After reading this chapter, your code will become more reliable, and memory leaks will become history.

Chapter 3, *Converting and Casting*, describes how to convert strings, numbers, and user-defined types to each other, how to safely cast polymorphic types, and how to write small and large parsers right inside the C++ source files. Multiple ways of converting data for everyday use and for rare cases are covered.

Chapter 4, *Compile-Time Tricks*, describes some basic examples of Boost libraries can be used in compile-time checking for tuning algorithms, and in other metaprogramming tasks. Understanding Boost sources and other Boost-like libraries is impossible without it.

Chapter 5, *Multithreading*, focuses on the basics of multithreaded programming and all of the stuff connected with them.

Chapter 6, *Manipulating Tasks*, shows calling the functional object a task. The main idea of this chapter is that we can split all the processing, computations, and interactions into functors (tasks) and process each of those tasks almost independently. Moreover, we may not block on some slow operations (such as receiving data from a socket or waiting for a time-out), but instead provide a callback task and continue working with other tasks. Once the OS finishes the slow operation, our callback will be executed.

Chapter 7, *Manipulating Strings*, shows different aspects of changing, searching, and representing strings. We'll see how some common string-related tasks can be easily done using the Boost libraries. It addresses very common string manipulation tasks.

Chapter 8, *Metaprogramming*, presents some cool and hard-to-understand metaprogramming methods. In this chapter, we'll go deeper and see how multiple types can be packed into a single tuple-like type. We'll make functions to manipulate collections of types, we'll see how types of compile-time collections can be changed, and how compile-time tricks can be mixed with runtime.

Chapter 9, *Containers*, is about boost containers and the things directly connected with them. This chapter provides information about the Boost classes that can be used in everyday programming, which will make your code much faster and the development of new applications easier.

Chapter 10, *Gathering Platform and Compiler Information*, describes different helper macros used to detect compiler, platform, and Boost features—macros that are widely used across boost libraries and that are essential for writing portable code that is able to work with any compiler flags.

Chapter 11, *Working with the System*, provides a closer look at the filesystem and how to create and delete files. We'll see how data can be passed between different system processes, how to read files at the maximum speed, and how to perform other tricks.

Chapter 12, *Scratching the Tip of the Iceberg*, is devoted to some big libraries and to giving you some basics to start with.

What you need for this book

You need a modern C++ compiler, Boost libraries (any version will be OK, 1.65 or a more recent version is recommended), and QtCreator/qmake, or just navigate to http://apolukhin.GitHub.io/Boost-Cookbook/ to run and experiment with examples online.

Who this book is for

This book is for developers looking to improve their knowledge of Boost and who would like to simplify their application development processes. Prior C++ knowledge and basic knowledge of the standard library is assumed.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it…, How it works…, There’s more…, and See also). To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There’s more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows:

“Remember that this library is not only a header, so your program has to link against the `libboost_program_options` library“.

A block of code is set as follows:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
int main(int argc, char *argv[])
{
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
#include <boost/program_options.hpp>
#include <iostream>
namespace opt = boost::program_options;
int main(int argc, char *argv[])
```

Any command-line input or output is written as follows:

```
 $ ./our_program.exe --apples=10 --oranges=20
Fruits count: 30
```

New terms and **important words** are shown in bold.

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you’re looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://GitHub.com/PacktPublishing/Boost-Cpp-Application-Development-Cookbook-Second-Edition. We also have other code bundles from our rich catalogue of books and videos available at https://github.com/PacktPublishing/. Check them out!

The source code files of the examples presented in this cookbook are also hosted in the author’s GitHub repository. You can visit the author’s repository at https://GitHub.com/apolukhin/Boost-Cookbook to obtain the latest version of the code..

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

Starting to Write Your Application

In this chapter, we will cover:

- Getting configuration options
- Storing any value in a container/variable
- Storing multiple chosen types in a container/variable
- Using a safer way to work with a container that stores multiple chosen types
- Returning a value or flag where there is no value
- Returning an array from a function
- Combining multiple values into one
- Binding and reordering function parameters
- Getting a human-readable type name
- Using the C++11 move emulation
- Making a noncopyable class
- Making a noncopyable but movable class
- Using C++14 and C++11 algorithms

Introduction

Boost is a collection of C++ libraries. Each library has been reviewed by many professional programmers before being accepted by Boost. Libraries are tested on multiple platforms using many compilers and many C++ standard library implementations. While using Boost, you can be sure that you are using one of the most portable, fast, and reliable solutions that is distributed under a license suitable for commercial and open source projects.

Many parts of Boost have been included into C++11, C++14, and C++17. Furthermore, Boost libraries will be included in the next standard of C++. You will find C++ standard-specific notes in each recipe of this book.

Without a long introduction, let’s get started!

In this chapter, we will see some recipes for everyday use. We’ll see how to get configuration options from different sources and what can be cooked up using some of the data types introduced by Boost library authors.

Getting configuration options

Take a look at some of the console programs, such as `cp` in Linux. They all have a fancy help; their input parameters do not depend on any position and have a human-readable syntax. For example:

```
$ cp --help
Usage: cp [OPTION]... [-T] SOURCE DEST
  -a, --archive           same as -dR --preserve=all
  -b                      like --backup but does not accept an argument
```

You can implement the same functionality for your program in 10 minutes. All you need is the `Boost.ProgramOptions` library.

Getting ready

Basic knowledge of C++ is all you need for this recipe. Remember that this library is not only a header, so your program has to link against the `libboost_program_options` library.

How to do it…

Let’s start with a simple program that accepts the count of `apples` and `oranges` as input and counts the total number of fruits. We want to achieve the following result:

```
 $ ./our_program.exe --apples=10 --oranges=20
Fruits count: 30
```

Perform the following steps:

1. Include the `boost/program_options.hpp` header and make an alias for the `boost::program_options` namespace (it is too long to type it!). We would also need an `<iostream>` header:

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

namespace opt = boost::program_options;
```

2. Now, we are ready to describe our options in the `main()` function:

```
int main(int argc, char *argv[])
{
    // Constructing an options describing variable and giving
    // it a textual description "All options".
    opt::options_description desc("All options");

    // When we are adding options, first parameter is a name
    // to be used in command line. Second parameter is a type
    // of that option, wrapped in value<> class. Third parameter
    // must be a short description of that option.
    desc.add_options()
        ("apples", opt::value<int>(), "how many apples do
                                       you have")
        ("oranges", opt::value<int>(), "how many oranges do you
                                        have")
        ("help", "produce help message")
    ;
```

3. Let’s parse the command line:

```
    // Variable to store our command line arguments.
    opt::variables_map vm;

    // Parsing and storing arguments.
    opt::store(opt::parse_command_line(argc, argv, desc), vm);

    // Must be called after all the parsing and storing.
    opt::notify(vm);
```

4. Let’s add some code for processing the `help` option:

```
    if (vm.count("help")) {
        std::cout << desc << "\n";
        return 1;
    }
```

5. Final step. Counting fruits may be implemented in the following way:

```
    std::cout << "Fruits count: "
```

```
                << vm["apples"].as<int>() + vm["oranges"].as<int>()
                << std::endl;

        } // end of `main`
```

Now, if we call our program with the `help` parameter, we'll get the following output:

```
All options:
    --apples arg          how many apples do you have
    --oranges arg        how many oranges do you have
    --help                    produce help message
```

As you can see, we do not provide a type for the `help` option's value, because we do not expect any values to be passed to it.

How it works…

This example is pretty simple to understand from code and comments. Running it produces the expected result:

```
 $ ./our_program.exe --apples=100 --oranges=20
Fruits count: 120
```

There’s more…

The C++ standard adopted many Boost libraries; however, you won’t find `Boost.ProgramOptions` even in C++17. Currently, there’s no plan to adopt it into C++2a.

The `ProgramOptions` library is very powerful and has many features. Here’s how to:

- Parse configuration option values directly into a variable and make that option a required one:

```
int oranges_var = 0;
desc.add_options()
    // ProgramOptions stores the option value into
    // the variable that is passed by pointer. Here value of
    // "--oranges" option will be stored into 'oranges_var'.
    ("oranges,o", opt::value<int>(&oranges_var)->required(),
                                            "oranges you have")
```

- Get some mandatory string option:

```
    // 'name' option is not marked with 'required()',
    // so user may not provide it.
    ("name", opt::value<std::string>(), "your name")
```

- Add short name for apple, set `10` as a default value for `apples`:

```
    // 'a' is a short option name for apples. Use as '-a 10'.
    // If no value provided, then the default value is used.
    ("apples,a", opt::value<int>()->default_value(10),
                               "apples that you have");
```

- Get the missing options from the configuration file:

```
opt::variables_map vm;

// Parsing command line options and storing values to 'vm'.
opt::store(opt::parse_command_line(argc, argv, desc), vm);

// We can also parse environment variables. Just use
// 'opt::store with' 'opt::parse_environment' function.

// Adding missing options from "apples_oranges.cfg" config file.
try {
    opt::store(
        opt::parse_config_file<char>("apples_oranges.cfg", desc),
        vm
    );
} catch (const opt::reading_file& e) {
    std::cout << "Error: " << e.what() << std::endl;
}
```

The configuration file syntax differs from the command-line syntax. We do not need to place minuses before the options. So, our `apples_oranges.cfg` file must look like this:
`oranges=20`

- Validate that all the required options were set:

```
try {
```

```
            // `opt::required_option` exception is thrown if
            // one of the required options was not set.
            opt::notify(vm);

        } catch (const opt::required_option& e) {
            std::cout << "Error: " << e.what() << std::endl;
            return 2;
        }
```

If we combine all the mentioned tips into a single executable, then its `help` command will produce this output:

```
$ ./our_program.exe --help
 All options:
   -o [ --oranges ] arg          oranges that you have
   --name arg                       your name
   -a [ --apples ] arg (=10)  apples that you have
   --help                              produce help message
```

Running it without a configuration file will produce the following output:

```
$ ./our_program.exe
 Error: can not read options configuration file 'apples_oranges.cfg'
 Error: the option '--oranges' is required but missing
```

Running the program with `oranges=20` in the configuration file will generate ++, because the default value for apples is `10`:

```
$ ./our_program.exe
 Fruits count: 30
```

See also

- Boost's official documentation contains many more examples and tells us about more advanced features of `Boost.ProgramOptions`, such as position-dependent options, nonconventional syntax, and more; this is available at http://boost.org/libs/program_options
- You can modify and run all the examples from this book online at http://apolukhin.github.io/Boost-Cookbook

Storing any value in a container/variable

If you have been programming in Java, C#, or Delphi, you will definitely miss the ability of creating containers with the `Object` value type in C++. The `Object` class in those languages is a basic class for almost all types, so you are able to assign almost any value to it at any time. Just imagine how great it would be to have such a feature in C++:

```
typedef std::unique_ptr<Object> object_ptr;

std::vector<object_ptr> some_values;
some_values.push_back(new Object(10));
some_values.push_back(new Object("Hello there"));
some_values.push_back(new Object(std::string("Wow!")));

std::string* p = dynamic_cast<std::string*>(some_values.back().get());
assert(p);
(*p) += " That is great!\n";
std::cout << *p;
```

Getting ready

We’ll be working with the header-only library. The basic knowledge of C++ is all you need for this recipe.

How to do it…

Boost offers a solution, the `Boost.Any` library, that has an even better syntax:

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

int main() {
    std::vector<boost::any> some_values;
    some_values.push_back(10);
    some_values.push_back("Hello there!");
    some_values.push_back(std::string("Wow!"));

    std::string& s = boost::any_cast<std::string&>(some_values.back());
    s += " That is great!";
    std::cout << s;
}
```

Great, isn’t it? By the way, it has an empty state, which could be checked using the `empty()` member function (just like in standard library containers).

You can get the value from `boost::any` using two approaches:

```
void example() {
    boost::any variable(std::string("Hello world!"));

    // Following method may throw a boost::bad_any_cast exception
    // if actual value in variable is not a std::string.
    std::string s1 = boost::any_cast<std::string>(variable);

    // Never throws. If actual value in variable is not a std::string
    // will return an NULL pointer.
    std::string* s2 = boost::any_cast<std::string>(&variable);
}
```

How it works…

The `boost::any` class just stores any value in it. To achieve this, it uses the **type erasure** technique (close to what Java or C# does with all types). To use this library you do not really need to know its internal implementation in detail, but here's a quick glance at the type erasure technique for the curious.

On the assignment of some variable of type `T`, `Boost.Any` instantiates a `holder<T>` type that may store a value of the specified type `T` and is derived from some base-type `placeholder`:

```
template<typename ValueType>
struct holder : public placeholder {
    virtual const std::type_info& type() const {
         return typeid(ValueType);
    }
     ValueType held;
};
```

A `placeholder` type has virtual functions for getting `std::type_info` of a stored type `T` and for cloning a stored type:

```
struct placeholder {
    virtual ~placeholder() {}
    virtual const std::type_info& type() const = 0;
};
```

`boost::any` stores `ptr`— a pointer to `placeholder`. When `any_cast<T>()` is used, `boost::any` checks that calling `ptr->type()` gives `std::type_info` equal to `typeid(T)` and returns `static_cast<holder<T>*>(ptr)->held`.

There’s more…

Such flexibility never comes without any cost. Copy constructing, value constructing, copy assigning, and assigning values to instances of `boost::any` do dynamic memory allocation; all the type casts do **RunTime Type Information** (**RTTI**) checks; `boost::any` uses virtual functions a lot. If you are keen on performance, the next recipe will give you an idea of how to achieve almost the same results without dynamic allocations and RTTI usage.

`boost::any` makes use of **rvalue references** but can not be used in **constexpr**.

The `Boost.Any` library was accepted into C++17. If your compiler is C++17 compatible and you wish to avoid using Boost for `any`, just replace the `boost` namespace with namespace `std` and include `<any>` instead of `<boost/any.hpp>`. Your standard library implementation may work slightly faster if you are storing tiny objects in `std::any`.

> *`std::any` has the `reset()` function instead of `clear()` and `has_value()` instead of `empty()`. Almost all exceptions in Boost derived from the `std::exception` class or from its derivatives, for example, `boost::bad_any_cast` is derived from `std::bad_cast`. It means that you can catch almost all Boost exceptions using `catch (const std::exception& e)`.*

See also

- Boost’s official documentation may give you some more examples; it can be found at http://boost.org/libs/any
- The *Using a safer way to work with a container that stores multiple chosen types* recipe for more info on the topic

Storing multiple chosen types in a container/variable

C++03 unions can only hold extremely simple types called **Plain Old Data** (**POD**). For example in C++03, you cannot store `std::string` or `std::vector` in a union.

Are you aware of the concept of **unrestricted unions** in C++11? Let me tell you about it briefly. C++11 relaxes requirements for unions, but you have to manage the construction and destruction of non POD types by yourself. You have to call in-place construction/destruction and remember what type is stored in a union. A huge amount of work, isn't it?

Can we have an unrestricted union like variable in C++03 that manages the object lifetime and remembers the type it has?

Getting ready

We’ll be working with the header-only library, which is simple to use. Basic knowledge of C++ is all you need for this recipe.

How to do it…

Let me introduce the `Boost.Variant` library to you.

1. The `Boost.Variant` library can store any of the types specified at compile time. It also manages in-place construction/destruction and even does not even require the C++11 standard:

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

int main() {
    typedef boost::variant<int, const char*, std::string> my_var_t;
    std::vector<my_var_t> some_values;
    some_values.push_back(10);
    some_values.push_back("Hello there!");
    some_values.push_back(std::string("Wow!"));

    std::string& s = boost::get<std::string>(some_values.back());
    s += " That is great!\n";
    std::cout << s;
}
```

   Great, isn't it?

2. `Boost.Variant` has no empty state, but has an `empty()` function which is useless and always returns `false`. If you need to represent an empty state, just add some simple type at the first position of the types supported by the `Boost.Variant` library. When `Boost.Variant` contains that type, interpret it as an empty state. Here is an example in which we will use a `boost::blank` type to represent an empty state:

```
void example1() {
    // Default constructor constructs an instance of boost::blank.
    boost::variant<
        boost::blank, int, const char*, std::string
    > var;

    // 'which()' method returns an index of a type
    // currently held by variant.
    assert(var.which() == 0); // boost::blank

    var = "Hello, dear reader";
    assert(var.which() != 0);
}
```

3. You can get a value from a variant using two approaches:

```
void example2() {
    boost::variant<int, std::string> variable(0);

    // Following method may throw a boost::bad_get
    // exception if actual value in variable is not an int.
    int s1 = boost::get<int>(variable);

    // If actual value in variable is not an int will return NULL.
    int* s2 = boost::get<int>(&variable);
}
```

How it works…

The `boost::variant` class holds an array of bytes and stores values in that array. The size of the array is determined at compile time by applying `sizeof()` and functions to get alignment to each of the template types. On assignment, or construction of `boost::variant`, the previous values are in-place destructed and new values are constructed on top of the byte array, using the placement new.

There's more…

The `Boost.Variant` variables usually do not dynamically allocate memory, and they do not require RTTI to be enabled. `Boost.Variant` is extremely fast and used widely by other Boost libraries. To achieve maximum performance, make sure that there is a simple type in the list of supported types at the first position. `boost::variant` takes advantage of C++11 rvalue references if they are available on your compiler.

`Boost.Variant` is part of the C++17 standard. `std::variant` differs slightly from the`boost::variant`:

- `std::variant` is declared in the `<variant>` header file rather than in `<boost.variant.hpp>`
- `std::variant` never ever allocates memory
- `std::variant` is usable with constexpr
- Instead of writing `boost::get<int>(&variable)`, you have to write `std::get_if<int>(&variable)` for `std::variant`
- `std::variant` can not recursively hold itself and misses some other advanced techniques
- `std::variant` can in-place construct objects
- `std::variant` has `index()` instead of `which()`

See also

- The *Using a safer way to work with a container that stores multiple chosen types* recipe
- Boost's official documentation contains more examples and descriptions of some other features of `Boost.Variant`, and can be found at: http://boost.org/libs/variant
- Experiment with the code online at http://apolukhin.github.io/Boost-Cookbook

Using a safer way to work with a container that stores multiple chosen types

Imagine that you are creating a wrapper around some SQL database interface. You decided that `boost::any` will perfectly match the requirements for a single cell of the database table.

Some other programmer will use your classes, and his/her task would be to get a row from the database and count the sum of the arithmetic types in a row.

This is what such a code would look like:

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

// This typedefs and methods will be in our header,
// that wraps around native SQL interface.
typedef boost::any cell_t;
typedef std::vector<cell_t> db_row_t;

// This is just an example, no actual work with database.
db_row_t get_row(const char* /*query*/) {
    // In real application 'query' parameter shall have a 'const
    // char*' or 'const std::string&' type? See recipe "Type
    // 'reference to string'" for an answer.
    db_row_t row;
    row.push_back(10);
    row.push_back(10.1f);
    row.push_back(std::string("hello again"));
    return row;
}

// This is how a user will use your classes
struct db_sum {
private:
    double& sum_;
public:
    explicit db_sum(double& sum)
        : sum_(sum)
    {}

    void operator()(const cell_t& value) {
        const std::type_info& ti = value.type();
        if (ti == typeid(int)) {
            sum_ += boost::any_cast<int>(value);
        } else if (ti == typeid(float)) {
            sum_ += boost::any_cast<float>(value);
        }
    }
};

int main() {
    db_row_t row = get_row("Query: Give me some row, please.");
    double res = 0.0;
    std::for_each(row.begin(), row.end(), db_sum(res));
    std::cout << "Sum of arithmetic types in database row is: "
              << res << std::endl;
}
```

If you compile and run this example, it will output a correct answer:

```
Sum of arithmetic types in database row is: 20.1
```

Do you remember what your own thoughts were when reading the implementation of `operator()`? I guess they were, *"And what about double, long, short, unsigned, and other types?"* The same thoughts will come into the head of a programmer who will use your interface. So, you need to carefully document values stored by your `cell_t` or use a more elegant solution as described in the following sections.

Getting ready

Reading the previous two recipes is highly recommended if you are not already familiar with the `Boost.Variant` and `Boost.Any` libraries.

How to do it…

The `Boost.Variant` library implements a visitor programming pattern for accessing the stored data, which is much safer than getting values via `boost::get<>`. This pattern forces the programmer to take care of each type in variant, otherwise the code will fail to compile. You can use this pattern via the `boost::apply_visitor` function, which takes a `visitor` functional object as the first parameter and a `variant` as the second parameter. If you are using a pre C++14 compiler, then `visitor` functional objects must derive from the `boost::static_visitor<T>` class, where `T` is a type being returned by a `visitor`. A `visitor` object must have overloads of `operator()` for each type stored by a variant.

Let's change the `cell_t` type to `boost::variant<int, float, string>` and modify our example:

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

// This typedefs and methods will be in header,
// that wraps around native SQL interface.
typedef boost::variant<int, float, std::string> cell_t;
typedef std::vector<cell_t> db_row_t;

// This is just an example, no actual work with database.
db_row_t get_row(const char* /*query*/) {
    // See recipe "Type 'reference to string'"
    // for a better type for 'query' parameter.
    db_row_t row;
    row.push_back(10);
    row.push_back(10.1f);
    row.push_back("hello again");
    return row;
}

// This is a code required to sum values.
// We can provide no template parameter
// to boost::static_visitor<> if our visitor returns nothing.
struct db_sum_visitor: public boost::static_visitor<double> {
    double operator()(int value) const {
        return value;
    }
    double operator()(float value) const {
        return value;
    }
    double operator()(const std::string& /*value*/) const {
        return 0.0;
    }
};

int main() {
    db_row_t row = get_row("Query: Give me some row, please.");
    double res = 0.0;
    for (auto it = row.begin(), end = row.end(); it != end; ++it) {
        res += boost::apply_visitor(db_sum_visitor(), *it);
    }

    std::cout << "Sum of arithmetic types in database row is: "
              << res << std::endl;
}
```

How it works…

At compile time, the `Boost.Variant` library generates a big `switch` statement, each case of which calls a `visitor` for a single type from the variant's list of types. At runtime, the index of the stored type is retrieved using `which()` and jumps to a correct case in `switch` statement is made. Something like this will be generated for `boost::variant<int, float, std::string>`:

```
switch (which())
{
case 0 /*int*/:
    return visitor(*reinterpret_cast<int*>(address()));
case 1 /*float*/:
    return visitor(*reinterpret_cast<float*>(address()));
case 2 /*std::string*/:
    return visitor(*reinterpret_cast<std::string*>(address()));
default: assert(false);
}
```

Here, the `address()` function returns a pointer to the internal storage of `boost::variant<int, float, std::string>`.

There’s more…

If we compare this example with the first example in this recipe, we’ll see the following advantages of `boost::variant`:

- We know what types a variable can store
- If a library writer of the SQL interface adds or modifies a type held by a `variant`, we’ll get a compile-time error instead of incorrect behavior

`std::variant` from C++17 also supports visitation. Just write `std::visit` instead of `boost::apply_visitor` and you’re done.

You can download the example code files for all Packt books that you have purchased from your account at http://www.PacktPub.com*. If you purchased this book elsewhere, you can visit* http://www.PacktPub.com/*support, and register to have the files emailed directly to you.*

See also

- After reading some recipes from Chapter 4, *Compile-Time Tricks*, you'll be able to make generic `visitor` objects that work correctly even if underlying types change
- Boost's official documentation contains more examples and a description of some other features of `Boost.Variant`; it is available at the following link: http://boost.org/libs/variant

Returning a value or flag where there is no value

Imagine that we have a function that does not throw an exception and returns a value or indicates that an error has occurred. In Java or C# programming languages, such cases are handled by comparing a return value from a function value with a `null` pointer. If the function returned `null`, then an error has occurred. In C++, returning a pointer from a function confuses library users and usually requires slow dynamic memory allocation.

Getting ready

Only basic knowledge of C++ is required for this recipe.

How to do it…

Ladies and gentlemen, let me introduce you to the `Boost.Optional` library using the following example:

The `try_lock_device()` function tries to acquire a lock for a device and may succeed or not, depending on different conditions (in our example it depends on some `try_lock_device_impl()` function call):

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

class locked_device {
    explicit locked_device(const char* /*param*/) {
        // We have unique access to device.
        std::cout << "Device is locked\n";
    }

    static bool try_lock_device_impl();

public:
    void use() {
        std::cout << "Success!\n";
    }

    static boost::optional<locked_device> try_lock_device() {
        if (!try_lock_device_impl()) {
            // Failed to lock device.
            return boost::none;
        }

        // Success!
        return locked_device("device name");
    }

    ~locked_device(); // Releases device lock.
};
```

The function returns the `boost::optional` variable that can be converted to a `bool`. If the returned value is equal to `true`, then the lock is acquired and an instance of a class to work with the device can be obtained by dereferencing the returned optional variable:

```
int main() {
    for (unsigned i = 0; i < 10; ++i) {
        boost::optional<locked_device> t
            = locked_device::try_lock_device();

        // optional is convertible to bool.
        if (t) {
            t->use();
            return 0;
        } else {
            std::cout << "...trying again\n";
        }
    }

    std::cout << "Failure!\n";
    return -1;
}
```

This program will output the following:

```
        ...trying again
        ...trying again
        Device is locked
        Success!
```

The default constructed `optional` variable is convertible to `false` and must not be dereferenced, because such an `optional` does not have an underlying type constructed.

How it works…

`boost::optional<T>` under the hood has a properly aligned array of bytes where the object of type `T` can be an in-place constructed. It also has a `bool` variable to remember the state of the object (is it constructed or not?).

There's more…

The `Boost.Optional` class does not use dynamic allocation and it does not require a default constructor for the underlying type. The current `boost::optional` implementation can work with C++11 rvalue references but is not usable with constexpr.

> *If you have a class `T` that has no empty state but your program logic requires an empty state or uninitialized `T`, then you have to come up with some workaround. Traditionally, users create some smart pointer to the class `T`, keep a `nullptr` in it, and dynamically allocate `T` if non empty state is required. Stop doing that! Use `boost::optional<T>` instead. It's a much faster and more reliable solution.*

The C++17 standard includes the `std::optional` class. Just replace `<boost/optional.hpp>` with `<optional>` and `boost::` with `std::` to use the standard version of this class. `std::optional` is usable with constexpr.

See also

Boost’s official documentation contains more examples and describes advanced features of `Boost.Optional` (like in-place construction). The documentation is available at the following link: http://boost.org/libs/optional.

Returning an array from a function

Let’s play a game of guessing! What can you tell about the following function?

```
char* vector_advance(char* val);
```

Should return values be deallocated by the programmer or not? Does the function attempt to deallocate the input parameter? Should the input parameter be zero-terminated, or should the function assume that the input parameter has a specified width?

Now, let’s make the task harder! Take a look at the following line:

```
char ( &vector_advance( char (&val)[4] ) )[4];
```

Do not worry. I’ve also been scratching my head for half an hour before getting an idea of what is happening here. `vector_advance` is a function that accepts and returns an array of four elements. Is there a way to write such a function clearly?

Getting ready

Only basic knowledge of C++ is required for this recipe.

How to do it…

We can rewrite the function like this:

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

typedef boost::array<char, 4> array4_t;
array4_t& vector_advance(array4_t& val);
```

Here, `boost::array<char, 4>` is just a simple wrapper around an array of four `char` elements.

This code answers all the questions from our first example and is much more readable than the code from the second example.

How it works…

`boost::array` is a fixed-size array. The first template parameter of `boost::array` is the element type and the second one is the size of an array. If you need to change the array size at runtime, use `std::vector` , `boost::container::small_vector`, `boost::container::stack_vector`, or `boost::container::vector` instead.

The `boost::array<>` class has no handwritten constructors and all its members are public, so the compiler will treat it as a POD type.

There’s more…

Let’s see some more examples of the usage of `boost::array`:

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

typedef boost::array<char, 4> array4_t;

array4_t& vector_advance(array4_t& val) {
    // C++11 lambda function
    const auto inc = [](char& c){ ++c; };

    // boost::array has begin(), cbegin(), end(), cend(),
    // rbegin(), size(), empty() and other functions that are
    // common for standard library containers.
    std::for_each(val.begin(), val.end(), inc);
    return val;
}

int main() {
    // We can initialize boost::array just like an array in C++11:
    // array4_t val = {0, 1, 2, 3};
    // but in C++03 additional pair of curly brackets is required.
    array4_t val = {{0, 1, 2, 3}};

    array4_t val_res;               // it is default constructible
    val_res = vector_advance(val);  // it is assignable

    assert(val.size() == 4);
    assert(val[0] == 1);
    /*val[4];*/ // Will trigger an assert because max index is 3

    // We can make this assert work at compile-time.
    // Interested? See recipe 'Check sizes at compile-time'
    assert(sizeof(val) == sizeof(char) * array4_t::static_size);
}
```

One of the biggest advantages of `boost::array` is that it does not allocated dynamic memory and provides exactly the same performance as a usual C array. People from the C++ Standard committee also liked it, so it was accepted to the C++11 standard. Try to include the `<array>` header and check for the availability of `std::array`. `std::array` has a better support for usage with constexpr since C++17.

See also

- Boost's official documentation gives a complete list of the `Boost.Array` methods with a description of the method's complexity and throw behavior. It is available at the following link: http://boost.org/libs/array.
- The `boost::array` function is widely used across recipes; for example, refer to the *Binding a value as a function parameter* recipe.

Combining multiple values into one

There is a very nice present for those who like `std::pair`. Boost has a library called `Boost.Tuple`. It is just like `std::pair`, but it can also work with triples, quads, and even bigger collections of types.

Getting ready

Only basic knowledge of C++ and a standard library is required for this recipe.

How to do it…

Perform the following steps to combine multiple values into one:

1. To start working with tuples, you need to include a proper header and declare a variable:

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

boost::tuple<int, std::string> almost_a_pair(10, "Hello");
boost::tuple<int, float, double, int> quad(10, 1.0f, 10.0, 1);
```

2. Getting a specific value is implemented via the `boost::get<N>()` function, where `N` is a zero-based index of a required value:

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

void sample1() {
    const int i = boost::get<0>(almost_a_pair);
    const std::string& str = boost::get<1>(almost_a_pair);
    const double d = boost::get<2>(quad);
}
```

   The `boost::get<>` function has many overloads and is used widely across Boost. We already saw how it can be used with other libraries in the *Storing multiple chosen types in a container/variable* recipe.

3. You can construct tuples using the `boost::make_tuple()` function, which is shorter to write, because you do not need to fully qualify the tuple type:

```
#include <boost/tuple/tuple.hpp>
#include <boost/tuple/tuple_comparison.hpp>
#include <set>

void sample2() {
    // Tuple comparison operators are
    // defined in header "boost/tuple/tuple_comparison.hpp"
    // Don't forget to include it!
    std::set<boost::tuple<int, double, int> > s;
    s.insert(boost::make_tuple(1, 1.0, 2));
    s.insert(boost::make_tuple(2, 10.0, 2));
    s.insert(boost::make_tuple(3, 100.0, 2));

    // Requires C++11
    const auto t = boost::make_tuple(0, -1.0, 2);
    assert(2 == boost::get<2>(t));
    // We can make a compile time assert for type
    // of t. Interested? See chapter 'Compile time tricks'
}
```

4. Another function that makes life easier is `boost::tie()`. It works almost as `make_tuple`, but adds a nonconst reference for each of the passed types. Such a tuple can be used to get values to a variable from another tuple. It can be better understood from the following example:

```
#include <boost/tuple/tuple.hpp>
#include <cassert>
```

```
void sample3() {
    boost::tuple<int, float, double, int> quad(10, 1.0f, 10.0, 1);
    int i;
    float f;
    double d;
    int i2;

    // Passing values from 'quad' variables
    // to variables 'i', 'f', 'd', 'i2'.
    boost::tie(i, f, d, i2) = quad;
    assert(i == 10);
    assert(i2 == 1);
}
```

How it works…

Some readers may wonder why we need a tuple when we can always write our own structures with better names; for example, instead of writing `boost::tuple<int, std::string>`, we can create a structure:

```
struct id_name_pair {
    int id;
    std::string name;
};
```

Well, this structure is definitely clearer than `boost::tuple<int, std::string>`. The main idea behind the tuple's library is to simplify template programming.

There’s more…

A tuple works as fast as `std::pair` (it does not allocate memory on a heap and has no virtual functions). The C++ committee found this class to be very useful and it was included in the standard library. You can find it in a C++11 compatible implementation in the header file `<tuple>` (don’t forget to replace all the `boost::` namespaces with `std::`).

The standard library version of tuple must have multiple micro optimizations and typically provides a slightly better user experience. However, there is no guarantee on the order of construction of tuple elements, so, if you need a tuple that constructs its elements starting from the first, you have to use the `boost::tuple`:

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

template <int I>
struct printer {
    printer() { std::cout << I; }
};

int main() {
    // Outputs 012
    boost::tuple<printer<0>, printer<1>, printer<2> > t;
}
```

The current Boost implementation of a tuple does not use variadic templates, does not support rvalue references, does not support C++17 structured bindings, and is not usable with constexpr.

See also

- Boost's official documentation contains more examples, information about performance, and abilities of `Boost.Tuple`. It is available at the link http://boost.org/libs/tuple.
- The *Converting all tuple elements to a string* recipe in Chapter 8, *Metaprogramming*, shows some advanced usages of tuples.

Binding and reordering function parameters

If you work with the standard library a lot and use the `<algorithm>` header, you definitely write a lot of functional objects. In C++14, you can use generic lambdas for that. In C++11, you only have non generic lambdas. In the earlier versions of the C++ standard, you can construct functional objects using adapter functions such as `bind1st`, `bind2nd`, `ptr_fun`, `mem_fun`, `mem_fun_ref`, or you can write them by hand (because adapter functions look scary). Here is some good news: `Boost.Bind` can be used instead of ugly adapter functions, and it provides a more human-readable syntax.

Getting ready

A knowledge of standard library functions and algorithms will be helpful.

How to do it…

Let's see some examples of the usage of `Boost.Bind` along with C++11 lambda classes:

1. All the samples require the following headers:

```
// Contains boost::bind and placeholders.
#include <boost/bind.hpp>

// Utility stuff required by samples.
#include <boost/array.hpp>
#include <algorithm>
#include <functional>
#include <string>
#include <cassert>
```

2. Count values greater than 5 as shown in the following code:

```
void sample1() {
    const boost::array<int, 12> v = {{
        1, 2, 3, 4, 5, 6, 7, 100, 99, 98, 97, 96
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](int x) { return 5 < x; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<int>(), 5, _1)
    );
    assert(count0 == count1);
}
```

3. This is how we may count empty strings:

```
void sample2() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](const std::string& s) { return s.empty(); }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(&std::string::empty, _1)
    );
    assert(count0 == count1);
}
```

4. Now, let's count strings with a length less than `5`:

```
void sample3() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [](const std::string& s) {  return s.size() < 5; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(
            std::less<std::size_t>(),
            boost::bind(&std::string::size, _1),
            5
        )
    );
    assert(count0 == count1);
}
```

5. Compare the strings:

```
void sample4() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
    std::string s(
        "Expensive copy constructor is called when binding"
    );

    const std::size_t count0 = std::count_if(v.begin(), v.end(),
        [&s](const std::string& x) {  return x < s; }
    );
    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<std::string>(), _1, s)
    );
    assert(count0 == count1);
}
```

How it works…

The `boost::bind` function returns a functional object that stores a copy of bound values and a copy of the original functional object. When the actual call to `operator()` is performed, the stored parameters are passed to the original functional object along with the parameters passed at the time of call.

There's more…

Take a look at the previous examples. When we are binding values, we copy a value into a functional object. For some classes this operation is expensive. Is there a way to bypass copying?

Yes, there is! `Boost.Ref` library will help us here! It contains two functions, `boost::ref()` and `boost::cref()`, the first of which allows us to pass a parameter as a reference, and the second one passes the parameter as a constant reference. The `ref()` and `cref()` functions just construct an object of type `reference_wrapper<T>` or `reference_wrapper<const T>`, which is implicitly convertible to a reference type. Let's change our last examples:

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

void sample5() {
    const boost::array<std::string, 3> v = {{
        "We ", "are", " the champions!"
    }};
    std::string s(
        "Expensive copy constructor is NOT called when binding"
    );

    const std::size_t count1 = std::count_if(v.begin(), v.end(),
        boost::bind(std::less<std::string>(), _1, boost::cref(s))
    );
    // ...
}
```

You can also reorder, ignore, and duplicate function parameters using `bind`:

```
void sample6() {
    const auto twice = boost::bind(std::plus<int>(), _1, _1);
    assert(twice(2) == 4);

    const auto minus_from_second = boost::bind(std::minus<int>(), _2, _1);
    assert(minus_from_second(2, 4) == 2);

    const auto sum_second_and_third = boost::bind(
        std::plus<int>(), _2, _3
    );
    assert(sum_second_and_third(10, 20, 30) == 50);
}
```

The functions `ref` , `cref`, and `bind` are accepted to the C++11 standard and are defined in the `<functional>` header in the `std::` namespace. All these functions do not dynamically allocate memory and do not use virtual functions. The objects returned by them are easy to optimize for a good compiler.

Standard library implementations of those functions may have additional optimizations to reduce compilation time or just compiler-specific optimizations. You may use the standard library versions of `bind`, `ref`, `cref` functions with any Boost library or even mix Boost and standard library versions.

If you are using the C++14 compiler, then use generic lambdas instead of `std::bind` and `boost::bind`, as they are less obscure and simpler to understand. C++17 lambdas are usable with constexpr, unlike `std::bind` and `boost::bind`.

See also

The official documentation contains many more examples and a description of advanced features at http://boost.org/libs/bind.

Getting a human-readable type name

There is often a need to get a readable type name at runtime:

```
#include <iostream>
#include <typeinfo>

template <class T>
void do_something(const T& x) {
    if (x == 0) {
        std::cout << "Error: x == 0. T is " << typeid(T).name()
        << std::endl;
    }
    // ...
}
```

However, the example from earlier is not very portable. It does not work when RTTI is disabled, and it does not always produce a nice human-readable name. On some platforms, code from earlier will output just `i` or `d`.

Things get worse if we need a type name without stripping the `const`, `volatile`, and references:

```
void sample1() {
    auto&& x = 42;
    std::cout << "x is "
              << typeid(decltype(x)).name()
              << std::endl;
}
```

Unfortunately, the preceding code outputs `int` in the best case, which is not what we were expecting.

Getting ready

Basic knowledge of C++ is required for this recipe.

How to do it

In the first case, we need a human-readable type name without qualifiers. The `Boost.TypeIndex` library will help us out:

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

template <class T>
void do_something_again(const T& x) {
    if (x == 0) {
        std::cout << "x == 0. T is " << boost::typeindex::type_id<T>()
                  << std::endl;
    }
    // ...
}
```

In the second case, we need to keep the qualifiers, so we need to call a slightly different function from the same library:

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

void sample2() {
    auto&& x = 42;
    std::cout << "x is "
              << boost::typeindex::type_id_with_cvr<decltype(x)>()
              << std::endl;
}
```

How it works...

The `Boost.TypeIndex` library has a lot of workarounds for different compilers and knows the most efficient way to produce a human-readable name for the type. If you provide a type as a template parameter, the library guarantees that all the possible type related computations will be performed at compile time and code will work even if RTTI is disabled.

`cvr` in `boost::typeindex::type_id_with_cvr` stands for `const`, `volatile`, and reference. That makes sure that the type won't be decayed.

There’s more…

All the `boost::typeindex::type_id*` functions return instances of `boost::typeindex::type_index`. It is very close to `std::type_index`; however, it additionally, it has a `raw_name()` method for getting a raw type name, and `pretty_name()` for getting human-readable type name.

Even in C++17, `std::type_index` and `std::type_info` return platform-specific type names representations that are rather hard to decode or use portably.

Unlike the standard library’s `typeid()`, some classes from `Boost.TypeIndex` are usable with constexpr. It means that you can get a textual representation of your type at compile time if you use a specific `boost::typeindex::ctti_type_index` class.

Users can invent their own RTTI implementations using the `Boost.TypeIndex` library. This could be useful for embedded developers and for applications that require extremely efficient RTTI tuned for particular types.

See also

Documentation on advanced features and more examples are available at http://boost.org/libs/type_index.

Using the C++11 move emulation

One of the greatest features of the C++11 standard is rvalue references. This feature allows us to modify temporary objects, stealing resources from them. As you can guess, the C++03 standard has no rvalue references, but using the `Boost.Move` library, you can write a portable code that emulates them.

Getting ready

It is highly recommended that you are at least familiar with the basics of C++11 rvalue references.

How to do it…

1. Imagine that you have a class with multiple fields, some of which are standard library containers:

```
namespace other {
    class characteristics{};
}

struct person_info {
    std::string name_;
    std::string second_name_;
    other::characteristics characteristic_;
    // ...
};
```

2. It is time to add the move assignment and move constructors to it! Just remember that in the C++03 standard library, containers have neither move operators nor move constructors.
3. The correct implementation of the move assignment is the same move constructing an object and swapping it with `this`. The correct implementation of the move constructor is close to the default construct and `swap`. So, let's start with the `swap` member function:

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

void person_info::swap(person_info& rhs) {
    name_.swap(rhs.name_);
    second_name_.swap(rhs.second_name_);
    boost::swap(characteristic_, rhs.characteristic_);
}
```

4. Now, put the following macro in the `private` section:

```
    BOOST_COPYABLE_AND_MOVABLE(person_info)
```

5. Write a copy constructor.
6. Write a copy assignment, taking the parameter as: `BOOST_COPY_ASSIGN_REF(person_info)`.

7. Write a `move` constructor and a move assignment, taking the parameter as `BOOST_RV_REF(person_info)`:

```
    struct person_info {
        // Fields declared here
        // ...
    private:
        BOOST_COPYABLE_AND_MOVABLE(person_info)
    public:
        // For the simplicity of example we will assume that
        // person_info default constructor and swap are very
        // fast/cheap to call.
        person_info();

        person_info(const person_info& p)
            : name_(p.name_)
            , second_name_(p.second_name_)
            , characteristic_(p.characteristic_)
        {}

        person_info(BOOST_RV_REF(person_info) person) {
            swap(person);
        }

        person_info& operator=(BOOST_COPY_ASSIGN_REF(person_info) person) {
            person_info tmp(person);
            swap(tmp);
            return *this;
        }

        person_info& operator=(BOOST_RV_REF(person_info) person) {
            person_info tmp(boost::move(person));
            swap(tmp);
            return *this;
        }

        void swap(person_info& rhs);
    };
```

8. Now, we have a portable fast implementation of the move assignment and move construction operators of the `person_info` class.

How it works…

Here is an example of how the move assignment can be used:

```
int main() {
    person_info vasya;
    vasya.name_ = "Vasya";
    vasya.second_name_ = "Snow";

    person_info new_vasya(boost::move(vasya));
    assert(new_vasya.name_ == "Vasya");
    assert(new_vasya.second_name_ == "Snow");
    assert(vasya.name_.empty());
    assert(vasya.second_name_.empty());

    vasya = boost::move(new_vasya);
    assert(vasya.name_ == "Vasya");
    assert(vasya.second_name_ == "Snow");
    assert(new_vasya.name_.empty());
    assert(new_vasya.second_name_.empty());
}
```

The `Boost.Move` library is implemented in a very efficient way. When the C++11 compiler is used, all the macros for rvalues emulation are expanded to C++11-specific features otherwise (on C++03 compilers), rvalues are emulated.

There's more…

Have you noticed the `boost::swap` call? It is a really helpful utility function, which first searches for a `swap` function in the namespace of a variable (in our example, it's namespace `other::`), and if there is no matching swap function, it uses the `std::swap`.

See also

- More information about emulation implementation can be found on the Boost website and in the sources of the `Boost.Move` library at http://boost.org/libs/move.
- The `Boost.Utility` library is the one that contains `boost::swap`, and it has many useful functions and classes. Refer to http://boost.org/libs/utility for its documentation.
- The *Initializing a base class by the member of derived* recipe in Chapter 2, *Managing Resources.*
- The *Making a noncopyable class* recipe.
- In the *Making a noncopyable but movable class* recipe, there is more info about `Boost.Move` and some examples on how we can use the movable objects in containers in a portable and efficient way.

Making a noncopyable class

You have almost certainly encountered certain situations, where a class owns some resources that must not be copied for technical reasons:

```
class descriptor_owner {
    void* descriptor_;

public:
    explicit descriptor_owner(const char* params);

    ~descriptor_owner() {
        system_api_free_descriptor(descriptor_);
    }
};
```

The C++ compiler in the preceding example generates a copy constructor and an assignment operator, so the potential user of the `descriptor_owner` class will be able to create the following awful things:

```
void i_am_bad() {
    descriptor_owner d1("O_o");
    descriptor_owner d2("^_^");

    // Descriptor of d2 was not correctly freed
    d2 = d1;

    // destructor of d2 will free the descriptor
    // destructor of d1 will try to free already freed descriptor
}
```

Getting ready

Only very basic knowledge of C++ is required for this recipe.

How to do it…

To avoid such situations, the `boost::noncopyable` class was invented. If you derive your own class from it, the copy constructor and assignment operator won't be generated by the C++ compiler:

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

class descriptor_owner_fixed : private boost::noncopyable {
    // ...
```

Now, the user won't be able to do bad things:

```
void i_am_good() {
    descriptor_owner_fixed d1("O_o");
    descriptor_owner_fixed d2("^_^");

    // Won't compile
    d2 = d1;

    // Won't compile either
    descriptor_owner_fixed d3(d1);
}
```

How it works…

A refined reader will note that we can achieve exactly the same result by:

- Making a copy constructor and an assignment operator of `descriptor_owning_fixed` private
- Defining them without actual implementation
- Explicitly deleting them using C++11 syntax `= delete;`

Yes, you are correct. Depending on the abilities of your compiler, `boost::noncopyable` class chooses the best way to make the class noncopyable.

`boost::noncopyable` also serves as a good documentation for your class. It never raises questions such as “Is the copy constructor body defined elsewhere?” or “Does it have a nonstandard copy constructor (with a non-const referenced parameter)?”

See also

- The *Making a noncopyable, but movable class* recipe will give you ideas on how to allow unique owning of a resource in C++03 by moving it
- You may find a lot of helpful functions and classes in the `Boost.Core` library's official documentation at http://boost.org/libs/core
- The *Initializing a base class by the member of derived* recipe in Chapter 2, *Managing Resources*
- The *Using C++11 move emulation* recipe

Making a noncopyable but movable class

Now, imagine the following situation: we have a resource that cannot be copied, which should be correctly freed in a destructor, and we want to return it from a function:

```
descriptor_owner construct_descriptor()
{
    return descriptor_owner("Construct using this string");
}
```

Actually, you can work around such situations using the `swap` method:

```
void construct_descriptor1(descriptor_owner& ret)
{
    descriptor_owner("Construct using this string").swap(ret);
}
```

However, such a workaround does not allow us to use `descriptor_owner` in containers. By the way, it looks awful!

Getting ready

It is highly recommended that you are at least familiar with the basics of C++11 rvalue references. Reading the *Using C++11 move emulation* recipe is also recommended.

How to do it…

Those readers who use C++11, already know about the move-only classes (like `std::unique_ptr` or `std::thread`). Using such an approach, we can make a move-only `descriptor_owner` class:

```
class descriptor_owner1 {
    void* descriptor_;

public:
    descriptor_owner1()
        : descriptor_(nullptr)
    {}

    explicit descriptor_owner1(const char* param);

    descriptor_owner1(descriptor_owner1&& param)
        : descriptor_(param.descriptor_)
    {
        param.descriptor_ = nullptr;
    }

    descriptor_owner1& operator=(descriptor_owner1&& param) {
        descriptor_owner1 tmp(std::move(param));
        std::swap(descriptor_, tmp.descriptor_);
        return *this;
    }

    void clear() {
        free(descriptor_);
        descriptor_ = nullptr;
    }

    bool empty() const {
        return !descriptor_;
    }

    ~descriptor_owner1() {
        clear();
    }
};

// GCC compiles the following in C++11 and later modes.
descriptor_owner1 construct_descriptor2() {
    return descriptor_owner1("Construct using this string");
}

void foo_rv() {
    std::cout << "C++11n";
    descriptor_owner1 desc;
    desc = construct_descriptor2();
    assert(!desc.empty());
}
```

This will work only on the C++11 compatible compilers. That is the right moment for `Boost.Move`! Let's modify our example, so it can be used on C++03 compilers.

According to the documentation, to write a movable but noncopyable type in portable syntax, we need to follow these simple steps:

1. Put the `BOOST_MOVABLE_BUT_NOT_COPYABLE(classname)` macro in the `private` section:

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

    class descriptor_owner_movable {
        void* descriptor_;

        BOOST_MOVABLE_BUT_NOT_COPYABLE(descriptor_owner_movable
```

2. Write a move constructor and a move assignment, taking the parameter as `BOOST_RV_REF(classname)`:

```
    public:
        descriptor_owner_movable()
            : descriptor_(NULL)
        {}

        explicit descriptor_owner_movable(const char* param)
            : descriptor_(strdup(param))
        {}

        descriptor_owner_movable(
            BOOST_RV_REF(descriptor_owner_movable) param
        ) BOOST_NOEXCEPT
            : descriptor_(param.descriptor_)
        {
            param.descriptor_ = NULL;
        }

        descriptor_owner_movable& operator=(
            BOOST_RV_REF(descriptor_owner_movable) param) BOOST_NOEXCEPT
        {
            descriptor_owner_movable tmp(boost::move(param));
            std::swap(descriptor_, tmp.descriptor_);
            return *this;
        }

        // ...
    };

    descriptor_owner_movable construct_descriptor3() {
        return descriptor_owner_movable("Construct using this string");
    }
```

How it works…

Now, we have a movable, but non copyable, class that can be used even on C++03 compilers and in `Boost.Containers`:

```
#include <boost/container/vector.hpp>
#include <your_project/descriptor_owner_movable.h>

int main() {
    // Following code will work on C++11 and C++03 compilers
    descriptor_owner_movable movable;
    movable = construct_descriptor3();
    boost::container::vector<descriptor_owner_movable> vec;
    vec.resize(10);
    vec.push_back(construct_descriptor3());

    vec.back() = boost::move(vec.front());
}
```

Unfortunately, C++03 standard library containers still won’t be able to use it (that is why we used a vector from `Boost.Containers` in the previous example).

There's more…

If you want to use `Boost.Containers` on C++03 compilers, but standard library containers on C++11 compilers, you can do the following simple trick. Add the header file to your project with the following content:

```
// your_project/vector.hpp
// Copyright and other stuff goes here

// include guards
#ifndef YOUR_PROJECT_VECTOR_HPP
#define YOUR_PROJECT_VECTOR_HPP

// Contains BOOST_NO_CXX11_RVALUE_REFERENCES macro.
#include <boost/config.hpp>

#if !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
// We do have rvalues
#include <vector>

namespace your_project_namespace {
  using std::vector;
} // your_project_namespace

#else
// We do NOT have rvalues
#include <boost/container/vector.hpp>

namespace your_project_namespace {
  using boost::container::vector;
} // your_project_namespace

#endif // !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
#endif // YOUR_PROJECT_VECTOR_HPP
```

Now, you can include `<your_project/vector.hpp>` and use a vector from the namespace `your_project_namespace`:

```
int main() {
    your_project_namespace::vector<descriptor_owner_movable> v;
    v.resize(10);
    v.push_back(construct_descriptor3());
    v.back() = boost::move(v.front());
}
```

However, beware of compiler and standard library implementation-specific issues! For example, this code will compile on GCC 4.7 in C++11 mode only if you mark the move constructor, destructor, and move assignment operators with `noexcept` or `BOOST_NOECEPT`.

See also

- The *Reducing code size and increasing performance of user-defined type in C++11* recipe in Chapter 10, *Gathering Platform and Compiler Information*, provides more info on `noexcept` and `BOOST_NOEXCEPT`.
- More information about `Boost.Move` can be found on Boost's website http://boost.org/libs/move.

Using C++14 and C++11 algorithms

C++11 has a bunch of new cool algorithms in `<algorithm>` header. C++14 has even more algorithms. If you're stuck with the pre-C++11 compiler, you have to write those from scratch. For example, if you wish to output characters from 65 to 125 code points, you have to write the following code on a pre-C++11 compiler:

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

boost::array<unsigned char, 60> chars_65_125_pre11() {
    boost::array<unsigned char, 60> res;

    const unsigned char offset = 65;
    for (std::size_t i = 0; i < res.size(); ++i) {
        res[i] = i + offset;
    }

    return res;
}
```

Getting ready

Basic knowledge of C++ is required for this recipe along with basic knowledge of `Boost.Array` library.

How to do it…

The `Boost.Algorithm` library has all the new C++11 and C++14 algorithms. Using it, you can rewrite the previous example in the following manner:

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/array.hpp>

boost::array<unsigned char, 60> chars_65_125() {
    boost::array<unsigned char, 60> res;
    boost::algorithm::iota(res.begin(), res.end(), 65);
    return res;
}
```

How it works…

As you are probably aware, `Boost.Algorithm` has a header file for each algorithm. Just include the header file and use the required function.

There's more…

It's boring to have a library that just implements algorithms from C++ standard. That's not innovative; that's not the Boost way! That's why you can find in `Boost.Algorithm`, functions that are not part of C++. Here, for example, is a function that converts input into hexadecimal representation:

```
#include <boost/algorithm/hex.hpp>
#include <iterator>
#include <iostream>

void to_hex_test1() {
    const std::string data = "Hello word";
    boost::algorithm::hex(
        data.begin(), data.end(),
        std::ostream_iterator<char>(std::cout)
    );
}
```

The preceding code outputs the following:

```
48656C6C6F20776F7264
```

What's more interesting, all the functions have additional overloads that accept a range as a first parameter instead of two iterators. **Range** is a concept from **Ranges TS**. Arrays and containers that have `.begin()` and `.end()` functions satisfy the range concept. With that knowledge the previous example could be shortened:

```
#include <boost/algorithm/hex.hpp>
#include <iterator>
#include <iostream>

void to_hex_test2() {
    const std::string data = "Hello word";
    boost::algorithm::hex(
        data,
        std::ostream_iterator<char>(std::cout)
    );
}
```

C++17 will have searching algorithms from `Boost.Algorithm`. `Boost.Algorithm` library will be soon extended with new algorithms and C++20 features like constexpr usable algorithms. Keep an eye on that library, as some day, it may get an out-of-the-box solution for a problem that you're dealing with.

See also

- Official documentation for `Boost.Algorithm` contains a full list of functions and short descriptions for them at http://boost.org/libs/algorithm
- Experiment with new algorithms online: http://apolukhin.github.io/Boost-Cookbook

Managing Resources

In this chapter, we will cover the following topics:

- Managing local pointers to classes that do not leave scope
- Reference counting of pointers to classes used across functions
- Managing local pointers to arrays that do not leave scope
- Reference counting of pointers to arrays used across functions
- Storing any functional objects in a variable
- Passing function pointer in a variable
- Passing C++11 lambda functions in a variable
- Containers of pointers
- Do it at scope exit!
- Initializing the base class by the member of the derived class

Introduction

In this chapter, we’ll continue to deal with datatypes, introduced by the Boost libraries, mostly focusing on working with the pointers. We’ll see how to easily manage resources, how to use a datatype capable of storing any functional objects, functions, and lambda expressions. After reading this chapter, your code will become more reliable and memory leaks will become history.

Managing local pointers to classes that do not leave scope

Sometimes, we are required to dynamically allocate memory and construct a class in that memory. That’s where the troubles start. Take a look at the following code:

```
bool foo1() {
    foo_class* p = new foo_class("Some data");

    const bool something_else_happened = some_function1(*p);
    if (something_else_happened) {
        delete p;
        return false;
    }

    some_function2(p);

    delete p;
    return true;
}
```

This code looks correct at first glance. But, what if `some_function1()` or `some_function2()` throws an exception? In that case, `p` won’t be deleted. Let’s fix it in the following way:

```
bool foo2() {
    foo_class* p = new foo_class("Some data");
    try {
        const bool something_else_happened = some_function1(*p);
        if (something_else_happened) {
            delete p;
            return false;
        }
       some_function2(p);
    } catch (...) {
        delete p;
        throw;
    }
    delete p;
    return true;
}
```

Now the code is correct, but ugly and hard to read. Can we do better than this?

Getting started

Basic knowledge of C++ and code behavior during exceptions is required.

How to do it…

Just take a look at the `Boost.SmartPtr` library. There is a `boost::scoped_ptr` class that may help you out:

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

bool foo3() {
    const boost::scoped_ptr<foo_class> p(new foo_class("Some data"));

    const bool something_else_happened = some_function1(*p);
    if (something_else_happened) {
       return false;
    }
    some_function2(p.get());
    return true;
}
```

Now, there is no chance that the resource will leak and the source code is much clearer.

If you have control over `some_function2(foo_class*)`, you may wish to rewrite it to take a reference to `foo_class` instead of a pointer. An interface with references is more intuitive than an interface with pointers unless you have a special agreement in your company that output parameters are taken only by pointer.

By the way, `Boost.Move` also has a `boost::movelib::unique_ptr` that you may use instead of `boost::scoped_ptr`:

```
#include <boost/move/make_unique.hpp>

bool foo3_1() {
    const boost::movelib::unique_ptr<foo_class> p
        = boost::movelib::make_unique<foo_class>("Some data");

    const bool something_else_happened = some_function1(*p);
    if (something_else_happened) {
       return false;
    }
    some_function2(p.get());
    return true;
}
```

How it works...

`boost::scoped_ptr<T>` and `boost::movelib::unique_ptr` are typical **RAII** classes. When an exception is thrown or a variable goes out of scope, the stack is unwound and the destructor is called. In the destructor, `scoped_ptr<T>` and `unique_ptr<T>` call `delete` for a pointer that they store. Because both of these classes by default call `delete`, it is safe to hold a `derived` class by a pointer to the `base` class if destructor of the base class is virtual:

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

struct base {
    virtual ~base(){}
};

class derived: public base {
    std::string str_;

public:
    explicit derived(const char* str)
        : str_(str)
    {}

    ~derived() /*override*/ {
        std::cout << "str == " << str_ << '\n';
    }
};

void base_and_derived() {
    const boost::movelib::unique_ptr<base> p1(
        boost::movelib::make_unique<derived>("unique_ptr")
    );

    const boost::scoped_ptr<base> p2(
        new derived("scoped_ptr")
    );
}
```

Running the `base_and_derived()` function will produce the following output:

```
    str == scoped_ptr
    str == unique_ptr
```

In C++, destructors for objects are called in the reverse construction order. That's why the destructor of `scoped_ptr` was called before the destructor of `unique_ptr`.

The `boost::scoped_ptr<T>` class template is neither copyable nor movable. The `boost::movelib::unique_ptr` class is a move-only class, and it uses move emulation on pre-C++11 compilers. Both classes store a pointer to the resource they own and do not require `T` to be a complete type (`T` can be forward declared).

Some compilers do not warn when an incomplete type is deleted, which may lead to errors that are hard to detect. Fortunately, that's not the case for Boost classes that have specific compile-time asserts for such cases. That makes `scoped_ptr` and `unique_ptr` perfect for implementing the **Pimpl** idiom:

```
// In header file:
struct public_interface {
    // ...
private:
    struct impl; // Forward declaration.
    boost::movelib::unique_ptr<impl> impl_;
};
```

There’s more…

Those classes are extremely fast. Compiler optimizes the code that uses `scoped_ptr` and `unique_ptr` to the machine code, which involve no additional overhead compared to the handwritten manual memory management code.

C++11 has a `std::unique_ptr<T, D>` class that uniquely owns the resource and behaves exactly like `boost::movelib::unique_ptr<T, D>`.

The C++ standard library has no `boost::scoped_ptr<T>`, but you could use `const std::unique_ptr<T>` instead. The only difference is that `boost::scoped_ptr<T>` still can call `reset()` unlike `const std::unique_ptr<T>`.

See also

- The documentation of the `Boost.SmartPtr` library contains lots of examples and other useful information about all the smart pointers' classes. You can read about them at http://boost.org/libs/smart_ptr.
- The `Boost.Move` docs may help you out if you use move emulation with `boost::movelib::unique_ptr` http://boost.org/libs/move.

Reference counting of pointers to classes used across functions

Imagine that you have some dynamically allocated structure containing data, and you want to process it in different threads of execution. The code to do this is as follows:

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

void process1(const foo_class* p);
void process2(const foo_class* p);
void process3(const foo_class* p);

void foo1() {
    while (foo_class* p = get_data()) // C way
    {
        // There will be too many threads soon, see
        // recipe 'Parallel execution of different tasks'
        // for a good way to avoid uncontrolled growth of threads
        boost::thread(boost::bind(&process1, p))
            .detach();
        boost::thread(boost::bind(&process2, p))
            .detach();
        boost::thread(boost::bind(&process3, p))
            .detach();


        // delete p; Oops!!!!
    }
}
```

We cannot deallocate `p` at the end of the `while` loop because it still can be used by threads that run `process` functions. These `process` functions cannot delete `p`, because they do not know that other threads are not using it anymore.

Getting ready

This recipe uses the `Boost.Thread` library, which is not a header-only library. Your program has to link against the `boost_thread`, `boost_chrono`, and `boost_system` libraries. Make sure that you do understand the concept of threads before reading further. Refer to the *See also* section for references on recipes that describe threads.

You also need some basic knowledge on `boost::bind` or `std::bind`, which are almost the same.

How to do it…

As you may have guessed, there is a class in Boost (and C++11) that may help you to deal with the problem. It is called `boost::shared_ptr`. It can be used as follows:

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

void process_sp1(const boost::shared_ptr<foo_class>& p);
void process_sp2(const boost::shared_ptr<foo_class>& p);
void process_sp3(const boost::shared_ptr<foo_class>& p);

void foo2() {
    typedef boost::shared_ptr<foo_class> ptr_t;
    ptr_t p;
    while (p = ptr_t(get_data())) // C way
    {
        boost::thread(boost::bind(&process_sp1, p))
            .detach();
        boost::thread(boost::bind(&process_sp2, p))
            .detach();
        boost::thread(boost::bind(&process_sp3, p))
            .detach();

        // no need to anything
    }
}
```

Another example of this is as follows:

```
#include <string>
#include <boost/smart_ptr/make_shared.hpp>

void process_str1(boost::shared_ptr<std::string> p);
void process_str2(const boost::shared_ptr<std::string>& p);

void foo3() {
    boost::shared_ptr<std::string> ps = boost::make_shared<std::string>(
        "Guess why make_shared<std::string> "
        "is faster than shared_ptr<std::string> "
        "ps(new std::string('this string'))"
    );

    boost::thread(boost::bind(&process_str1, ps))
            .detach();
    boost::thread(boost::bind(&process_str2, ps))
            .detach();
}
```

How it works…

The shared_ptr class has an atomic reference counter inside. When you copy it, the reference counter is incremented, and when its destructor is called, the reference counter is decremented. When the count of references equals to zero, delete is called for the object pointed by shred_ptr.

Now, let's find out what's happening in the case of boost::thread (boost::bind(&process_sp1, p)). The function process_sp1 takes a parameter as a reference, so why is it not deallocated when we do get out of the while loop? The answer is simple. The functional object returned by bind() contains a copy of the shared pointer, and that means the data pointed by p won't be deallocated until the functional object is destroyed. The functional object is copied into the thread and is kept alive until the threads executes.

Getting back to boost::make_shared, let's take a look at shared_ptr<std::string> ps(new int(0)). In this case, we have two calls to new:

- While constructing a pointer to an integer via new int(0)
- While constructing a shared_ptr class internal reference counter that is allocated on the heap

Use make_shared<T> to have only one call to new. A make_shared<T> allocates a single chunk of memory and constructs an atomic counter and the T object in that chunk.

There's more…

The atomic reference counter guarantees correct behavior of `shared_ptr` across the threads, but you must remember that atomic operations are not as fast as non-atomic. `shared_ptr` touches the atomic variable on assignments, copy constructions, and on destruction of a non moved away `shared_ptr`. It means that on C++11 compatible compilers, you may reduce atomic operation's count using move constructions and move assignments where possible. Just use `shared_ptr<T> p1(std::move(p))` if you are not going to use the `p` variable any more. If you are not going to modify the pointed value, it is recommended to make it `const`. Just add `const` to the template parameter of the smart pointer, and the compiler will make sure that you do not modify memory:

```
void process_cstr1(boost::shared_ptr<const std::string> p);
void process_cstr2(const boost::shared_ptr<const std::string>& p);

void foo3_const() {
    boost::shared_ptr<const std::string> ps
        = boost::make_shared<const std::string>(
            "Some immutable string"
        );

    boost::thread(boost::bind(&process_cstr1, ps))
            .detach();
    boost::thread(boost::bind(&process_cstr2, ps))
            .detach();

    // *ps = "qwe"; // Compile time error, string is const!
}
```

Confused with `const`? Here's a mapping of smart pointer constness to simple pointer constness:

`shared_ptr<T>`	`T*`
`shared_ptr<const T>`	`const T*`
`const shared_ptr<T>`	`T* const`
`const shared_ptr<const T>`	`const T* const`

The `shared_ptr` calls and the `make_shared` function are a part of C++11, and they are declared in the header `<memory>` in the `std::` namespace. They have almost the same characteristics as Boost versions.

See also

- Refer to Chapter 5, *Multithreading*, for more information about `Boost.Thread` and atomic operations.
- Refer to the *Binding and reordering function parameters* recipe in Chapter 1, *Starting to Write Your Application*, for more information about `Boost.Bind`.
- Refer to the Chapter 3, recipe *Converting smart pointers* for some information on how to convert `shared_ptr<U>` to `shared_ptr<T>`.
- The documentation of the `Boost.SmartPtr` library contains lots of examples and other useful information about all the smart pointer's classes. Refer to the link http://boost.org/libs/smart_ptr to read about them.

Managing pointers to arrays that do not leave scope

We already saw how to manage pointers to a resource in the *Managing pointers to classes that do not leave scope* recipe. But, when we deal with arrays, we need to call `delete[]` instead of a simple `delete`. Otherwise, there will be a memory leak. Take a look at the following code:

```
void may_throw1(char ch);
void may_throw2(const char* buffer);

void foo() {
    // we cannot allocate 10MB of memory on stack,
    // so we allocate it on heap
    char* buffer = new char[1024 * 1024 * 10];

    // Oops. Here comes some code, that may throw.
    // It was a bad idea to use raw pointer as the memory may leak!!
    may_throw1(buffer[0]);
    may_throw2(buffer);

    delete[] buffer;
}
```

Getting ready

The knowledge of C++ exceptions and templates are required for this recipe.

How to do it…

The `Boost.SmartPointer` library has not only the `scoped_ptr<>` class, but also a `scoped_array<>` class:

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

void foo_fixed() {
    // We allocate array on heap
    boost::scoped_array<char> buffer(new char[1024 * 1024 * 10]);

    // Here comes some code, that may throw,
    // but now exception won't cause a memory leak
    may_throw1(buffer[0]);
    may_throw2(buffer.get());

    // destructor of 'buffer' variable will call delete[]
}
```

The `Boost.Move` library's `boost::movelib::unique_ptr<>` class can work with arrays too. You just need to indicate that it is storing an array by providing `[]` at the end of the template parameter:

```
#include <boost/move/make_unique.hpp>

void foo_fixed2() {
    // We allocate array on heap
    const boost::movelib::unique_ptr<char[]> buffer
        = boost::movelib::make_unique<char[]>(1024 * 1024 * 10);

    // Here comes some code, that may throw,
    // but now exception won't cause a memory leak
    may_throw1(buffer[0]);
    may_throw2(buffer.get());

    // destructor of 'buffer' variable will call delete[]
}
```

How it works…

`scoped_array<>` works exactly like a `scoped_ptr<>` class, but calls `delete[]` in the destructor instead of `delete`. The `unique_ptr<T[]>` does the same thing.

There's more…

The `scoped_array<>` class has same guarantees and design as `scoped_ptr<>`. It has neither additional memory allocations nor virtual function's call. It cannot be copied and also is not a part of C++11. `std::unique_ptr<T[]>` is part of the C++11 and has the same guarantees and performance as the `boost::movelib::unique_ptr<T[]>` class.

Actually, `make_unique<char[]>(1024)` *is not the same as* `new char[1024]`*, because the first one does value initialization and the second one does the default initialization. The equivalent function for default-initialization is* `boost::movelib::make_unique_definit`*.*

Note that Boost version could also work on pre-C++11 compilers and even emulates rvalues on them, making `boost::movelib::unique_ptr` a move only type. If your standard library does not provide `std::make_unique` then `Boost.SmartPtr` may help you out. It provides `boost::make_unique` that returns a `std::unique_ptr` in header `boost/smart_ptr/make_unique.hpp`. It also provides `boost::make_unique_noinit` for default-initialization in the same header. C++17 does not have a `make_unique_noinit` function.

Using `new` *for memory allocation and manual memory management in C++ is a bad habit. Use* `make_unique` *and* `make_shared` *functions wherever possible.*

See also

- The documentation of the `Boost.SmartPtr` library contains lots of examples and other useful information about all the smart pointer's classes, you can read about them at http://boost.org/libs/smart_ptr.
- The `Boost.Move` docs may help you out if you wish to use move emulation with `boost::movelib::unique_ptr`, read about them at http://boost.org/libs/move.

Reference counting of pointers to arrays used across functions

We continue coping with pointers, and our next task is to reference count an array. Let's take a look at the program that gets some data from the stream and processes it in different threads. The code to do this is as follows:

```
#include <cstring>
#include <boost/thread.hpp>
#include <boost/bind.hpp>

void do_process(const char* data, std::size_t size);

void do_process_in_background(const char* data, std::size_t size) {
    // We need to copy data, because we do not know,
    // when it will be deallocated by the caller.
    char* data_cpy = new char[size];
    std::memcpy(data_cpy, data, size);

    // Starting thread of execution to process data.
    boost::thread(boost::bind(&do_process, data_cpy, size))
            .detach();
    boost::thread(boost::bind(&do_process, data_cpy, size))
            .detach();

    // Oops!!! We cannot delete[] data_cpy, because
    // do_process() function may still work with it.
}
```

Just the same problem that occurred in the *Reference counting of pointers to classes used across functions* recipe.

Getting ready

This recipe uses the `Boost.Thread` library, which is not a header-only library, so your program will need to link against the `boost_thread`, `boost_chrono`, and `boost_system` libraries. Make sure that you do understand the concept of threads before reading further.

You'll also need some basic knowledge on `boost::bind` or `std::bind`, which is almost the same.

How to do it…

There are four solutions. The main difference between them is of type and construction of the `data_cpy` variable. All those solutions do exactly the same things that are described in the beginning of this recipe, but without memory leaks. The solutions are as follows:

- The first solution works well for cases when array size is known at compile time:

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

template <std::size_t Size>
void do_process_shared(const boost::shared_ptr<char[Size]>& data);

template <std::size_t Size>
void do_process_in_background_v1(const char* data) {
    // Same speed as in 'First solution'.
    boost::shared_ptr<char[Size]> data_cpy
        = boost::make_shared<char[Size]>();
    std::memcpy(data_cpy.get(), data, Size);

    // Starting threads of execution to process data.
    boost::thread(boost::bind(&do_process_shared<Size>, data_cpy))
            .detach();
    boost::thread(boost::bind(&do_process_shared<Size>, data_cpy))
            .detach();

    // data_cpy destructor will deallocate data when
    // reference count is zero.
}
```

- Since Boost 1.53, `shared_ptr` itself can take care of arrays of unknown bound. The second solution:

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

void do_process_shared_ptr(
        const boost::shared_ptr<char[]>& data,
        std::size_t size);

void do_process_in_background_v2(const char* data, std::size_t size) {
    // Faster than 'First solution'.
    boost::shared_ptr<char[]> data_cpy = boost::make_shared<char[]>(size);
    std::memcpy(data_cpy.get(), data, size);

    // Starting threads of execution to process data.
    boost::thread(boost::bind(&do_process_shared_ptr, data_cpy, size))
            .detach();
    boost::thread(boost::bind(&do_process_shared_ptr, data_cpy, size))
            .detach();

    // data_cpy destructor will deallocate data when
    // reference count is zero.
}
```

- The third solution:

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

void do_process_shared_ptr2(
        const boost::shared_ptr<char>& data,
        std::size_t size);
```

```
void do_process_in_background_v3(const char* data, std::size_t size) {
    // Same speed as in 'First solution'.
    boost::shared_ptr<char> data_cpy(
                new char[size],
                boost::checked_array_deleter<char>()
    );
    std::memcpy(data_cpy.get(), data, size);

    // Starting threads of execution to process data.
    boost::thread(boost::bind(&do_process_shared_ptr2, data_cpy, size))
            .detach();
    boost::thread(boost::bind(&do_process_shared_ptr2, data_cpy, size))
            .detach();

    // data_cpy destructor will deallocate data when
    // reference count is zero.
}
```

- The last solution is deprecated since Boost 1.65, but may be of use in case of antique Boost versions:

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

void do_process_shared_array(
        const boost::shared_array<char>& data,
        std::size_t size);

void do_process_in_background_v4(const char* data, std::size_t size) {
    // We need to copy data, because we do not know, when it will be
    // deallocated by the caller.
    boost::shared_array<char> data_cpy(new char[size]);
    std::memcpy(data_cpy.get(), data, size);

    // Starting threads of execution to process data.
    boost::thread(
        boost::bind(&do_process_shared_array, data_cpy, size)
    ).detach();
    boost::thread(
        boost::bind(&do_process_shared_array, data_cpy, size)
    ).detach();

    // No need to call delete[] for data_cpy, because
    // data_cpy destructor will deallocate data when
    // reference count is zero.
}
```

How it works…

In all examples, **smart pointer** classes count references and call `delete[]` for a pointer when the count of references becomes equal to zero. The first and second examples are simple. In the third example, we provide a custom `deleter` object for a `shared` pointer. The `deleter` object of a smart pointer is called when the smart pointer decides to free resources. When smart pointer is constructed without explicit `deleter`, the default `deleter` is constructed that calls `delete` or `delete[]` depending on the template type of the smart pointer.

There's more…

The fourth solution is the most conservative, because prior to Boost 1.53 the functionality of the second solution was not implemented in `shared_ptr`. The first and second solutions are the fastest ones as they use only one memory allocation call. The third solution can be used with older versions of Boost and with C++11 standard library's `std::shared_ptr<>` (just don't forget to change `boost::checked_array_deleter<T>()` to `std::default_delete<T[]>()`).

Actually, `boost::make_shared<char[]>(size)` *is not the same as* `new char[size]`*, because it involves value-initialization of all elements. The equivalent function for default-initialization is* `boost::make_shared_noinit`*.*

Beware! C++11 and C++14 versions of `std::shared_ptr` cannot work with arrays! Only since C++17 `std::shared_ptr<T[]>` must work properly. If you are planning to write portable code, consider using `boost::shared_ptr`, `boost::shared_array`, or explicitly pass a `deleter` to `std::shared_ptr`.

`boost::shared_ptr<T[]>`*,* `boost::shared_array`*, and C++17* `std::shared_ptr<T[]>` *have* `operator[](std::size_t index)` *that allows you to access elements of shared array by index.* `boost::shared_ptr<T>` *and* `std::shared_ptr<T>` *with custom* `deleter` *have no* `operator[]`*, which makes them less useful.*

See also

The documentation of the `Boost.SmartPtr` library contains lots of examples and other useful information about all the smart pointers classes. You can read about it at http://boost.org/libs/smart_ptr.

Storing any functional objects in a variable

Consider the situation when you are developing a library that has its API declared in the header files and implementation in the source files. This library shall have a function that accepts any functional objects. Take a look at the following code:

```
// making a typedef for function pointer accepting int
// and returning nothing
typedef void (*func_t)(int);

// Function that accepts pointer to function and
// calls accepted function for each integer that it has.
// It cannot work with functional objects :(
void process_integers(func_t f);

// Functional object
class int_processor {
   const int min_;
   const int max_;
   bool& triggered_;

public:
    int_processor(int min, int max, bool& triggered)
        : min_(min)
        , max_(max)
        , triggered_(triggered)
    {}

    void operator()(int i) const {
        if (i < min_ || i > max_) {
            triggered_ = true;
        }
    }
};
```

How do you change the `process_integers` function to accept any functional objects?

Getting ready

Reading the *Storing any value in container/variable* recipe in Chapter 1, *Starting to Write Your Application*, is recommended before starting off with this recipe.

How to do it…

There is a solution and it is called a `Boost.Function` library. It allows you to store any function, a member function, or a functional object if its signature is a match to the one described in the template argument:

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

typedef boost::function<void(int)> fobject_t;

// Now this function may accept functional objects
void process_integers(const fobject_t& f);

int main() {
    bool is_triggered = false;
    int_processor fo(0, 200, is_triggered);
    process_integers(fo);
    assert(is_triggered);
}
```

How it works…

The `fobject_t` object stores in itself functional objects and erases their exact type. It is safe to use the `boost::function` for stateful objects:

```
bool g_is_triggered = false;
void set_functional_object(fobject_t& f) {
    // Local variable
    int_processor fo( 100, 200, g_is_triggered);

    f = fo;
    // now 'f' holds a copy of 'fo'

    // 'fo' leavs scope and will be destroyed,
    // but it's OK to use 'f' in outer scope.
}
```

Does `boost::function` recall the `boost::any` class? That’s because it uses the same technique **type erasure** for storing any function objects.

There’s more…

The `boost::function` class has a default constructor and has an empty state. Checking for an empty/default constructed state can be done like this:

```
void foo(const fobject_t& f) {
    // boost::function is convertible to bool
    if (f) {
        // we have value in 'f'
        // ...
    } else {
        // 'f' is empty
        // ...
    }
}
```

The `Boost.Function` library has an insane amount of optimizations. It may store small functional objects without additional memory allocations and has optimized move assignment operators. It is accepted as a part of C++11 standard library and is defined in the `<functional>` header in the `std::` namespace.

`boost::function` uses an RTTI for objects stored inside it. If you disable RTTI, the library will continue to work, but will dramatically increase the size of a compiled binary.

See also

- The official documentation of `Boost.Function` contains more examples, performance measures, and class reference documentation. Refer to the link http://boost.org/libs/function to read about it.
- The *Passing function pointer in a variable* recipe.
- The *Passing C++11 lambda functions in a variable* recipe.

Passing function pointer in a variable

We are continuing with the previous example, and now we want to pass a pointer to a function in our `process_integers()` method. Shall we add an overload for just function pointers, or is there a more elegant way?

Getting ready

This recipe is continuing the previous one. You must read the previous recipe first.

How to do it…

Nothing needs to be done as `boost::function<>` is also constructible from the function pointers:

```
void my_ints_function(int i);

int main() {
    process_integers(&my_ints_function);
}
```

How it works…

A pointer to `my_ints_function` will be stored inside the `boost::function` class, and calls to `boost::function` will be forwarded to the stored pointer.

There’s more…

The `Boost.Function` library provides a good performance for pointers to functions, and it will not allocate memory on heap. Standard library `std::function` is also effective in storing function pointers. Since Boost 1.58, the `Boost.Function` library can store functions and functional objects that have call signature with rvalue references:

```
boost::function<int(std::string&&)> f = &something;
f(std::string("Hello")); // Works
```

See also

- The official documentation of `Boost.Function` contains more examples, performance measures, and class reference documentation. Follow http://boost.org/libs/function to read about it.
- The *Passing C++11 lambda functions in a variable* recipe.

Passing C++11 lambda functions in a variable

We are continuing with the previous example, and now we want to use a lambda function with our `process_integers()` method.

Getting ready

This recipe is continuing the series of the previous two. You must read them first. You will also need a C++11 compatible compiler or at least a compiler with C++11 lambda support.

How to do it…

Nothing needs to be done as `boost::function<>` is also usable with lambda functions of any difficulty:

```
#include <deque>
//#include "your_project/process_integers.h"

void sample() {
    // lambda function with no parameters that does nothing
    process_integers([](int /*i*/){});

    // lambda function that stores a reference
    std::deque<int> ints;
    process_integers([&ints](int i){
        ints.push_back(i);
    });

    // lambda function that modifies its content
    std::size_t match_count = 0;
    process_integers([ints, &match_count](int i) mutable {
        if (ints.front() == i) {
           ++ match_count;
        }
        ints.pop_front();
    });
}
```

There's more…

Performance of the lambda function storage in `Boost.Functional` is the same as in other cases. While a functional object produced by the lambda expression is small enough to fit in an instance of `boost::function`, no dynamic memory allocation is performed. Calling an object stored in `boost::function` is close to the speed of calling a function by a pointer. Copying `boost::function` allocates heap memory only if initial `boost::function` has a stored object that does not fit in it without allocation. Moving instances does not allocate and deallocate memory.

Remember that `boost::function` implies an optimization barrier for the compiler. It means that:

```
std::for_each(v.begin(), v.end(), [](int& v) { v += 10; });
```

Is usually better optimized by the compiler than:

```
const boost::function<void(int&)> f0(
    [](int& v) { v += 10; }
);
std::for_each(v.begin(), v.end(), f0);
```

This is why you should try to avoid using `Boost.Function` when its usage is not really required. In some cases, the C++11 `auto` keyword can be handy instead:

```
const auto f1 = [](int& v) { v += 10; };
std::for_each(v.begin(), v.end(), f1);
```

See also

Additional information about performance and `Boost.Function` can be found on the official documentation page at http://www.boost.org/libs/function.

Containers of pointers

There are such cases when we need to store pointers in the container. The examples are: storing polymorphic data in containers, forcing fast copy of data in containers, and strict exception requirements for operations with data in containers. In such cases, C++ programmer has the following choices:

- Store pointers in containers and take care of their destructions using `delete`:

```
#include <set>
#include <algorithm>
#include <cassert>

template <class T>
struct ptr_cmp {
    template <class T1>
    bool operator()(const T1& v1, const T1& v2) const {
        return operator ()(*v1, *v2);
    }

    bool operator()(const T& v1, const T& v2) const {
        return std::less<T>()(v1, v2);
    }
};

void example1() {
    std::set<int*, ptr_cmp<int> > s;
    s.insert(new int(1));
    s.insert(new int(0));

    // ...
    assert(**s.begin() == 0);
    // ...

    // Oops! Any exception in the above code leads to
    // memory leak.

    // Deallocating resources.
    std::for_each(s.begin(), s.end(), [](int* p) { delete p; });
}
```

  Such an approach is error prone and requires a lot of writing

- Store C++11 smart pointers in containers:

```
#include <memory>
#include <set>

void example2_cpp11() {
    typedef std::unique_ptr<int> int_uptr_t;
    std::set<int_uptr_t, ptr_cmp<int> > s;
    s.insert(int_uptr_t(new int(1)));
    s.insert(int_uptr_t(new int(0)));

    // ...
    assert(**s.begin() == 0);
    // ...

    // Resources will be deallocated by unique_ptr<>.
}
```

  This solution is a good one, but it cannot be used in C++03, and you still need to write a comparator functional object.

C++14 has a `std::make_unique` function for construction of `std::uniue_ptr`s. Using it instead of `new` is a good coding style!

- Use `Boost.SmartPtr` in the container:

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

void example3() {
    typedef boost::shared_ptr<int> int_sptr_t;
    std::set<int_sptr_t, ptr_cmp<int> > s;
    s.insert(boost::make_shared<int>(1));
    s.insert(boost::make_shared<int>(0));

    // ...
    assert(**s.begin() == 0);
    // ...

    // Resources will be deallocated by shared_ptr<>.
}
```

This solution is portable, but it adds performance penalties (atomic counter requires additional memory, and its increments/decrements are not as fast as non-atomic operations) and you still need to write comparators.

Getting ready

Knowledge of standard library containers is required for better understanding of this recipe.

How to do it…

The `Boost.PointerContainer` library provides a good and portable solution:

```
#include <boost/ptr_container/ptr_set.hpp>

void correct_impl() {
    boost::ptr_set<int> s;
    s.insert(new int(1));
    s.insert(new int(0));

    // ...
    assert(*s.begin() == 0);
    // ...

    // Resources will be deallocated by container itself.
}
```

How it works…

The `Boost.PointerContainer` library has classes `ptr_array`, `ptr_vector`, `ptr_set`, `ptr_multimap`, and others. Those classes deallocate pointers as required and simplify access to data pointed by the pointer (no need for additional dereference in `assert(*s.begin() == 0);`).

There’s more…

When we want to clone some data, we need to define a freestanding function `T*new_clone(const T& r)` in the namespace of the object to be cloned. Moreover, you may use the default `T* new_clone(const T& r)` implementation if you include the header file `<boost/ptr_container/clone_allocator.hpp>`, as shown in the following code:

```
#include <boost/ptr_container/clone_allocator.hpp>
#include <boost/ptr_container/ptr_vector.hpp>
#include <cassert>

void theres_more_example() {
    // Creating vector of 10 elements with values 100
    boost::ptr_vector<int> v;
    int value = 100;
    v.resize(10, &value); // Beware! No ownership of pointer!

    assert(v.size() == 10);
    assert(v.back() == 100);
}
```

C++ standard library has no pointer containers, but you can achieve the same functionality using a container of `std::unique_ptr`. By the way, since Boost 1.58, there is a `boost::movelib::unique_ptr` class that is usable in C++03. You can mix it with containers from the `Boost.Container` library to have C++11 functionality for storing pointers:

```
#include <boost/container/set.hpp>
#include <boost/move/make_unique.hpp>
#include <cassert>

void example2_cpp03() {
    typedef boost::movelib::unique_ptr<int> int_uptr_t;
    boost::container::set<int_uptr_t, ptr_cmp<int> > s;
    s.insert(boost::movelib::make_unique<int>(1));
    s.insert(boost::movelib::make_unique<int>(0));
    // ...
    assert(**s.begin() == 0);
}
```

Not all the developers know the Boost libraries well. It is more developer-friendly to use functions and classes that have C++ standard library alternatives, as the developers usually are more aware of the standard library features. So if there’s no big difference for you, use `Boost.Container` with `boost::movelib::unique_ptr`.

See also

- The official documentation contains detailed reference for each class, follow the link http://boost.org/libs/ptr_container to read about it.
- The first four recipes of this chapter give you some examples about smart pointers' usage.
- Multiple recipes in Chapter 9, *Containers* describe the `Boost.Container` library features. Take a look at that chapter for cool, useful, and fast containers.

Do it at scope exit!

If you were dealing with languages, such as Java, C#, or Delphi, you obviously were using the `try {} finally{}` construction. Let me briefly describe to you what do these language constructions do.

When a program leaves the current scope via return or exception, code in the `finally` block is executed. This mechanism is used as a replacement for the RAII pattern:

```
// Some pseudo code (suspiciously similar to Java code)
try {
    FileWriter f = new FileWriter("example_file.txt");
    // Some code that may throw or return
    // ...
} finally {
    // Whatever happened in scope, this code will be executed
    // and file will be correctly closed
    if (f != null) {
        f.close()
    }
}
```

Is there a way to do such a thing in C++?

Getting ready

Basic C++ knowledge is required for this recipe. Knowledge of code behavior during thrown exception will be appreciated.

How to do it…

C++ uses the RAII pattern instead of `try {} finally{}` construction. The `Boost.ScopeExit` library was designed to allow user definition of RAII wrappers right in the function body:

```
#include <boost/scope_exit.hpp>
#include <cstdlib>
#include <cstdio>
#include <cassert>

int main() {
    std::FILE* f = std::fopen("example_file.txt", "w");
    assert(f);

    BOOST_SCOPE_EXIT(f) {
    // Whatever happened in outer scope, this code will be executed
    // and file will be correctly closed.
        std::fclose(f);
    } BOOST_SCOPE_EXIT_END

    // Some code that may throw or return.
    // ...
}
```

How it works…

The `f` is passed by the value via `BOOST_SCOPE_EXIT(f)`. When the program leaves the scope of execution, the code between `BOOST_SCOPE_EXIT(f) {` and `} BOOST_SCOPE_EXIT_END` is be executed. If we wish to pass the value by reference, use the `&` symbol in the `BOOST_SCOPE_EXIT` macro. If we wish to pass multiple values, just separate them by commas.

Passing references to a pointer does not work well on some compilers. The `BOOST_SCOPE_EXIT(&f)` macro cannot be compiled there, which is why we do not capture it by reference in the example.

There’s more…

To capture this inside a member function, we shall use a special symbol `this_`:

```
class theres_more_example {
public:
    void close(std::FILE*);

    void theres_more_example_func() {
        std::FILE* f = 0;
        BOOST_SCOPE_EXIT(f, this_) { // Capturing `this` as 'this_'
            this_->close(f);
        } BOOST_SCOPE_EXIT_END
    }
};
```

The `Boost.ScopeExit` library allocates no additional memory on heap and does not use virtual functions. Use the default syntax and do not define `BOOST_SCOPE_EXIT_CONFIG_USE_LAMBDAS` because otherwise scope exit will be implemented using `boost::function`, which may allocate additional memory and imply an optimization barrier. You may achieve close to the `BOOST_SCOPE_EXIT` results using `boost::movelib::unique_ptr` or `std::unique_ptr` by specifying a custom `deleter`:

```
#include <boost/move/unique_ptr.hpp>
#include <cstdio>

void unique_ptr_example() {
    boost::movelib::unique_ptr<std::FILE, int(*)(std::FILE*)> f(
        std::fopen("example_file.txt", "w"), // open file
        &std::fclose  // specific deleter
    );
    // ...
}
```

If you write two or more similar bodies for `BOOST_SCOPE_EXIT`, then it’s time to think about some refactoring and moving the code to a fully functional RAII class.

See also

The official documentation contains more examples and use cases. You can read about it at http://boost.org/libs/scope_exit.

Initializing the base class by the member of the derived class

Let’s take a look at the following example. We have some base class that has virtual functions and must be initialized with reference to the `std::ostream` object:

```
#include <boost/noncopyable.hpp>
#include <sstream>

class tasks_processor: boost::noncopyable {
    std::ostream& log_;

protected:
    virtual void do_process() = 0;

public:
    explicit tasks_processor(std::ostream& log)
        : log_(log)
    {}

    void process() {
        log_ << "Starting data processing";
        do_process();
    }
};
```

We also have a derived class that has a `std::ostream` object and implements the `do_process()` function:

```
class fake_tasks_processor: public tasks_processor {
    std::ostringstream logger_;

    virtual void do_process() {
        logger_ << "Fake processor processed!";
    }

public:
    fake_tasks_processor()
        : tasks_processor(logger_) // Oops! logger_ does not exist here
        , logger_()
    {}
};
```

This is not a very common case in programming, but when such mistakes happen, it is not always simple to get the idea of bypassing it. Some people try to bypass it by changing the order of `logger_` and the base type initialization:

```
    fake_tasks_processor()
        : logger_() // Oops! It is still constructed AFTER tasks_processor.
        , tasks_processor(logger_)
    {}
```

It won’t work as per expectations because direct base classes are initialized before non-static data members, regardless of the order of the member initializers.

Getting started

Basic knowledge of C++ is required for this recipe.

How to do it…

The Boost.Utility library provides a quick solution for such cases. Solution is called the boost::base_from_member template. To use it, you need to carry out the following steps:

1. Include the base_from_member.hpp header:

```
#include <boost/utility/base_from_member.hpp>
```

2. Derive your class from boost::base_from_member<T> where T is a type that must be initialized before the base (take care about the order of the base classes; boost::base_from_member<T> must be placed before the class that uses T):

```
class fake_tasks_processor_fixed
    : boost::base_from_member<std::ostringstream>
    , public tasks_processor
```

3. Correctly, write the constructor as follows:

```
{
    typedef boost::base_from_member<std::ostringstream> logger_t;

    virtual void do_process() {
        logger_t::member << "Fake processor processed!";
    }

public:
    fake_tasks_processor_fixed()
        : logger_t()
        , tasks_processor(logger_t::member)
    {}
};
```

How it works…

Direct base classes are initialized before non-static data members and in declaration order as they appear in the base-specifier list. If we need to initialize base class `B` with *something*, we need to make that *something* a part of a base class `A` that is declared before `B`. In other words, `boost::base_from_member` is just a simple class that holds its template parameter as a non-static data member:

```
template < typename MemberType, int UniqueID = 0 >
class base_from_member {
protected:
    MemberType  member;
    //      Constructors go there...
};
```

There’s more…

As you may see, `base_from_member` has an integer as a second template argument. This is done for cases when we need multiple `base_from_member` classes of the same type:

```
class fake_tasks_processor2
    : boost::base_from_member<std::ostringstream, 0>
    , boost::base_from_member<std::ostringstream, 1>
    , public tasks_processor
{
    typedef boost::base_from_member<std::ostringstream, 0> logger0_t;
    typedef boost::base_from_member<std::ostringstream, 1> logger1_t;

    virtual void do_process() {
        logger0_t::member << "0: Fake processor2 processed!";
        logger1_t::member << "1: Fake processor2 processed!";
    }

public:
    fake_tasks_processor2()
        : logger0_t()
        , logger1_t()
        , tasks_processor(logger0_t::member)
    {}
};
```

The `boost::base_from_member` class neither applies additional dynamic memory allocations nor has virtual functions. The current implementation does support **perfect forwarding** and **variadic templates** if your compiler supports them.
C++ standard library has no `base_from_member`.

See also

- The `Boost.Utility` library contains many helpful classes and functions; documentation for getting more information about it is at http://boost.org/libs/utility
- The *Making a noncopyable class* recipe in Chapter 1, *Starting to Write Your Application*, contains more examples of classes from `Boost.Utility`
- Also, the *Using C++11 move emulation* recipe in Chapter 1, *Starting to Write Your Application*, contains more examples of classes from `Boost.Utility`

Converting and Casting

In this chapter, we will cover:

- Converting strings to numbers
- Converting numbers to strings
- Converting numbers to numbers
- Converting user-defined types to/from strings
- Converting smart pointers
- Casting polymorphic objects
- Parsing simple input
- Parsing complex input

Introduction

Now, that we know some of the basic Boost types, it is time to get to know data-converting functions. In this chapter, we’ll see how to convert strings, numbers, pointers, and user-defined types to each other, how to safely cast polymorphic types, and how to write small and big parsers right inside the C++ source files.

Converting strings to numbers

Converting strings to numbers in C++ makes a lot of people depressed because of their inefficiency and user unfriendliness. See how string `100` can be converted to `int`:

```
#include <sstream>

void sample1() {
    std::istringstream iss("100");
    int i;
    iss >> i;

    // ...
}
```

It is better not to think, how many unnecessary operations, virtual function calls, atomic operations, and memory allocations occurred during the conversion from earlier. By the way, we do not need the `iss` variable any more, but it will be alive until the end of scope.

C methods are not much better:

```
#include <cstdlib>

void sample2() {
    char * end;
    const int i = std::strtol ("100", &end, 10);

    // ...
}
```

Did it convert the whole value to `int` or stopped somewhere in the middle? To understand that we must check the content of the `end` variable. After that we'll have a useless `end` variable getting in the way until the end of scope. And we wanted an `int`, but `strtol` returns `long int`. Did the converted value fit in `int`?

Getting ready

Only basic knowledge of C++ and standard library is required for this recipe.

How to do it…

There is a library in Boost, which will help you cope with depressing difficulty of string to number conversions. It is called `Boost.LexicalCast` and consists of a `boost::bad_lexical_cast` exception class and a few `boost::lexical_cast` and `boost::conversion::try_lexical_convert` functions:

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

void sample3() {
    const int i = boost::lexical_cast<int>("100");
    // ...
}
```

It can be used even for non-zero-terminated strings:

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

void sample4() {
    char chars[] = {'x', '1', '0', '0', 'y' };
    const int i = boost::lexical_cast<int>(chars + 1, 3);
    assert(i == 100);
}
```

How it works…

The `boost::lexical_cast` function accepts string as input and converts it to the type specified in triangular brackets. The `boost::lexical_cast` function will even check bounds for you:

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

void sample5() {
    try {
        // short usually may not store values greater than 32767
        const short s = boost::lexical_cast<short>("1000000");
        assert(false); // Must not reach this line.
    } catch (const boost::bad_lexical_cast& e) {
        std::cout << e.what() << '\n';
    }
}
```

The preceding code outputs:

```
bad lexical cast: source type value could not be interpreted as target.
```

It also checks for the correct syntax of input and throws an exception if input is wrong:

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

void sample6() {
    try {
        const int i = boost::lexical_cast<int>("This is not a number!");
        assert(false); // Must not reach this line.
    } catch (const boost::bad_lexical_cast& /*e*/) {}
}
```

Since Boost 1.56, there is a `boost::conversion::try_lexical_convert` function that reports errors by return code. It could be useful in performance critical places where bad input could often occur:

```
#include <boost/lexical_cast.hpp>
#include <cassert>

void sample7() {
    int i = 0;
    const bool ok = boost::conversion::try_lexical_convert("Bad stuff", i);
    assert(!ok);
}
```

There's more…

`lexical_cast`, just like all `std::stringstream` classes, uses `std::locale` and can convert localized numbers, but also has an impressive set of optimizations for **C locale** and for locales without number groupings:

```
#include <boost/lexical_cast.hpp>
#include <locale>

void sample8() {
    try {
        std::locale::global(std::locale("ru_RU.UTF8"));
        // In Russia coma sign is used as a decimal separator.
        float f = boost::lexical_cast<float>("1,0");
        assert(f < 1.01 && f > 0.99);
        std::locale::global(std::locale::classic()); // Restoring C locale
    } catch (const std::runtime_error&) { /* locale is not supported */ }
}
```

The C++ standard library has no `lexical_cast`, but since C++17 has `std::from_chars` functions that could be used to create amazingly fast generic converters. Note that those converters do not use locales at all, so they have slightly different functionality because of that. `std::from_chars` functions were designed to not allocate memory, do not throw exceptions, and have no atomic or some other heavy operations.

See also

- Refer to the *Converting numbers to strings* recipe for information about the `boost::lexical_cast` performance.
- The official documentation of `Boost.LexicalCast` contains some examples, performance measures, and answers to frequently asked questions. It is available at http://boost.org/libs/lexical_cast.

Converting numbers to strings

In this recipe, we will continue discussing the lexical conversions, but now we will be converting numbers to strings using `Boost.LexicalCast`. As usual, `boost::lexical_cast` will provide a very simple way to convert the data.

Getting ready

Only basic knowledge of C++ and a standard library is required for this recipe.

How to do it…

Let’s convert integer `100` to `std::string` using `boost::lexical_cast`:

```
#include <cassert>
#include <boost/lexical_cast.hpp>

void lexical_cast_example() {
    const std::string s = boost::lexical_cast<std::string>(100);
    assert(s == "100");
}
```

Compare it against the traditional C++ conversion method:

```
#include <cassert>
#include <sstream>

void cpp_convert_example() {
    std::stringstream ss;  // Slow/heavy default constructor.
    ss << 100;
    std::string s;
    ss >> s;

    // Variable 'ss' will dangle all the way, till the end
    // of scope. Multiple virtual methods and heavy
    // operations were called during the conversion.
    assert(s == "100");
}
```

And against the C conversion method:

```
#include <cassert>
#include <cstdlib>

void c_convert_example() {
     char buffer[100];
     std::sprintf(buffer, "%i", 100);

     // You will need an unsigned long long int type to
     // count how many times errors were made in 'printf'
     // like functions all around the world. 'printf'
     // functions are a constant security threat!

     // But wait, we still need to construct a std::string.
     const std::string s = buffer;
     // Now we have a 'buffer' variable that is not used.

     assert(s == "100");
}
```

How it works…

The `boost::lexical_cast` function may also accept numbers as input and convert them to a string type specified as the template parameter (in triangular brackets). This is pretty close to what we have done in the previous recipe.

There's more…

A careful reader will notice that in case of `lexical_cast`, we have an additional call to string copy the constructor and that such a call degrades performance. It is true, but only for old or bad compilers. Modern compilers implement a **Named Return Value Optimization** (**NRVO**), which eliminates the unnecessary calls to copy constructors and destructor. Even if the C++11-compatible compilers don't detect NRVO, they use a move constructor of `std::string`, which is fast and efficient. The *Performance* section of the `Boost.LexicalCast` documentation shows the conversion speed on different compilers for different types. In most cases, `lexical_cast` is faster than the `std::stringstream` and `printf` functions.

If `boost::array` or `std::array` is passed to `boost::lexical_cast` as the output parameter type, less dynamic memory allocations will occur (or there will be no memory allocations at all, it depends on the `std::locale` implementation).
C++11 has `std::to_string` and `std::to_wstring` functions that are declared in the `<string>` header. Those functions use locales and have behavior very close to `boost::lexical_cast<std::string>` and `boost::lexical_cast<std::wstring>`, respectively. C++17 has `std::to_chars` functions that convert numbers to char arrays amazingly fast. `std::to_chars` do not allocate memory, do not throw exception, and may report errors using error codes. If you need really fast conversion functions that do not use locales, then use `std::to_chars`.

See also

- Boost's official documentation contains tables that compare the `lexical_cast` performance against other conversion approaches. In most cases, `lexical_cast` is faster than other approaches http://boost.org/libs/lexical_cast.
- The *Converting strings to numbers* recipe.
- The *Converting user defined types to/from strings* recipe.

Converting numbers to numbers

You might remember situations where you were writing the following code:

```
void some_function(unsigned short param);
int foo();

void do_something() {
    // Some compilers may warn, that int is being converted to
    // unsigned short and that there is a possibility of loosing
    // data.
    some_function(foo());
}
```

Usually, programmers just ignore such warnings by implicitly casting to the `unsigned short` datatype, as demonstrated in the following code snippet:

```
// Warning suppressed.
some_function(
    static_cast<unsigned short>(foo())
);
```

But, what if `foo()` returns numbers that do not fit into `unsigned short`? This leads to hard detectable errors. Such errors may exist in code for years before they get caught and fixed. Take a look at the `foo()` definition:

```
// Returns -1 if error occurred.
int foo() {
    if (some_extremely_rare_condition()) {
        return -1;
    } else if (another_extremely_rare_condition()) {
        return 1000000;
    }
    return 42;
}
```

Getting ready

Only basic knowledge of C++ is required for this recipe.

How to do it…

The library `Boost.NumericConversion` provides a solution for such cases. Just replace `static_cast` with `boost::numeric_cast`. The latter will throw an exception when the source value cannot be stored in target:

```
#include <boost/numeric/conversion/cast.hpp>

void correct_implementation() {
    // 100% correct.
    some_function(
        boost::numeric_cast<unsigned short>(foo())
    );
}

void test_function() {
    for (unsigned int i = 0; i < 100; ++i) {
        try {
            correct_implementation();
        } catch (const boost::numeric::bad_numeric_cast& e) {
            std::cout << '#' << i << ' ' << e.what() << std::endl;
        }
    }
}
```

Now, if we run `test_function()`, it will output the following:

```
#47 bad numeric conversion: negative overflow
#58 bad numeric conversion: positive overflow
```

We can even detect specific overflow types:

```
void test_function1() {
   for (unsigned int i = 0; i < 100; ++i) {
       try {
           correct_implementation();
       } catch (const boost::numeric::positive_overflow& e) {
           // Do something specific for positive overflow.
           std::cout << "POS OVERFLOW in #" << i << ' '
                     << e.what() << std::endl;
       } catch (const boost::numeric::negative_overflow& e) {
           // Do something specific for negative overflow.
           std::cout <<"NEG OVERFLOW in #" << i << ' '
                     << e.what() << std::endl;
       }
   }
}
```

The `test_function1()` function will output the following:

```
NEG OVERFLOW in #47 bad numeric conversion: negative overflow
POS OVERFLOW in #59 bad numeric conversion: positive overflow
```

How it works…

`boost::numeric_cast` checks if the value of the input parameter fits into the new type without losing data and throws an exception if something is lost during conversion.

The `Boost.NumericConversion` library has a very fast implementation. It can do a lot of work at compile time, for example, when converting to types of a wider range, the source will be converted to target type by just via `static_cast`.

There’s more…

The `boost::numeric_cast` function is implemented via `boost::numeric::converter`, which can be tuned to use different overflow, range checking, and rounding policies. But usually, `numeric_cast` is just what you need.

Here is a small example that demonstrates how to make our own overflow handler for `boost::numeric::cast`:

```
template <class SourceT, class TargetT>
struct mythrow_overflow_handler {
    void operator() (boost::numeric::range_check_result r) {
        if (r != boost::numeric::cInRange) {
            throw std::logic_error("Not in range!");
        }
    }
};

template <class TargetT, class SourceT>
TargetT my_numeric_cast(const SourceT& in) {
    typedef boost::numeric::conversion_traits<
        TargetT, SourceT
    > conv_traits;
    typedef boost::numeric::converter <
        TargetT,
        SourceT,
        conv_traits, // default conversion traits
        mythrow_overflow_handler<SourceT, TargetT> // !!!
    > converter;

    return converter::convert(in);
}
```

Here’s how to use our custom converter:

```
void example_with_my_numeric_cast() {
    short v = 0;
    try {
        v = my_numeric_cast<short>(100000);
    } catch (const std::logic_error& e) {
        std::cout << "It works! " << e.what() << std::endl;
    }
}
```

The preceding code outputs the following:

```
It works! Not in range!
```

Even C++17 does not have facilities for safe numeric casts. However, work in that direction is ongoing. We have all the chances to see such facilities after the year 2020.

See also

The Boost's official documentation contains detailed description of all the template parameters of numeric converter; it is available at the following link: http://boost.org/libs/numeric/conversion

Converting user-defined types to/from strings

There is a feature in `Boost.LexicalCast` that allows users to use their own types with `lexical_cast`. This feature requires from the user to write the correct `std::ostream` and `std::istream` operators for the type.

How to do it…

1. All you need is to provide `operator<<` and `operator>>` stream operators. If your class is already streamable, nothing needs to be done:

```
#include <iostream>
#include <stdexcept>

// Negative number that does not store minus sign.
class negative_number {
    unsigned short number_;

public:
    explicit negative_number(unsigned short number = 0)
        : number_(number)
    {}

    // ...
    unsigned short value_without_sign() const {
        return number_;
    }
};

inline std::ostream&
    operator<<(std::ostream& os, const negative_number& num)
{
    os << '-' << num.value_without_sign();
    return os;
}

inline std::istream& operator>>(std::istream& is, negative_number& num)
{
    char ch;
    is >> ch;
    if (ch != '-') {
        throw std::logic_error(
            "negative_number class stores ONLY negative values"
        );
    }

    unsigned short s;
    is >> s;
    num = negative_number(s);
    return is;
}
```

2. Now, we may use `boost::lexical_cast` for conversions to and from the `negative_number` class. Here's an example:

```
#include <boost/lexical_cast.hpp>
#include <boost/array.hpp>
#include <cassert>

void example1() {
    const negative_number n
        = boost::lexical_cast<negative_number>("-100");
    assert(n.value_without_sign() == 100);

    const int i = boost::lexical_cast<int>(n);
    assert(i == -100);

    typedef boost::array<char, 10> arr_t;
    const arr_t arr = boost::lexical_cast<arr_t>(n);
    assert(arr[0] == '-');
    assert(arr[1] == '1');
    assert(arr[2] == '0');
    assert(arr[3] == '0');
```

```
        assert(arr[4] == 0);
    }
```

How it works…

The `boost::lexical_cast` function can detect and use stream operators to convert user-defined types.

The `Boost.LexicalCast` library has many optimizations for basic types and they will be triggered, when user-defined type is being cast to basic type or when basic type is being cast to user-defined type.

There’s more…

The `boost::lexical_cast` function may also convert to wide character strings, but the correct `basic_istream` and `basic_ostream` operator overloads are required for that:

```
template <class CharT>
std::basic_ostream<CharT>&
    operator<<(std::basic_ostream<CharT>& os, const negative_number& num)
{
    os << static_cast<CharT>('-') << num.value_without_sign();
    return os;
}

template <class CharT>
std::basic_istream<CharT>&
    operator>>(std::basic_istream<CharT>& is, negative_number& num)
{
    CharT ch;
    is >> ch;
    if (ch != static_cast<CharT>('-')) {
        throw std::logic_error(
            "negative_number class stores ONLY negative values"
        );
    }
    unsigned short s;
    is >> s;
    num = negative_number(s);
    return is;
}

void example2() {
    const negative_number n = boost::lexical_cast<negative_number>(L"-1");
    assert(n.value_without_sign() == 1);

    typedef boost::array<wchar_t, 10> warr_t;
    const warr_t arr = boost::lexical_cast<warr_t>(n);
    assert(arr[0] == L'-');
    assert(arr[1] == L'1');
    assert(arr[2] == 0);
}
```

The `Boost.LexicalCast` library is not a part of C++. A lot of Boost libraries use it, and I hope that it will make your life easier as well.

See also

- The `Boost.LexicalCast` documentation contains some examples, performance measures, and answers to frequently asked questions; it is available at http://boost.org/libs/lexical_cast
- The *Converting strings to numbers* recipe
- The *Converting numbers to strings* recipe

Converting smart pointers

Here’s a problem:

1. You have a class named `some_class`:

```
struct base {
    virtual void some_methods() = 0;
    virtual ~base();
};

struct derived: public base {
    void some_methods() /*override*/;
    virtual void derived_method() const;

    ~derived() /*override*/;
};
```

2. You have a third-party API that returns constructed `derived` by shared pointer to `base` and accepts shared pointer to `const derived` in other functions:

```
#include <boost/shared_ptr.hpp>
boost::shared_ptr<const base> construct_derived();
void im_accepting_derived(boost::shared_ptr<const derived> p);
```

3. You have to make the following code work:

```
void trying_hard_to_pass_derived() {
    boost::shared_ptr<const base> d = construct_derived();

    // Oops! Compile time error:
    // ‘const struct base; has no member named ‘derived_method;.
    d->derived_method();

    // Oops! Compile time error:
    // could not convert ‘d; to ‘boost::shared_ptr<const derived>;.
    im_accepting_derived(d);
}
```

How do you solve the problem in a nice manner?

Getting started

Basic knowledge of C++ and smart pointers is required for this recipe.

How to do it…

The solution would be to use special casts for smart pointers. In this particular case, we need to use `dynamic_cast` functionality, so we are using `boost::dynamic_pointer_cast`:

```
void trying_hard_to_pass_derived2() {
    boost::shared_ptr<const derived> d
        = boost::dynamic_pointer_cast<const derived>(
            construct_derived()
        );

    if (!d) {
        throw std::runtime_error(
            "Failed to dynamic cast"
        );
    }

    d->derived_method();
    im_accepting_derived(d);
}
```

How it works…

The Boost library has a lot of functions for smart pointer conversions. All of them accept a smart pointer and a template parameter `T`, where `T` is the desired template type of the smart pointer. Those conversion operator mimic the behavior of built-in casts while correctly managing the reference counting and other smart pointer internals:

- `boost::static_pointer_cast<T>(p)` - does `static_cast<T*>(p.get())`
- `boost::dynamic_pointer_cast<T>(p)` - does `dynamic_cast<T*>(p.get())`
- `boost::const_pointer_cast<T>(p)` - does `const_cast<T*>(p.get())`
- `boost::reinterpret_pointer_cast<T>(p)` - does `reinterpret_cast<T*>(p.get())`

There’s more…

All the `boost::*_pointer_cast` functions can work with smart pointers from the standard library and C pointers if you include `<boost/pointer_cast.hpp>`.
In C++11, the standard library has `std::static_pointer_cast`, `std::dynamic_pointer_cast`, and `std::const_pointer_cast` defined in the `<memory>` header, however, it is only for `std::shared_ptr`.
The C++17 standard library has `std::reinterpret_pointer_cast`, but it is only for `std::shared_ptr`.

See also

- The `Boost.SmartPointer` library documentation contains more examples on pointer casts for a standard library at http://boost.org/libs/smart_ptr/pointer_cast.html
- Casts reference for `boost::shared_ptr` is available at http://boost.org/libs/smart_ptr/shared_ptr.htm
- The *Casting polymorphic objects* recipe in this chapter will show you a better way of doing dynamic casts

Casting polymorphic objects

Imagine that some programmer designed such an awful interface as follows (this is a good example of how interfaces should not be written):

```
struct object {
    virtual ~object() {}
};

struct banana: public object {
    void eat() const {}
    virtual ~banana(){}
};

struct penguin: public object {
    bool try_to_fly() const {
        return false; // penguins do not fly
    }
    virtual ~penguin(){}
};

object* try_produce_banana();
```

Our task is to make a function that eats bananas, and throws exceptions if something different came instead of banana (`try_produce_banana()` may return `nullptr`)so if we dereference a value returned by it without checking we are in danger of dereferencing a null pointer.

Getting started

Basic knowledge of C++ is required for this recipe.

How to do it…

So we need to write the following code:

```
void try_eat_banana_impl1() {
    const object* obj = try_produce_banana();
    if (!obj) {
        throw std::bad_cast();
    }
    dynamic_cast<const banana&>(*obj).eat();
}
```

Ugly, isn’t it? `Boost.Conversion` provides a slightly better solution:

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

void try_eat_banana_impl2() {
    const object* obj = try_produce_banana();
    boost::polymorphic_cast<const banana*>(obj)->eat();
}
```

How it works…

The `boost::polymorphic_cast` function just wraps around code from the first example, and that is all. It checks input for null and then tries to do a dynamic cast. Any error during those operations will throw a `std::bad_cast` exception.

There’s more…

The `Boost.Conversion` library also has a `polymorphic_downcast` function, which should be used only for downcasts that certainly shall succeed. In debug mode (when `NDEBUG` is not defined), it will check for the correct downcast using `dynamic_cast`. When `NDEBUG` is defined, the `polymorphic_downcast` function will just do a `static_cast` operation. It is a good function to use in performance-critical sections, still leaving the ability to detect errors in debug compilations.

Since Boost 1.58 there is a `boost::polymorphic_pointer_downcast` and `boost::polymorphic_pointer_cast` function in `Boost.Conversion` libraries. Those functions allow you to safely cast smart pointers and have the same characteristics as `boost::polymorphic_cast` and `boost::polymorphic_downcast`.

The C++ standard library lacks `polymorphic_cast` and `polymorphic_downcast`.

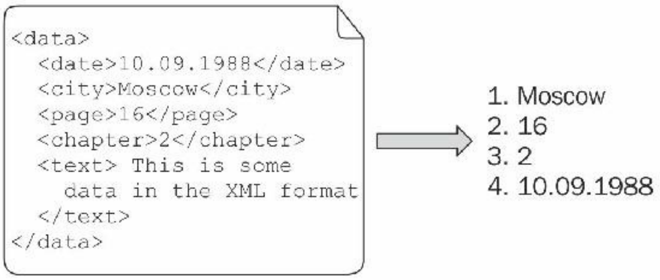
See also

- Initially, the `polymorphic_cast` idea was proposed in the book *The C++ Programming Language*, *Bjarne Stroustrup*. Refer to this book for more information and some good ideas on different topics.
- The official documentation may also be helpful; it is available at http://boost.org/libs/conversion.
- Refer to the previous recipe for more information on casting smart pointers.

Parsing simple input

It is a common task to parse a small text. Such situations are always a dilemma: shall we use some third-party professional and good tools for parsing such as Bison or ANTLR, or shall we try to write it by hand using only C++ and standard library? The third-party tools are good for handling the parsing of complex texts, and it is easy to write parsers using them, but they require additional tools for creating C++ or C code from their grammar, and add more dependencies to your project.

Handwritten parsers are usually hard to maintain, but they require nothing except a C++ compiler.



```
<data>
  <date>10.09.1988</date>
  <city>Moscow</city>
  <page>16</page>
  <chapter>2</chapter>
  <text> This is some
    data in the XML format
  </text>
</data>
```

```
1. Moscow
2. 16
3. 2
4. 10.09.1988
```

What a typical parser do

Let’s start from a very simple task to parse a date in the ISO format as follows:

```
YYYY-MM-DD
```

The following are the examples of possible input:

```
2013-03-01
2012-12-31  // (woo-hoo, it almost a new year!)
```

We will need parser’s grammar from the following link http://www.ietf.org/rfc/rfc333:

```
   date-fullyear   = 4DIGIT
   date-month      = 2DIGIT  ; 01-12
   date-mday       = 2DIGIT  ; 01-28, 01-29, 01-30, 01-31 based on
                             ; month/year
   full-date       = date-fullyear "-" date-month "-" date-mday
```

Getting ready

Make sure that you are familiar with the placeholder’s concept or read the *Reordering parameters of function* and *Binding a value as a function parameter* recipes in Chapter 1, *Starting to Write Your Application*. Basic knowledge of parsing tools would be good.

How to do it…

Let me introduce to you a `Boost.Spirit` library. It allows writing parsers (as well as lexers and generators) directly in C++ code, which are immediately executable (do not require additional tools for C++ code generation). Grammar of `Boost.Spirit` is very close to **Extended Backus-Naur Form** (**EBNF**), which is used for expressing grammar by many standards and understood by other popular parsers. Grammar at the beginning of this chapter is in EBNF:

1. We need to include the following headers:

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

2. Now, it’s time to make a `date` structure to hold the parsed data:

```
struct date {
    unsigned short year;
    unsigned short month;
    unsigned short day;
};
```

3. Let’s look at the parser (step-by-step description of how it works can be found in the next section):

```
// See recipe "Type 'reference to string'" for a better type
// than std::string for parameter 's'
date parse_date_time1(const std::string& s) {
    using boost::spirit::qi::_1;
    using boost::spirit::qi::ushort_;
    using boost::spirit::qi::char_;
    using boost::phoenix::ref;

    date res;
    const char* first = s.data();
    const char* const end = first + s.size();
    const bool success = boost::spirit::qi::parse(first, end,

        // Implementation of 'full-date' rule from EBNF grammar.
        ushort_[ ref(res.year) = _1 ] >> char_('-')
           >> ushort_[ ref(res.month) = _1 ] >> char_('-')
           >> ushort_[ ref(res.day) = _1 ]

    );

    if (!success || first != end) {
        throw std::logic_error("Parsing failed");
    }
    return res;
}
```

4. Now, we may use this parser wherever we want:

```
int main() {
    const date d = parse_date_time1("2017-12-31");
    assert(d.year == 2017);
    assert(d.month == 12);
    assert(d.day == 31);
```

```
        }
```

How it works…

This is a very simple implementation; it does not check the count of digits for numbers. Parsing occurs in the `boost::spirit::qi::parse` function. Let's simplify it a little bit, removing the actions on successful parsing:

```
const bool success = boost::spirit::qi::parse(first, end,
     ushort_ >> char_('-') >> ushort_ >> char_('-') >> ushort_
);
```

The `first` argument points to the beginning of data to parse. It must be a nonconstant variable because the `parse` function will modify it to point to the end of the parsed sequence.
The `end` argument points to the position that goes after the last element to parse. `first` and `end` shall be iterators or pointers.

The third argument to the function is a parsing rule. It looks exactly like the EBNF rule:

```
date-fullyear "-" date-month "-" date-md
```

We just replaced white spaces with the `>>` operator.

The `parse` function returns `true` on success. If we want to make sure that the whole string was successfully parsed, we need to check for the parser's return value and equality of the `end` and modified `first` iterators.

Now, we need to deal with actions on successful parse and this recipe will be over. Semantic actions in `Boost.Spirit` are written inside `[]` and they can be written using function pointers, function objects, `boost::bind`, `std::bind` (or the other `bind()` implementations), or C++11 lambda functions.

So, you could also write a rule for `YYYY` using C++11 lambda:

```
const auto y = [&res](unsigned short s) { res.year = s; };
// ...

ushort_[y] >> char_('-') >> // ...
```

You cannot put the lambda definition directly inside the `[]` because the C++ compiler will think that it's an attribute. As a workaround, you can make an `auto` variable with the lambda function in it and use that variable in parser rule description (just like it was done in the preceding code snippet).

Now, let's take a look at the month's semantic action closer:

```
ushort_[ ref(res.month) = _1 ]
```

For those who read the book from the beginning, the preceding code reminds about `boost::bind`, `boost::ref`, and placeholders. `ref(res.month)` means pass `res.month` as a modifiable reference and `_1` means the first input parameter, which would be a number (result of `ushort_` parsing).

There’s more…

Now let’s modify our parser, so it could take care of the digits count. For that purpose, we will take the `unit_parser` class template and just set up the correct parameters:

```
date parse_date_time2(const std::string& s) {
    using boost::spirit::qi::_1;
    using boost::spirit::qi::uint_parser;
    using boost::spirit::qi::char_;
    using boost::phoenix::ref;

    date res;

    // Use unsigned short as output type; require Radix 10 and
    // from 2 to 2 digits.
    uint_parser<unsigned short, 10, 2, 2> u2_;

    // Use unsigned short as output type; require Radix 10 and
    // from 4 to 4 digits.
    uint_parser<unsigned short, 10, 4, 4> u4_;

    const char* first = s.data();
    const char* const end = first + s.size();
    const bool success = boost::spirit::qi::parse(first, end,

        u4_ [ ref(res.year) = _1 ] >> char_('-')
            >> u2_ [ ref(res.month) = _1 ] >> char_('-')
            >> u2_ [ ref(res.day) = _1 ]

    );
    if (!success || first != end) {
        throw std::logic_error("Parsing failed");
    }
    return res;
}
```

Don’t worry if those examples seem complicated. The first time I was also frightened by `Boost.Spirit`, but now it really simplifies my life. You are extremely brave, if this code does not scare you.

Do not write parsers in header files because it increases compilation times of your project. Write parsers in source files and hide all the `Boost.Spirit` internals in that file. If we adjust the previous example to follow that rule then the header file would look like this:

```
// Header file.
#ifndef MY_PROJECT_PARSE_DATE_TIME
#define MY_PROJECT_PARSE_DATE_TIME

#include <string>

struct date {
    unsigned short year;
    unsigned short month;
    unsigned short day;
};

date parse_date_time2(const std::string& s);

#endif // MY_PROJECT_PARSE_DATE_TIME
```

Also take care of iterator types passed to the `boost::spirit::parse` function. The less different types of iterators you use, the less size of binary you get.

If you are now thinking that parsing date was simpler to implement by hand using standard library, you are right! But only for now. Take a look at the next recipe, it will give you more examples on `Boost.Spirit` usage and extend this example for cases, when writing parser by hand is harder than using `Boost.Spirit`.

The `Boost.Spirit` library is not a part of C++ and would not be proposed for inclusion in the nearest future. However, it works pretty well with modern C++ features, so use them if your compiler supports C++11:

```
date parse_date_time2_cxx(const std::string& s) {
    using boost::spirit::qi::uint_parser;
    using boost::spirit::qi::char_;

    date res;

    uint_parser<unsigned short, 10, 2, 2> u2_;
    uint_parser<unsigned short, 10, 4, 4> u4_;

    const auto y = [&res](unsigned short s) { res.year = s; };
    const auto m = [&res](unsigned short s) { res.month = s; };
    const auto d = [&res](unsigned short s) { res.day = s; };

    const char* first = s.data();
    const char* const end = first + s.size();
    const bool success = boost::spirit::qi::parse(first, end,
        u4_[y] >> char_('-') >> u2_[m] >> char_('-') >> u2_[d]
    );

    if (!success || first != end) {
        throw std::logic_error("Parsing failed");
    }
    return res;
}
```

See also

- The *Reordering parameters of function* recipe in Chapter 1, *Starting to Write Your Application*.
- The *Binding a value as a function parameter* recipe.
- `Boost.Spirit` is a huge header-only library. A separate book may be written about it. Feel free to use its documentation at http://boost.org/libs/spirit.

Parsing complex input

In the previous recipe, we were writing a simple parser for date. Imagine that some time has passed and the task has changed. Now, we need to write a date-time parser that supports multiple input formats and zone offsets. Our parser must understand the following inputs:

```
2012-10-20T10:00:00Z      // date time with zero zone offset
2012-10-20T10:00:00       // date time without zone offset
2012-10-20T10:00:00+09:15 // date time with zone offset
2012-10-20-09:15          // date time with zone offset
10:00:09+09:15            // time with zone offset
```

Getting ready

We’ll be using the `Boost.Spirit` library, which was described in the *Parsing simple input* recipe. Read it before getting hands on with this recipe.

How to do it…

1. Let's start by writing a date-time structure that will hold a parsed result:

```
#include <stdexcept>
#include <cassert>

struct datetime {
    enum zone_offsets_t {
        OFFSET_NOT_SET,
        OFFSET_Z,
        OFFSET_UTC_PLUS,
        OFFSET_UTC_MINUS
    };

private:
    unsigned short year_;
    unsigned short month_;
    unsigned short day_;

    unsigned short hours_;
    unsigned short minutes_;
    unsigned short seconds_;

    zone_offsets_t zone_offset_type_;
    unsigned int zone_offset_in_min_;

    static void dt_assert(bool v, const char* msg) {
        if (!v) {
            throw std::logic_error(
                "Assertion failed in datetime: " + std::string(msg)
            );
        }
    }

public:
    datetime()
        : year_(0), month_(0), day_(0)
        , hours_(0), minutes_(0), seconds_(0)
        , zone_offset_type_(OFFSET_NOT_SET), zone_offset_in_min_(0)
    {}

    // Getters: year(), month(), day(), hours(), minutes(),
    // seconds(), zone_offset_type(), zone_offset_in_min()
    // ...

    // Setters: set_year(unsigned short), set_day(unsigned short), ...
    //
    // void set_*(unsigned short val) {
    //     Some dt_assert.
    //     Setting the '*_' to 'val'.
    // }
    // ...

};
```

2. Now, let's write a function for setting the zone offset:

```
void set_zone_offset(datetime& dt, char sign, unsigned short hours
    , unsigned short minutes)
{
    dt.set_zone_offset(
        sign == '+'
        ? datetime::OFFSET_UTC_PLUS
        : datetime::OFFSET_UTC_MINUS
    );
    dt.set_zone_offset_in_min(hours * 60 + minutes);
}
```

3. Writing a parser can be split into writing a few simple parsers. We start with writing a zone-offset parser:

```
// Default includes for Boost.Spirit.
#include <boost/spirit/include/qi.hpp>
#include <boost/spirit/include/phoenix_core.hpp>
#include <boost/spirit/include/phoenix_operator.hpp>

// We'll use bind() function from Boost.Spirit,
// because it interates better with parsers.
#include <boost/spirit/include/phoenix_bind.hpp>

datetime parse_datetime(const std::string& s) {
    using boost::spirit::qi::_1;
    using boost::spirit::qi::_2;
    using boost::spirit::qi::_3;
    using boost::spirit::qi::uint_parser;
    using boost::spirit::qi::char_;
    using boost::phoenix::bind;
    using boost::phoenix::ref;

    datetime ret;

    // Use unsigned short as output type; require Radix 10 and
    // from 2 to 2 digits.
    uint_parser<unsigned short, 10, 2, 2> u2_;

    // Use unsigned short as output type; require Radix 10 and
    // from 4 to 4 digits.
    uint_parser<unsigned short, 10, 4, 4> u4_;

    boost::spirit::qi::rule<const char*, void()> timezone_parser
        = -( // unary minus means optional rule

            // Zero offset
            char_('Z')[ bind(
                &datetime::set_zone_offset, &ret, datetime::OFFSET_Z
            ) ]

            | // OR

            // Specific zone offset
            ((char_('+')|char_('-')) >> u2_ >> ':' >> u2_) [
                bind(&set_zone_offset, ref(ret), _1, _2, _3)
            ]
        );
```

4. Let’s finish our example by writing the remaining parsers:

```
    boost::spirit::qi::rule<const char*, void()> date_parser =
           u4_ [ bind(&datetime::set_year, &ret, _1) ] >> '-'
        >> u2_ [ bind(&datetime::set_month, &ret, _1) ] >> '-'
        >> u2_ [ bind(&datetime::set_day, &ret, _1) ];

    boost::spirit::qi::rule<const char*, void()> time_parser =
            u2_ [ bind(&datetime::set_hours, &ret, _1) ] >> ':'
         >> u2_ [ bind(&datetime::set_minutes, &ret, _1) ] >> ':'
         >> u2_ [ bind(&datetime::set_seconds, &ret, _1) ];

    const char* first = s.data();
    const char* const end = first + s.size();
    const bool success = boost::spirit::qi::parse(first, end,
        (
            (date_parser >> 'T' >> time_parser)
            | date_parser
            | time_parser
        )
        >> timezone_parser
    );
```

```
        if (!success || first != end) {
            throw std::logic_error("Parsing of '" + s + "' failed");
        }
        return ret;
    } // end of parse_datetime() function
```

How it works…

A very interesting variable here is `boost::spirit::qi::rule<const char*, void()>`. It erases the exact type of a resulting parser and allows you to write parsers for recursive grammars. It also allows you to write parsers in source files and export them to headers without significantly affecting compilation times of your project. For example:

```
// Somewhere in header file
class example_1 {
    boost::spirit::qi::rule<const char*, void()> some_rule_;
public:
    example_1();
};

// In source file
example_1::example_1() {
    some_rule_ = /* ... a lot of parser code ... */;
}
```

Note that this class implies optimization barrier for compilers, so do not use it when it is not required.

Sometimes we used `>> ':'` instead of `>> char_(':')`. The first approach is more limited: you cannot bind actions to it and cannot do new rules by just combining chars (for example, you cannot write `char_('+')|char_('-')` without using `char_` at all). But for better performance use the first approach, because some compilers may optimize it slightly better.

There’s more…

We can make our example slightly faster by removing the `rule<>` objects that do type erasure. Just replace them with the C++11 `auto` keyword.

The `Boost.Spirit` library generates very fast parsers; there are some performance measures at the official site. Official documentation contains advanced recommendations for writing faster parsers.

The usage of `boost::phoenix::bind` is not mandatory, but without it the rule that parses specific zone offset in `timezone_parser` will be dealing with the `boost::fusion::vector<char, unsigned short, unsigned short>` type. Using `bind(&set_zone_offset, ref(ret), _1, _2, _3)` seems to be a more reader-friendly solution.

When parsing big files, consider reading the *Fastest way to read files* recipe in Chapter 11, *Working with the System*, because incorrect work with files may slow down your program much more than parsing.

Compiling the code that uses the `Boost.Spirit` (or `Boost.Fusion`) library may take a lot of time, because of the huge amount of template instantiations. When experimenting with the `Boost.Spirit` library use modern compilers, they provide better compilation times.

See also

The `Boost.Spirit` library is worth writing a separate book, it’s impossible to describe all of its features in a few recipes, so referring to the documentation will help you to get more information about it. It is available at http://boost.org/libs/spirit . There, you’ll find much more examples, ready parsers, and information on how to write lexers and generators directly in C++11 code using Boost.

Compile-Time Tricks

In this chapter, we will cover the following:

- Checking sizes at compile time
- Enabling function template usage for integral types
- Disabling function template usage for real types
- Creating a type from a number
- Implementing a type trait
- Selecting an optimal operator for a template parameter
- Getting a type of expression in C++03

Introduction

In this chapter, we'll see some basic examples on how the Boost libraries can be used in compile-time checking, for tuning algorithms, and in other metaprogramming tasks.

Some readers may ask, *"Why should we care about compile-time things?"* That's because the released version of the program is compiled once and runs multiple times. The more we do at compile time, the less work remains for runtime, resulting in much faster and reliable programs. Runtime checks are executed only if a part of the code with the check is executed. Compile-time checks will prevent your program from compiling, ideally with a meaningful compiler error message.

This chapter is possibly one of the most important. Understanding Boost sources and other Boost-like libraries is impossible without it.

Checking sizes at compile time

Let’s imagine that we are writing some serialization function that stores values in a buffer of a specified size:

```
#include <cstring>
#include <boost/array.hpp>

// C++17 has std::byte out of the box!
// Unfortunately this is as C++03 example.
typedef unsigned char byte_t;

template <class T, std::size_t BufSizeV>
void serialize_bad(const T& value, boost::array<byte_t, BufSizeV>& buffer) {
    // TODO: check buffer size.
    std::memcpy(&buffer[0], &value, sizeof(value));
}
```

This code has the following problems:

- The size of the buffer is not checked, so it may overflow
- This function can be used with **non-trivially copyable** types, which would lead to incorrect behavior

We may partially fix it by adding some asserts, for example:

```
template <class T, std::size_t BufSizeV>
void serialize_bad(const T& value, boost::array<byte_t, BufSizeV>& buffer) {
    // TODO: think of something better.
    assert(BufSizeV >= sizeof(value));
    std::memcpy(&buffer[0], &value, sizeof(value));
}
```

But, this is a bad solution. Runtime checks do not trigger the assert during testing in the debug mode if the function was not called. Runtime checks may even be optimized out in the release mode, so very bad things may happen.
`BufSizeV` and `sizeof(value)` values are known at compile time. It means that, instead of having a runtime assert, we can force this code to fail compilation if the buffer is too small.

Getting ready

This recipe requires some knowledge of C++ templates and the `Boost.Array` library.

How to do it…

Let’s use `Boost.StaticAssert` and `Boost.TypeTraits` libraries to correct the solution. Here’s how:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/has_trivial_copy.hpp>

template <class T, std::size_t BufSizeV>
void serialize(const T& value, boost::array<byte_t, BufSizeV>& buffer) {
    BOOST_STATIC_ASSERT(BufSizeV >= sizeof(value));
    BOOST_STATIC_ASSERT(boost::has_trivial_copy<T>::value);

    std::memcpy(&buffer[0], &value, sizeof(value));
}
```

How it works…

The `BOOST_STATIC_ASSERT` macro can be used only if an assert expression can be evaluated at compile time and is implicitly convertible to `bool`. It means that you may only use `sizeof()`, static constants, constexpr variables, constexpr functions with parameters known at compile time, and other constant expressions in `BOOST_STATIC_ASSERT`. If assert expression evaluates to `false`, `BOOST_STATIC_ASSERT` will stop compilation. In the case of a `serialize` function, if the first static assertion fails, it means that the user misused the `serialize` function and provided a very small buffer.

Here are some more examples:

```
BOOST_STATIC_ASSERT(3 >= 1);

struct some_struct { enum enum_t { value = 1}; };
BOOST_STATIC_ASSERT(some_struct::value);

template <class T1, class T2>
struct some_templated_struct
{
    enum enum_t { value = (sizeof(T1) == sizeof(T2))};
};
BOOST_STATIC_ASSERT((some_templated_struct<int, unsigned int>::value));

template <class T1, class T2>
struct some_template {
    BOOST_STATIC_ASSERT(sizeof(T1) == sizeof(T2));
};
```

If the `BOOST_STATIC_ASSERT` macro's assert expression has a comma sign in it, we must wrap the whole expression in additional brackets.

The last example is very close to what we can see on the second line of the `serialize()` function. So now, it is time to discover more about the `Boost.TypeTraits` library. This library provides a large number of compile-time metafunctions that allow us to get information about types and modify types. The metafunctions usages look like `boost::function_name<parameters>::value` or `boost::function_name<parameters>::type`. The metafunction `boost::has_trivial_copy<T>::value` returns `true` only if `T` is a simple copyable type.

Let's take a look at some more examples:

```
#include <iostream>
#include <boost/type_traits/is_unsigned.hpp>
#include <boost/type_traits/is_same.hpp>
#include <boost/type_traits/remove_const.hpp>

template <class T1, class T2>
void type_traits_examples(T1& /*v1*/, T2& /*v2*/)  {
    // Returns true if T1 is an unsigned number
    std::cout << boost::is_unsigned<T1>::value;

    // Returns true if T1 has exactly the same type, as T2
    std::cout << boost::is_same<T1, T2>::value;

    // This line removes const modifier from type of T1.
    // Here is what will happen with T1 type if T1 is:
    // const int => int
    // int => int
    // int const volatile => int volatile
    // const int& => const int&
    typedef typename boost::remove_const<T1>::type t1_nonconst_t;
}
```

Some compilers may compile this code even without the `typename` keyword, but such behavior violates the C++ standard, so it is highly recommended to write `typename`.

There’s more…

The `BOOST_STATIC_ASSSERT` macro has a more verbose variant called `BOOST_STATIC_ASSSERT_MSG` that tries hard to output an error message in the compiler log (or in the IDE window) if assertion fails. Take a look at the following code:

```
template <class T, std::size_t BufSizeV>
void serialize2(const T& value, boost::array<byte_t, BufSizeV>& buf) {
    BOOST_STATIC_ASSERT_MSG(boost::has_trivial_copy<T>::value,
        "This serialize2 function may be used only "
        "with trivially copyable types."
    );

    BOOST_STATIC_ASSERT_MSG(BufSizeV >= sizeof(value),
        "Can not fit value to buffer. "
        "Make the buffer bigger."
    );

    std::memcpy(&buf[0], &value, sizeof(value));
}

int main() {
    // Somewhere in code:
    boost::array<unsigned char, 1> buf;
    serialize2(std::string("Hello word"), buf);
}
```

The preceding code will give the following result during compilation on the g++ compiler in the C++11 mode:

```
boost/static_assert.hpp:31:45: error: static assertion failed: This serialize2 function may be
used only with trivially copyable types.
 #     define BOOST_STATIC_ASSERT_MSG( ... ) static_assert(__VA_ARGS__)
                                             ^
Chapter04/01_static_assert/main.cpp:76:5: note: in expansion of macro ‘BOOST_STATIC_ASSERT_MSG;
     BOOST_STATIC_ASSERT_MSG(boost::has_trivial_copy<T>::value,
     ^~~~~~~~~~~~~~~~~~~~~~


boost/static_assert.hpp:31:45: error: static assertion failed: Can not fit value to buffer. Make
the buffer bigger.
 #     define BOOST_STATIC_ASSERT_MSG( ... ) static_assert(__VA_ARGS__)
                                             ^
Chapter04/01_static_assert/main.cpp:81:5: note: in expansion of macro ‘BOOST_STATIC_ASSERT_MSG;
     BOOST_STATIC_ASSERT_MSG(BufSizeV >= sizeof(value),
     ^~~~~~~~~~~~~~~~~~~~~~
```

Neither `BOOST_STATIC_ASSSERT`, nor `BOOST_STATIC_ASSSERT_MSG`, nor any of the type traits entity implies a runtime penalty. All these functions are executed at compile time and do not add a single assembly instruction to the resulting binary file. The C++11 standard has `static_assert(condition, "message")` that is equivalent to Boost’s `BOOST_STATIC_ASSSERT_MSG`. The `BOOST_STATIC_ASSERT` functionality of asserting at compile time without a user provided message is available in C++17 as `static_assert(condition)`. You do not have to include header files to be able to use your compiler built in `static_assert`.

The `Boost.TypeTraits` library was partially accepted into the C++11 standard. You may thus find traits in the `<type_traits>` header in the `std::` namespace. C++11 `<type_traits>` has some functions that do not exist in `Boost.TypeTraits`, but some other metafunctions exist only in Boost. Metafunctions that have a name starting with `has_` are renamed in the standard library to metafunctions with names starting with `is_`. Thus, `has_trivial_copy` became `is_trivially_copyable` and so forth.

C++14 and Boost 1.65 have shortcuts for all the type traits that have a `::type` member. Those shortcuts allow you to write `remove_const_t<T1>` instead of `typename remove_const<T1>::type`. Note that, in the case of Boost 1.65, the shortcuts require a C++11 compatible compiler as they could be implemented only using **type aliases**:

```
template <class T>
using remove_const_t = typename remove_const<T>::type;
```

C++17 added `_v` shortcuts for type traits that have `::value`. Since C++17, you can just write `std::is_unsigned_v<T1>` instead of `std::is_unsigned<T1>::value`. This trick is usually implemented using `variable templates`:

```
template <class T>
inline constexpr bool is_unsigned_v = is_unsigned<T>::value;
```

When there is a similar trait in Boost and in the standard library, do opt for the Boost version if you are writing a project that must work on a pre-C++11 compilers. Otherwise, the standard library version may work slightly better, in rare cases.

See also

- The next recipes in this chapter will give you more examples and ideas of how static asserts and type traits may be used.
- Read the official documentation of `Boost.StaticAssert` for more examples at:

http://boost.org/libs/static_assert.

Enabling function template usage for integral types

It’s a common situation, when we have a class template that implements some functionality:

```
// Generic implementation.
template <class T>
class data_processor {
    double process(const T& v1, const T& v2, const T& v3);
};
```

Now, imagine that we have two additional versions of that class, one for integral, and another for real types:

```
// Integral types optimized version.
template <class T>
class data_processor_integral {
    typedef int fast_int_t;
    double process(fast_int_t v1, fast_int_t v2, fast_int_t v3);
};

// SSE optimized version for float types.
template <class T>
class data_processor_sse {
    double process(double v1, double v2, double v3);
};
```

Now the question: How to make the compiler to automatically choose the correct class for a specified type?

Getting ready

This recipe requires some knowledge of C++ templates.

How to do it…

We'll be using `Boost.Core` and `Boost.TypeTraits` to resolve the problem:

1. Let's start by including headers:

```
#include <boost/core/enable_if.hpp>
#include <boost/type_traits/is_integral.hpp>
#include <boost/type_traits/is_float.hpp>
```

2. Let's add an additional template parameter with a default value to our generic implementation:

```
// Generic implementation.
template <class T, class Enable = void>
class data_processor {
    // ...
};
```

3. Modify optimized versions in the following way, so that now they will now be treated by the compiler as template partial specializations:

```
// Integral types optimized version.
template <class T>
class data_processor<
    T,
    typename boost::enable_if_c<boost::is_integral<T>::value >::type
>
{
    // ...
};

// SSE optimized version for float types.
template <class T>
class data_processor<
    T,
    typename boost::enable_if_c<boost::is_float<T>::value >::type
>
{
    // ...
};
```

4. And, that's it! Now, the compiler will automatically choose the correct class:

```
template <class T>
double example_func(T v1, T v2, T v3) {
    data_processor<T> proc;
    return proc.process(v1, v2, v3);
}

int main () {
    // Integral types optimized version
    // will be called.
    example_func(1, 2, 3);
    short s = 0;
    example_func(s, s, s);

    // Real types version will be called.
    example_func(1.0, 2.0, 3.0);
    example_func(1.0f, 2.0f, 3.0f);

    // Generic version will be called.
    example_func("Hello", "word", "processing");
}
```

How it works…

The `boost::enable_if_c` template is a tricky one. It makes use of the **Substitution Failure Is Not An Error** (**SFINAE**) principle, which is used during **template instantiation**. This is how the principle works; if an invalid argument or return type is formed during the instantiation of a function or class template, the instantiation is removed from the overload resolution set and does not cause a compilation error. Now the tricky part, `boost::enable_if_c<true>` has a member type accessible via `::type`, but `boost::enable_if_c<false>` has no `::type`. Let's get back to our solution and see how the SFINAE works with different types passed to the `data_processor` class as the `T` parameter.

If we pass an `int` as the `T` type, first the compiler will try to instantiate template partial specializations from *step 3*, before using our nonspecialized generic version. When it tries to instantiate a `float` version, the `boost::is_float<T>::value` metafunction returns `false`. The `boost::enable_if_c<false>::type` metafunction cannot be correctly instantiated because `boost::enable_if_c<false>` has no `::type`, and that is the place where SFINAE acts. Because the class template cannot be instantiated, this must be interpreted as not an error, compiler skips this template specialization. The next partial specialization is the one that is optimized for integral types. The `boost::is_integral<T>::value` metafunction returns `true`, and `boost::enable_if_c<true>::type` can be instantiated, which makes it possible to instantiate the whole `data_processor` specialization. The compiler found a matching partial specialization, so it does not need to try to instantiate the nonspecialized method.

Now, let's try to pass some nonarithmetic type (for example, `const char *`), and let's see what the compiler will do. First, the compiler tries to instantiate template partial specializations. The specializations with `is_float<T>::value` and `is_integral<T>::value` fail to instantiate, so the compiler tries to instantiate our generic version and succeeds.

Without `boost::enable_if_c<>`, all the partially specialized versions may be instantiated at the same time for any type, which leads to ambiguity and failed compilation.

If you are using templates and compiler reports that cannot choose between two template classes of methods, you probably need `boost::enable_if_c<>`.

There’s more…

Another version of this method is called `boost::enable_if` without `_c` at the end. The difference between them is that `enable_if_c` accepts constant as a template parameter; the short version accepts an object that has a `value` static member. For example, `boost::enable_if_c<boost::is_integral<T>::value >::type` is equal to `boost::enable_if<boost::is_integral<T> >::type`.

Before Boost 1.56 the `boost::enable_if` metafunctions were defined in the header `<boost/utility/enable_if.hpp>` instead of `<boost/core/enable_if.hpp>`.

C++11 has `std::enable_if` defined in the `<type_traits>` header, which behaves exactly like `boost::enable_if_c`. No difference between them exists, except that Boost’s version works on non C++11 compilers too, providing better portability.

C++14 has a shortcut `std::enable_if_t` that must be used without `typename` and `::type`:

```
template <class T>
class data_processor<
    T, std::enable_if_t<boost::is_float<T>::value >
>;
```

All the enabling functions are executed only at compile time and do not add a performance overhead at runtime. However, adding an additional template parameter may produce a bigger classname in `typeid(your_class).name()`, and add an extremely tiny performance overhead while comparing two `typeid()` results on some platforms.

See also

- The next recipes will give you more examples of the `enable_if` usage.
- You may also consult the official documentation of `Boost.Core`. It contains many examples and a lot of useful classes (which are used widely in this book). Follow the link http://boost.org/libs/core to read about it.
- You may also read some articles about template partial specializations at http://msdn.microsoft.com/en-us/library/3967w96f%28v=vs.110%29.aspx.

Disabling function template usage for real types

We continue working with Boost metaprogramming libraries. In the previous recipe, we saw how to use `enable_if_c` with classes; now it is time to take a look at its usage in template functions.

Imagine that, in your project, you have a template function that works with all the available types:

```
template <class T>
T process_data(const T& v1, const T& v2, const T& v3);
```

That function exist for a long time. You have written a lot of code that uses it. Suddenly, you came up with an optimized version of the `process_data` function but only for types that do have an `T::operator+=(const T&)`:

```
template <class T>
T process_data_plus_assign(const T& v1, const T& v2, const T& v3);
```

You’ve got a huge code base and it may take months to manually change `process_data` to the `process_data_plus_assign` for types that have the right operators. So, you do not want to change the already written code. Instead, you want to force the compiler to automatically use an optimized function in place of the default one if that’s possible.

Getting ready

Read the previous recipe to get an idea of what `boost::enable_if_c` does and for an understanding of the concept of SFINAE. A basic knowledge of templates is still required.

How to do it…

Template magic can be done using the Boost libraries. Let's see how to do it:

1. We will need the `boost::has_plus_assign<T>` metafunction and the `<boost/enable_if.hpp>` header:

```
#include <boost/core/enable_if.hpp>
#include <boost/type_traits/has_plus_assign.hpp>
```

2. Now, we disable default implementation for types with the `plus assign` operator:

```
// Modified generic version of process_data
template <class T>
typename boost::disable_if_c<boost::has_plus_assign<T>::value,T>::type
    process_data(const T& v1, const T& v2, const T& v3);
```

3. Enable the optimized version for types with the `plus assign` operator:

```
// This process_data will call a process_data_plus_assign.
template <class T>
typename boost::enable_if_c<boost::has_plus_assign<T>::value, T>::type
    process_data(const T& v1, const T& v2, const T& v3)
{
    return process_data_plus_assign(v1, v2, v3);
}
```

4. Now, the optimized version is used wherever possible:

```
int main() {
    int i = 1;
    // Optimized version.
    process_data(i, i, i);

    // Default version.
    // Explicitly specifing template parameter.
    process_data<const char*>("Testing", "example", "function");
}
```

How it works…

The `boost::disable_if_c<bool_value>::type` metafunction disables the method, if `bool_value` equals `true`. It works just like `boost::enable_if_c<!bool_value>::type`.

A class passed as the second parameter for `boost::enable_if_c` or `boost::disable_if_c` is returned via `::type` in the case of successful substitution. In other words, `boost::enable_if_c<true, T>::type` is the same as `T`.

Let's go through the `process_data(i, i, i)` case, step by step. We pass an `int` as `T` type and the compiler searches for function `process_data(int, int, int)`. Because there is no such function, the next step is to instantiate a template version of `process_data`. However, there are two template `process_data` functions. For example, the compiler starts instantiating the templates from our second (optimized) version; in that case, it successfully evaluates the `typename boost::enable_if_c<boost::has_plus_assign<T>::value, T>::type` expression, and gets the `T` return type. But, the compiler does not stop; it continues instantiation attempts and tries to instantiate our first version of the function. During substitution of `typename boost::disable_if_c<boost::has_plus_assign<T>::value` a failure happens, which is not treated as an error due to the SFINAE rule. There are no more template `process_data` functions, so the compiler stops instantiating. As you can see, without `enable_if_c` and `disable_if_c`, the compiler would be able to instantiate both templates and there will be an ambiguity.

There’s more…

As in the case of `enable_if_c` and `enable_if`, there is a `disable_if` version of the disabling function:

```
// First version
template <class T>
typename boost::disable_if<boost::has_plus_assign<T>, T>::type
    process_data2(const T& v1, const T& v2, const T& v3);

// process_data_plus_assign
template <class T>
typename boost::enable_if<boost::has_plus_assign<T>, T>::type
    process_data2(const T& v1, const T& v2, const T& v3);
```

C++11 has neither `disable_if_c` nor `disable_if`, but you are free to use `std::enable_if<!bool_value>::type` instead.

Before Boost 1.56 the `boost::disable_if` metafunctions were defined in the header `<boost/utility/enable_if.hpp>` instead of `<boost/core/enable_if.hpp>`.

As it was mentioned in the previous recipe, all the enabling and disabling functions are executed only at compile time and do not add performance overhead at runtime.

See also

- Read this chapter from the beginning to get more examples of compile-time tricks.
- Consider reading the `Boost.TypeTraits` official documentation for more examples and a full list of metafunctions at http://boost.org/libs/type_traits.
- The `Boost.Core` library may provide you with more examples of `boost::enable_if` usage; read about it at http://boost.org/libs/core.

Creating a type from a number

We have now seen examples of how we can choose between functions using `boost::enable_if_c`. Let’s forget about that technique for this chapter and use a different approach. Consider the following example, where we have a generic method for processing POD datatypes:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_pod.hpp>

// Generic implementation.
template <class T>
T process(const T& val) {
    BOOST_STATIC_ASSERT((boost::is_pod<T>::value));
    // ...
}
```

We also have some processing functions optimized for sizes 1, 4, and 8 bytes. How do we rewrite the `process` function so that it can dispatch calls to optimized processing functions?

Getting ready

Reading at least the first recipe from this chapter is highly recommended, so that you will not be confused by all the things that are happening here. Templates and metaprogramming will not scare you (or just get ready to see a lot of them).

How to do it…

We are going to see how the size of a template type can be converted to a variable of some type, and how that variable can be used for deducing the right function overload.

1. Let’s define our generic and optimized versions of the `process_impl` function:

```
#include <boost/mpl/int.hpp>

namespace detail {
    // Generic implementation.
    template <class T, class Tag>
    T process_impl(const T& val, Tag /*ignore*/) {
        // ...
    }

    // 1 byte optimized implementation.
    template <class T>
    T process_impl(const T& val, boost::mpl::int_<1> /*ignore*/) {
        // ...
    }

    // 4 bytes optimized implementation.
    template <class T>
    T process_impl(const T& val, boost::mpl::int_<4> /*ignore*/) {
        // ...
    }

    // 8 bytes optimized implementation.
    template <class T>
    T process_impl(const T& val, boost::mpl::int_<8> /*ignore*/) {
        // ...
    }
} // namespace detail
```

2. Now, we are ready to write a process function:

```
// Dispatching calls:
template <class T>
T process(const T& val) {
    BOOST_STATIC_ASSERT((boost::is_pod<T>::value));
    return detail::process_impl(val, boost::mpl::int_<sizeof(T)>());
}
```

How it works…

The most interesting part here is that boost::mpl::int_<sizeof(T)>(). sizeof(T) executes at compile time, so its output can be used as a template parameter. The class boost::mpl::int_<> is just an empty class that holds a compile-time value of integral type. In the Boost.MPL library, such classes are called **integral constants**. It can be implemented as shown in the following code:

```
template <int Value>
struct int_ {
    static const int value = Value;
    typedef int_<Value> type;
    typedef int value_type;
};
```

We need an instance of this class, which is why we have a round parentheses at the end of boost::mpl::int_<sizeof(T)>().

Now, let’s take a closer look at how the compiler will decide which process_impl function to use. First of all, the compiler tries to match functions that have a non-template second parameter . If sizeof(T) is 4, the compiler tries to search the function with signatures like process_impl(T, boost::mpl::int_<4>) and finds our 4 bytes optimized version from the detail namespace. If sizeof(T) is 34, the compiler can not find the function with a signature like process_impl(T, boost::mpl::int_<34>), and uses a template function process_impl(const T& val, Tag /*ignore*/).

There’s more…

The `Boost.MPL` library has several data structures for metaprogramming. In this recipe, we only scratched the tip of the iceberg. You may find the following integral constant classes from MPL useful:

- `bool_`
- `int_`
- `long_`
- `size_t`
- `char_`

All the `Boost.MPL` functions (except the `for_each` runtime function) are executed at compile time and won’t add runtime overhead.
The `Boost.MPL` library is not a part of C++. However, C++ reuses many tricks from that library. C++11 in the header file `type_traits` has an `std::integral_constant<type, value>` class that could be used in the same way as in the preceding example. You could even define your own **type aliases** using it:

```
template <int Value>
using int_ = std::integral_constant<int, Value>;
```

See also

- The recipes from Chapter 8, *Metaprogramming*, will give you more examples of the `Boost.MPL` library usage. If you feel confident, you may also try to read the library documentation at http://boost.org/libs/mpl link.
- Read more examples of tags usage at http://boost.org/libs/type_traits/doc/html/boost_typetraits/examples/fill.html and http://boost.org/libs/type_traits/doc/html/boost_typetraits/examples/copy.html.

Implementing a type trait

We need to implement a type trait that returns `true` if the `std::vector` type is passed to it as a template parameter and `false` otherwise.

Getting ready

Some basic knowledge of the `Boost.TypeTrait` or standard library type traits is required.

How to do it…

Let’s see how to implement a type trait:

```
#include <vector>
#include <boost/type_traits/integral_constant.hpp>

template <class T>
struct is_stdvector: boost::false_type {};

template <class T, class Allocator>
struct is_stdvector<std::vector<T, Allocator> >: boost::true_type  {};
```

How it works…

Almost all the work is done by the `boost::true_type` and `boost::false_type` classes. The `boost::true_type` class has a boolean `::value` static constant in it that equals `true`. The `boost::false_type` class has a boolean `::value` static constant in it that equals `false`. These two classes also have some `typedef`s to cooperate well with the `Boost.MPL` library.

Our first `is_stdvector` structure is a generic structure that will be used always when a template specialized version of such structure is not found. Our second `is_stdvector` structure is a template specialization for the `std::vector` types (note that it is derived from `true_type`). So, when we pass `std::vector` type to the `is_stdvector` structure, a template specialized version is chosen by the compiler. If we pass a data type other than `std::vector`, then the generic version is used which is derived from `false_type`.

There is no public keyword before `boost::false_type` and, `boost::true_type` in our trait, because we use `struct` keyword, and by default, it uses public inheritance.

There's more…

Those readers who use the C++11 compatible compilers may use the `true_type` and `false_type` types declared in the `<type_traits>` header for creating their own type traits. Since C++17, the standard library has a `bool_constant<true_or_false>` type alias that you may use for convenience.

As usual, the Boost versions of the classes and functions are more portable because they can be used on pre-C++11 compilers.

See also

- Almost all the recipes from this chapter use type traits. Refer to the `Boost.TypeTraits` documentation for more examples and information at http://boost.org/libs/type_traits
- See the previous recipe to get more information on integral constants and how the `true_type` and `false_type` may be implemented from scratch.

Selecting an optimal operator for a template parameter

Imagine that we are working with classes from different vendors that implement different numbers of arithmetic operations and have constructors from integers. We do want to make a function that increments by any one class that is passed to it. Also, we want this function to be effective! Take a look at the following code:

```
template <class T>
void inc(T& value) {
    // TODO:
    // call ++value
    // or call value ++
    // or value += T(1);
    // or value = value + T(1);
}
```

Getting ready

Some basic knowledge of the C++ templates, and the `Boost.TypeTrait` or standard library type traits is required.

How to do it…

All the selecting can be done at compile time. This can be achieved using the `Boost.TypeTraits` library, as shown in the following steps:

1. Let's start by making correct functional objects:

```
namespace detail {
    struct pre_inc_functor {
    template <class T>
        void operator()(T& value) const {
           ++ value;
        }
    };

    struct post_inc_functor {
    template <class T>
        void operator()(T& value) const {
            value++;
        }
    };

    struct plus_assignable_functor {
    template <class T>
        void operator()(T& value) const {
            value += T(1);
        }
    };

    struct plus_functor {
    template <class T>
        void operator()(T& value) const {
            value = value + T(1);
        }
    };
}
```

2. After that, we will need a bunch of type traits:

```
#include <boost/type_traits/conditional.hpp>
#include <boost/type_traits/has_plus_assign.hpp>
#include <boost/type_traits/has_plus.hpp>
#include <boost/type_traits/has_post_increment.hpp>
#include <boost/type_traits/has_pre_increment.hpp>
```

3. We are ready to deduce the correct functor and use it:

```
template <class T>
void inc(T& value) {
    // call ++value
    // or call value ++
    // or value += T(1);
    // or value = value + T(1);

    typedef detail::plus_functor step_0_t;

    typedef typename boost::conditional<
      boost::has_plus_assign<T>::value,
      detail::plus_assignable_functor,
      step_0_t
    >::type step_1_t;

    typedef typename boost::conditional<
      boost::has_post_increment<T>::value,
      detail::post_inc_functor,
      step_1_t
    >::type step_2_t;

    typedef typename boost::conditional<
      boost::has_pre_increment<T>::value,
      detail::pre_inc_functor,
      step_2_t
    >::type step_3_t;

    step_3_t() // Default construction of the functor.
        (value); // Calling operator() of the functor.
}
```

How it works…

All the magic is done via the `conditional<bool Condition, class T1, class T2>` metafunction. When `true` is passed into the metafunction as a first parameter, it returns `T1` via the `::type typedef`. When `false` is passed into the metafunction as a first parameter, it returns `T2` via the `::type typedef`. It acts like some kind of compile-time `if` statement.

So, `step0_t` holds a `detail::plus_functor` metafunction and `step1_t` holds `step0_t` or `detail::plus_assignable_functor`. The `step2_t` type holds `step1_t` or `detail::post_inc_functor`. The `step3_t` type holds `step2_t` or `detail::pre_inc_functor`. What each `step*_t typedef` holds is deduced using type trait.

There's more…

There is a C++11 version of this function, which can be found in the `<type_traits>` header in the `std::` namespace. Boost has multiple versions of this function in different libraries; for example, `Boost.MPL` has function `boost::mpl::if_c`, which acts exactly like `boost::conditional`. It also has a version `boost::mpl::if_` (without `c` at the end), which calls `::type` for its first template argument; and if it is derived from `boost::true_type`, it returns its second argument during the `::type` call. Otherwise, it returns the last template parameter. We can rewrite our `inc()` function to use `Boost.MPL`, as shown in the following code:

```
#include <boost/mpl/if.hpp>

template <class T>
void inc_mpl(T& value) {
    typedef detail::plus_functor step_0_t;

    typedef typename boost::mpl::if_<
      boost::has_plus_assign<T>,
      detail::plus_assignable_functor,
      step_0_t
    >::type step_1_t;

    typedef typename boost::mpl::if_<
      boost::has_post_increment<T>,
      detail::post_inc_functor,
      step_1_t
    >::type step_2_t;

    typedef typename boost::mpl::if_<
      boost::has_pre_increment<T>,
      detail::pre_inc_functor,
      step_2_t
    >::type step_3_t;

    step_3_t()   // Default construction of the functor.
        (value); // Calling operator() of the functor.
}
```

C++17 has an `if constexpr` construction that makes the preceding example much simpler:

```
template <class T>
void inc_cpp17(T& value) {
    if constexpr (boost::has_pre_increment<T>()) {
        ++value;
    } else if constexpr (boost::has_post_increment<T>()) {
        value++;
    } else if constexpr(boost::has_plus_assign<T>()) {
        value += T(1);
    } else {
        value = value + T(1);
    }
}
```

Integral constants in the standard library, `Boost.MPL` and `Boost.TypeTraits` have a constexpr conversion operator. For example, it means that an instance of `std::true_type` can be converted to `true` value. In the preceding example, `boost::has_pre_increment<T>` denotes a type, appending `()`, or C++11 curly brackets `{}` make an instance of that type, that is convertible to `true` or `false` values.

See also

- The recipe *Enabling template functions usage for integral types.*
- The recipe *Disabling template functions usage for real types.*
- The `Boost.TypeTraits` documentation has a full list of available metafunctions. Follow the link http://boost.org/libs/type_traits to read about it.
- The recipes from Chapter 8, *Metaprogramming*, will give you more examples of the `Boost.MPL` library usage. If you feel confident, you may also try to read its documentation at http://boost.org/libs/mpl link.

Getting a type of expression in C++03

In the previous recipes, we saw some examples of `boost::bind` usage. It may be a useful tool in pre-C++11 word, but it is hard to store `boost::bind` result as a variable in C++03.

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

const ??? var = boost::bind(std::plus<int>(), _1, _1);
```

In C++11, we can use `auto` keyword instead of `???` and that will work. Is there a way to do it in C++03?

Getting ready

A knowledge of the C++11 `auto` and `decltype` keywords may help you to understand this recipe.

How to do it…

We will need a `Boost.Typeof` library for getting a return type of expression:

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

BOOST_AUTO(var, boost::bind(std::plus<int>(), _1, _1));
```

How it works…

It just creates a variable with the name `var`, and the value of the expression is passed as a second argument. The type of `var` is detected from the type of expression.

There's more…

An experienced C++ reader will note that there are more keywords in the C++11 for detecting the types of expression. Maybe `Boost.Typeof` has a macro for them too. Let's take a look at the following C++11 code:

```
typedef decltype(0.5 + 0.5f) type;
```

Using `Boost.Typeof`, the preceding code can be written in the following way:

```
typedef BOOST_TYPEOF(0.5 + 0.5f) type;
```

C++11 version's `decltype(expr)` deduces and returns the type of `expr`.

```
template<class T1, class T2>
auto add(const T1& t1, const T2& t2) ->decltype(t1 + t2) {
    return t1 + t2;
};
```

Using `Boost.Typeof`, the preceding code can be written like this:

```
// Long and portable way:
template<class T1, class T2>
struct result_of {
    typedef BOOST_TYPEOF_TPL(T1() + T2()) type;
};

template<class T1, class T2>
typename result_of<T1, T2>::type add(const T1& t1, const T2& t2) {
    return t1 + t2;
};

// ... or ...

// Shorter version that may crush some compilers.
template<class T1, class T2>
BOOST_TYPEOF_TPL(T1() + T2()) add(const T1& t1, const T2& t2) {
    return t1 + t2;
};
```

C++11 has a special syntax for specifying return type at the end of the function declaration. Unfortunately, this cannot be emulated in C++03, so we cannot use `t1` and `t2` variables in a macro.

You can freely use the results of the `BOOST_TYPEOF()` functions in templates and in any other compile-time expressions:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_same.hpp>
BOOST_STATIC_ASSERT((
    boost::is_same<BOOST_TYPEOF(add(1, 1)), int>::value
));
```

Unfortunately, however this magic does not always work without help. For example, user-defined classes are not always detected, so the following code may fail on some compilers:

```
namespace readers_project {
    template <class T1, class T2, class T3>
    struct readers_template_class{};
}

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

```
typedef
    readers_project::readers_template_class<int, int, float>
readers_template_class_1;

typedef BOOST_TYPEOF(boost::get<0>(
    boost::make_tuple(readers_template_class_1(), 1)
)) readers_template_class_deduced;

BOOST_STATIC_ASSERT((
    boost::is_same<
        readers_template_class_1,
        readers_template_class_deduced
    >::value
));
```

In such situations, you may give `Boost.Typeof` a helping hand and register a template:

```
BOOST_TYPEOF_REGISTER_TEMPLATE(
        readers_project::readers_template_class /*class name*/,
        3 /*number of template classes*/
)
```

However, the three most popular compilers correctly detected the type even without `BOOST_TYPEOF_REGISTER_TEMPLATE` and without C++11.

See also

The official documentation of `Boost.Typeof` has more examples. Follow the link http://boost.org/libs/typeof to read about it.

Multithreading

In this chapter, we will cover:

- Creating a thread of execution
- Syncing access to a common resource
- Fast access to a common resource using atomics
- Creating a work_queue class
- Multiple-readers-single-writer lock
- Creating variables that are unique per thread
- Interrupting a thread
- Manipulating a group of threads
- Initializing a shared variable safely
- Locking multiple mutexes

Introduction

In this chapter, we'll take care of threads and all the stuff connected with them. Basic knowledge of multithreading is encouraged.

Multithreading means multiple threads of execution exist within a single process. Threads may share process resources and have their own resources. Those threads of execution may run independently on different CPUs, leading to faster and more responsible programs. The `Boost.Thread` library provides unification across operating system interfaces to work with threads. It is not a header-only library, so all the examples from this chapter need to link against the `libboost_thread` and `libboost_system` libraries.

Creating a thread of execution

On modern multi-core compilers, to achieve maximal performance (or just to provide a good user experience), programs usually use multiple threads of execution. Here is a motivating example in which we need to create and fill a big file in a thread that draws the user interface:

```
#include <cstddef> // for std::size_t

bool is_first_run();

// Function that executes for a long time.
void fill_file(char fill_char, std::size_t size, const char* filename);

// Called in thread that draws a user interface:
void example_without_threads() {
    if (is_first_run()) {
        // This will be executing for a long time during which
        // users interface freezes...
        fill_file(0, 8 * 1024 * 1024, "save_file.txt");
    }
}
```

Getting ready

This recipe requires knowledge of `boost::bind` or `std::bind`.

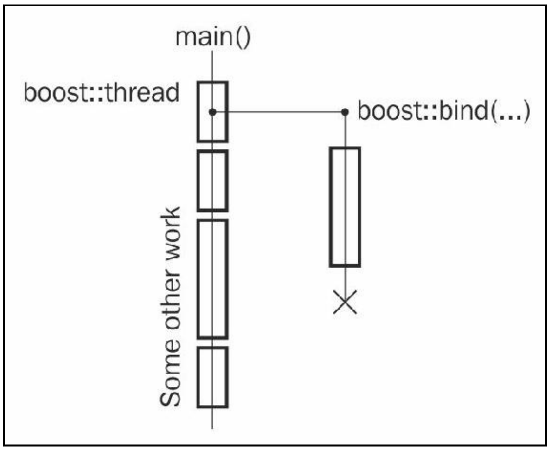
How to do it…

Starting a thread of execution was never so easy:

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

// Called in thread that draws a user interface:
void example_with_threads() {
    if (is_first_run()) {
        boost::thread(boost::bind(
            &fill_file,
            0,
            8 * 1024 * 1024,
            "save_file.txt"
        )).detach();
    }
}
```

How it works…

The `boost::thread` variable accepts a functional object that can be called without parameters (we provided one using `boost::bind`) and creates a separate thread of execution. That functional object is copied into a constructed thread of execution and run there. The return value of the functional object is ignored.



We are using version 4 of the `Boost.Thread` in all recipes (defined `BOOST_THREAD_VERSION` to `4`). Important differences between `Boost.Thread` versions are highlighted.

After that, we call the `detach()` function, which does the following:

- The thread of execution is detached from the `boost::thread` variable but continues its execution
- The `boost::thread` variable starts to hold a `Not-A-Thread` state

Without a call to `detach()`, the destructor of `boost::thread` will notice that it still holds an OS thread and will call `std::terminate`. It terminates our program without calling destructors, freeing up resources and without other cleanup.

Default constructed threads also have a `Not-A-Thread` state, and they do not create a separate thread of execution.

There’s more…

What if we want to make sure that a file was created and written before doing some other job? In that case, we need to join thread in the following way:

```
void example_with_joining_threads() {
    if (is_first_run()) {
        boost::thread t(boost::bind(
            &fill_file,
            0,
            8 * 1024 * 1024,
            "save_file.txt"
        ));

        // Do some work.
        // ...

        // Waiting for thread to finish.
        t.join();
    }
}
```

After the thread is joined, the `boost::thread` variable holds a `Not-A-Thread` state and its destructor does not call `std::terminate`.

Remember that the thread must be joined or detached before its destructor is called. Otherwise, your program will terminate!

With `BOOST_THREAD_VERSION=2` defined, the destructor of `boost::thread` calls `detach()`, which does not lead to `std::terminate`. But doing so breaks compatibility with `std::thread` and, some day, when your project is moving to the C++ standard library threads, or when `BOOST_THREAD_VERSION=2` isn’t supported, this will give you a lot of surprises. Version 4 of `Boost.Thread` is more explicit and strong, which is usually preferable in C++ language.

Beware that `std::terminate()` is called when any exception that is not of type `boost::thread_interrupted` leaves a boundary of the functional object that was passed to the `boost::thread` constructor.

There is a very helpful wrapper that works as a RAII wrapper around the thread and allows you to emulate the `BOOST_THREAD_VERSION=2` behavior; it is called `boost::scoped_thread<T>`, where `T` can be one of the following classes:

- `boost::interrupt_and_join_if_joinable`: To interrupt and join a thread at destruction
- `boost::join_if_joinable`: To join a thread at destruction
- `boost::detach`: To detach a thread at destruction

Here is a short example:

```
#include <boost/thread/scoped_thread.hpp>

void some_func();

void example_with_raii() {
    boost::scoped_thread<boost::join_if_joinable> t(
        boost::thread(&some_func)
    );
    // 't' will be joined at scope exit.
}
```

The `boost::thread` class was accepted as a part of the C++11 standard and you can find it in the `<thread>` header in the `std::` namespace. There is no big difference between the Boost's Version 4 and C++11 standard library versions of the `thread` class. However, `boost::thread` is available on the C++03 compilers, so its usage is more versatile.

There is a very good reason for calling `std::terminate` instead of joining by default! C and C++ languages are often used in life critical software. Such software is controlled by other software, called **watchdog***. Those watchdogs can easily detect that an application has terminated but can not always detect deadlocks or detect them with longer delays. For example, for defibrillator software, it's safer to terminate, than hang on `join()` for a few seconds waiting for a watchdog reaction. Keep that in mind when designing such applications.*

See also

- All the recipes in this chapter use `Boost.Thread`. You may continue reading to get more information about the library.
- The official documentation has a full list of the `boost::thread` methods and remarks about their availability in the C++11 standard library. Follow the link http://boost.org/libs/thread for its official documentation.
- The *Interrupting a thread* recipe will give you an idea of what the `boost::interrupt_and_join_if_joinable` class does.

Syncing access to a common resource

Now that we know how to start threads of execution, we want to have access to some common resources from different threads:

```
#include <cassert>
#include <cstddef>
#include <iostream>

// In previous recipe we included
// <boost/thread.hpp>, which includes all
// the classes of Boost.Thread.
// Following header includes only boost::thread.
#include <boost/thread/thread.hpp>

int shared_i = 0;

void do_inc() {
    for (std::size_t i = 0; i < 30000; ++i) {
        const int i_snapshot = ++shared_i;
        // Do some work with i_snapshot.
        // ...
    }
}

void do_dec() {
    for (std::size_t i = 0; i < 30000; ++i) {
        const int i_snapshot = --shared_i;
        // Do some work with i_snapshot.
        // ...
    }
}

void run() {
    boost::thread t1(&do_inc);
    boost::thread t2(&do_dec);

    t1.join();
    t2.join();

    assert(global_i == 0); // Oops!
    std::cout << "shared_i == " << shared_i;
}
```

This `Oops!` is not written there accidentally. For some people, it may be a surprise, but there is a big chance that `shared_i` won't be equal to `0`:

```
    shared_i == 19567
```

Modern compilers and processors have a huge number of different tricky optimizations that can break the preceding code. We won't discuss them here, but there is a useful link in the See also section of the document that briefly describes them.

Things get even worse in cases when a common resource is a non-trivial class; segmentation faults and memory leaks may (and will) occur.

We need to change the code so that only one thread modifies the `shared_i` variable at a single moment of time and so that all the processor and compiler optimizations that inflict multithreaded code are bypassed.

Getting ready

Basic knowledge of threads is recommended for this recipe.

How to do it…

Let’s see how we can fix the previous example and make `shared_i` equal at the end of the run:

1. First of all, we’ll need to create a **mutex**:

```
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>

int shared_i = 0;
boost::mutex i_mutex;
```

2. Put all the operations that modify or get data from the `shared_i` variable between the following:

```
      {   // Critical section begin
          boost::lock_guard<boost::mutex> lock(i_mutex);
```

And the following:

```
      }   // Critical section end
```

Here’s how it should look:

```
void do_inc() {
    for (std::size_t i = 0; i < 30000; ++i) {
        int i_snapshot;
        { // Critical section begin.
            boost::lock_guard<boost::mutex> lock(i_mutex);
            i_snapshot = ++shared_i;
        } // Critical section end.

        // Do some work with i_snapshot.
        // ...
    }
}

void do_dec() {
    for (std::size_t i = 0; i < 30000; ++i) {
        int i_snapshot;
        { // Critical section begin.
            boost::lock_guard<boost::mutex> lock(i_mutex);
            i_snapshot = -- shared_i;
        } // Critical section end.

        // Do some work with i_snapshot.
        // ...
    }
}
```

How it works…

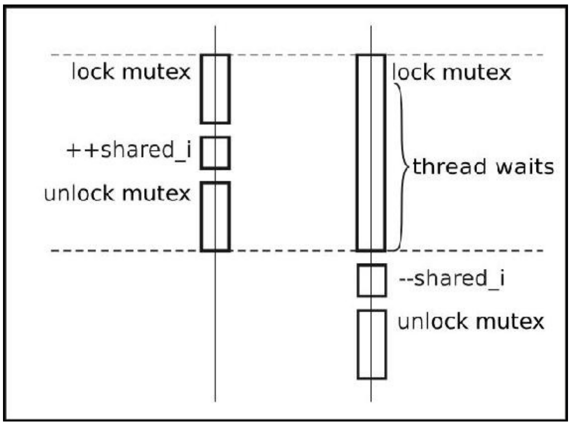
The `boost::mutex` class takes care of all the synchronization stuff. When a thread tries to lock it via the `boost::lock_guard<boost::mutex>` variable and there is no other thread holding a lock, it successfully acquires unique access to the section of code until the lock is unlocked or destroyed. If some other thread already holds a lock, the thread that tried to acquire the lock waits until another thread unlocks the lock. All the locking/unlocking operations imply specific instructions so that the changes made in a critical section are visible to all threads. Also, you no longer need to:

- Make sure that modified values of resources are visible to all cores
- Make sure that values are not just modified in the processor's register
- Force the processor to not reorder the instructions
- Force the compiler to not reorder the instructions
- Force the compiler to not remove writes to storage that is not read
- A bunch of other nasty compiler/architecture specific stuff

If you have a variable that is used from different threads and at least one thread modifies that variable, usually, all the code that uses it must be treated as a critical section and secured by a mutex.

The `boost::lock_guard` class is a very simple RAII class that stores a reference to the mutex, locks it in the single-parameter constructor, and unlocks it in the destructor.

In the curly bracket usage in the preceding example the `lock` variable is constructed inside them so that, on reaching the `// Critical section end.` closing bracket, the destructor for the `lock` variable is called and the mutex is unlocked. Even if some exception occurs in the critical section, the mutex is correctly unlocked.



If you initialize a common variable and then construct threads that only read it, then no mutex or other synchronization is required.

There’s more…

Locking a mutex is potentially a very slow operation, which may stop your code for a long time, until some other thread releases a lock. Try to make critical sections as small as possible; try to have less of them in your code.

Let’s take a look at how some operating systems handle locking on a multicore CPU. When `thread #1`, running on CPU 1, tries to lock a mutex that is already locked by another thread, `thread #1` is stopped by the OS till the lock is released. The stopped thread does not eat processor resources, so the OS executes other threads on CPU 1. Now, we have some threads running on CPU 1; some other thread releases the lock, and now the OS has to resume execution of a `thread #1`. So, it resumes its execution on a currently free CPU, for example, CPU2.

This results in CPU cache misses, so the code runs slightly slower after mutex is released. Usually, things are not so bad because a good OS tries hard to resume the thread on the same CPU that it was using before. Unfortunately, such OS-specific optimizations are not always possible. Reduce the count of critical sections and their sizes to reduce the chance of thread suspending and cache misses.

Do not attempt to lock a `boost::mutex` variable twice in the same thread; it will lead to a **deadlock**. If locking a mutex multiple times from a single thread is required, use `boost::recursive_mutex` instead from the `<boost/thread/recursive_mutex.hpp>` header. Locking it multiple times does not lead to a deadlock. The `boost::recursive_mutex` releases the lock only after `unlock()` is called once for each `lock()` call. Avoid using `boost::recursive_mutex` when it is not required, because it is slower than `boost::mutex` and usually indicates bad code flow design.

The `boost::mutex`, `boost::recursive_mutex`, and `boost::lock_guard` classes were accepted to the C++11 standard library, and you may find them in the `<mutex>` header in the `std::` namespace. No big difference between Boost and standard library versions exists. The Boost version may have some extensions (which are marked in the official documentation as *EXTENSION*) and provide better portability because they can be used even on pre-C++11 compilers.

See also

- The next recipe will give you ideas on how to make this example much faster (and shorter).
- Read the first recipe from this chapter to get more information about the `boost::thread` class. The official documentation at http://boost.org/libs/thread may help you too.
- To get more information about why the first example fails and how multiprocessors work with common resources, see *Memory Barriers: a Hardware View for Software Hackers* at http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2010.07.23a.pdf. Note that this is a hard topic.

Fast access to common resource using atomics

In the previous recipe, we saw how to safely access a common resource from different threads. But in that recipe, we were doing two system calls (in locking and unlocking `mutex`) to just get the value from an integer:

```
{   // Critical section begin.
    boost::lock_guard<boost::mutex> lock(i_mutex);
    i_snapshot = ++ shared_i;
}   // Critical section end.
```

This looks lame and slow! Can we make the code from the previous recipe better?

Getting ready

Reading the first recipe is all you need to start with this one. Some basic knowledge of multithreading will be welcome.

How to do it…

Let's see how to improve our previous example:

1. Now, we need different headers:

```
#include <cassert>
#include <cstddef>
#include <iostream>

#include <boost/thread/thread.hpp>
#include <boost/atomic.hpp>
```

2. Changing the type of `shared_i` is required:

```
boost::atomic<int> shared_i(0);
```

3. Remove all the `boost::lock_guard` variables:

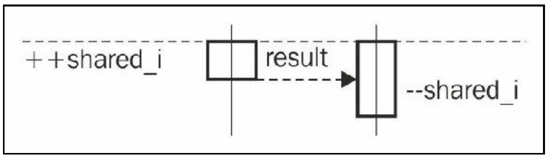
```
void do_inc() {
    for (std::size_t i = 0; i < 30000; ++i) {
        const int i_snapshot = ++ shared_i;

        // Do some work with i_snapshot.
        // ...
    }
}

void do_dec() {
    for (std::size_t i = 0; i < 30000; ++i) {
        const int i_snapshot = -- shared_i;

        // Do some work with i_snapshot.
        // ...
    }
}
```

4. That's it! Now, it works:

```
int main() {
    boost::thread t1(&do_inc);
    boost::thread t2(&do_dec);

    t1.join();
    t2.join();

    assert(shared_i == 0);
    std::cout << "shared_i == " << shared_i << std::endl;

    assert(shared_i.is_lock_free());
}
```

How it works…

Processors provide specific **atomic operations** that cannot be interfered with by other processors or processor cores. These operations appear to occur instantaneously for a system. `Boost.Atomic` provides classes that wrap around system-specific atomic operations, cooperate with the compiler to disable optimizations that may break multithreaded work with a variable, and provide a unified portable interface to work with atomic operations. If two atomic operations on the same memory location start simultaneously from different threads, one of the operations waits till the other one finishes and then reuses the result of the previous operation.



In other words, it is safe to use the `boost::atomic<>` variables from different threads simultaneously. Each operation on the atomic variable is seen by the system as a single transaction. Series of operations on the atomic variables are treated by the system as a series of independent transactions:

```
--shared_i;    // Transaction #1
// Some other may change value of `shared_i`!!
++shared_i;    // Transaction #2
```

Never ever avoid synchronization for a variable that is modified from multiple threads. Even if the variable is a `bool` and all you do is read or write `true`/`false` to it! The compiler has all the rights to optimize away all the stores and reads, breaking your code in a million ways that nobody can even imagine. Guess whom a good employer would punish for such breakage? (The compiler is not the right answer to that question!)

There’s more…

The `Boost.Atomic` library can work only with POD types; otherwise, behavior is undefined. Some platforms/processors do not provide atomic operations for some types, so `Boost.Atomic` emulates atomic behavior using `boost::mutex`. The atomic type does not use `boost::mutex` if the type specific macro is set to `2`:

```
#include <boost/static_assert.hpp>
BOOST_STATIC_ASSERT(BOOST_ATOMIC_INT_LOCK_FREE == 2);
```

The `boost::atomic<T>::is_lock_free` member function depends on runtime, so it is not good for compile-time checks but may provide a more readable syntax when the runtime check is enough:

```
assert(shared_i.is_lock_free());
```

Atomics work much faster than mutexes but are still much slower than non-atomic operations. If we compare the execution time of a recipe that uses mutexes (0:00.08 seconds) and the execution time of the preceding example in this recipe (0:00.02 seconds), we’ll see the difference (tested on 30,0000 iterations).

All the known to the book author standard library implementations had issues with atomic operations. All of them! Do not write your own atomics. If you think that your own implementation of atomics would be better and you wish to waste some time—write it, check it using special tools, and think again. Repeat until you understand that you’re wrong.

The C++11 compatible compilers should have all the atomic classes, `typedefs`, and macro in the `<atomic>` header in the `std::` namespace. Compiler-specific implementations of `std::atomic` may work faster than the Boost’s version, if the compiler correctly supports the C++11 memory model and is specially trained to optimize `std::atomic` variables.

See also

The official documentation at http://boost.org/libs/atomic may give you many more examples and some theoretical information on the topic.

Creating work_queue class

Let’s call the functional object that takes no arguments (a task, for short).

```
typedef boost::function<void()> task_t;
```

Now, imagine a situation where we have threads that post tasks and threads that execute posted tasks. We need to design a class that can be safely used by both types of those threads. This class must have functions for:

- Getting a task or waiting for a task till it is posted by another thread
- Checking and getting a task if we have one (returning an empty task if no tasks remain)
- Posting tasks

Getting ready

Make sure that you feel comfortable with `boost::thread` or `std::thread` , know basics of mutexes, and are aware of `boost::function` or `std::function`.

How to do it…

The class that we are going to implement is close by functionality to `std::queue<task_t>` but also has thread synchronization. Let's start:

1. We need the following headers and members:

```
#include <deque>
#include <boost/function.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>
#include <boost/thread/condition_variable.hpp>

class work_queue {
public:
    typedef boost::function<void()> task_type;type;

private:
    std::deque<task_type> tasks_;
    boost::mutex tasks_mutex_;
    boost::condition_variable cond_;
```

2. A function for putting a task in a queue must look like this:

```
public:
    void push_task(const task_type& task) {
        boost::unique_lock<boost::mutex> lock(tasks_mutex_);
        tasks_.push_back(task);
        lock.unlock();

        cond_.notify_one();
    }
```

3. A non-blocking function for getting a pushed task or an empty task (if no tasks remain):

```
    task_type try_pop_task() {
        task_type ret;
        boost::lock_guard<boost::mutex> lock(tasks_mutex_);
        if (!tasks_.empty()) {
            ret = tasks_.front();
            tasks_.pop_front();
        }

        return ret;
    }
```

4. Blocking function for getting a pushed task or for blocking while the task is pushed by another thread:

```
    task_type pop_task() {
        boost::unique_lock<boost::mutex> lock(tasks_mutex_);
        while (tasks_.empty()) {
            cond_.wait(lock);
        }

        task_type ret = tasks_.front();
        tasks_.pop_front();

        return ret;
    }
};
```

Here’s how a `work_queue` class may be used:

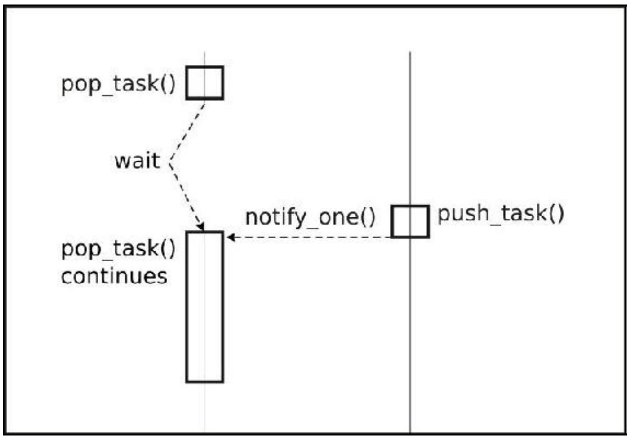
```
#include <boost/thread/thread.hpp>

work_queue g_queue;

void some_task();
const std::size_t tests_tasks_count = 3000 /*000*/;

void pusher() {
    for (std::size_t i = 0; i < tests_tasks_count; ++i) {
        g_queue.push_task(&some_task);
    }
}

void popper_sync() {
    for (std::size_t i = 0; i < tests_tasks_count; ++i) {
        work_queue::task_type t = g_queue.pop_task();
        t();         // Executing task.
    }
}

int main() {
    boost::thread pop_sync1(&popper_sync);
    boost::thread pop_sync2(&popper_sync);
    boost::thread pop_sync3(&popper_sync);

    boost::thread push1(&pusher);
    boost::thread push2(&pusher);
    boost::thread push3(&pusher);

    // Waiting for all the tasks to push.
    push1.join();
    push2.join();
    push3.join();
    g_queue.flush();

    // Waiting for all the tasks to pop.
    pop_sync1.join();
    pop_sync2.join();
    pop_sync3.join();


    // Asserting that no tasks remained,
    // and falling though without blocking.
    assert(!g_queue.try_pop_task());

    g_queue.push_task(&some_task);

    // Asserting that there is a task,
    // and falling though without blocking.
    assert(g_queue.try_pop_task());
}
```

How it works…

In this example, we see a new RAII class `boost::unique_lock`. It is just a `boost::lock_guard` class with additional functionality for explicit unlocking and locking mutexes.

Back to our `work_queue` class. Let's start with the `pop_task()` function. In the beginning, we are acquiring a lock and checking for available tasks. If there is a task, we return it; otherwise, `cond_.wait(lock)` is called. This method atomically unlocks the lock and pauses the thread of execution till some other thread notifies the current thread.

Now, let's take a look at the `push_task` method. In it, we also acquire a lock, push a task in `tasks_.queue`, unlock the lock, and call `cond_.notify_one()`, which wakes up the thread (if any) waiting in `cond_.wait(lock)`. So, after that, if some thread is waiting on a conditional variable in a `pop_task()` method, the thread will continue its execution, call `lock.lock()` deep inside `cond_.wait(lock)`, and check `tasks_empty()` in `while`. Because we just added a task in `tasks_`, we'll get out from the `while` loop, unlock the `<mutex>` (the `lock` variable gets out of scope), and return a task.



You must check conditions in a loop, not just in an `if` statement! The `if` statement leads to errors, as the operating system sometimes may wake up the threads without any notify calls from the user.

There’s more…

Note that we explicitly unlocked mutex before calling `notify_one()`. However, without unlocking, our example still works.

But, in that case, the thread that has woken up may be blocked once again during an attempt to call `lock.lock()` deep inside `cond_wait(lock)`, which leads to more context switches and worse performance.

With `tests_tasks_count` set to `3000000` and without explicit unlocking, this example runs for 7 seconds:

```
$time -f E ./work_queue
0:07.38
```

With explicit unlocking, this example runs for 5 seconds:

```
$ time -f E ./work_queue
0:05.39
```

You may also notify all the threads waiting on a specific conditional variable using `cond_.notify_all()`.

Some extremely exotic operating systems had an extremely rare issue with calling `notify_one()` outside the critical section (without holding a lock) on Boost before version 1.64 https://github.com/boostorg/thread/pull/105*. It’s doubtful that you will ever work with those. But, anyway, to avoid issues on those platforms, you may add a `flush()` function to the `work_queue` class that holds a lock and calls `notify_all()`:*

```
void flush() {
boost::lock_guard<boost::mutex> lock(tasks_mutex_);
cond_.notify_all();
}
```

Call `flush()` when you are done with pushing tasks in a queue to force the wakeup of all the threads.

The C++11 standard has `std::condition_variable` declared in the `<condition_variable>` header and `std::unique_lock` declared in the `<mutex>` header. Use the Boost version if you use C++03 compiler or just use some of the Boost’s extensions.

*The `work_queue` class could be significantly improved by adding support for **rvalue references** and calling `std::move(tasks_.front())`. This will make the code in the critical section much faster, resulting in less threads, suspends, and wakeups, less cache misses and a much better performance.*

See also

- The first three recipes in this chapter provide a lot of useful information about `Boost.Thread`
- The official documentation may give you many more examples and some theoretical information on the topic; it can be found at http://boost.org/libs/thread

Multiple-readers-single-writer lock

Imagine that we are developing some online services. We have an unordered map of registered users with some properties for each user. This set is accessed by many threads, but it is very rarely modified. All operations with the following set are done in a thread-safe manner:

```
#include <unordered_map>
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>

struct user_info {
    std::string address;
    unsigned short age;

    // Other parameters
    // ...
};

class users_online {
    typedef boost::mutex mutex_t;

    mutable mutex_t                             users_mutex_;
    std::unordered_map<std::string, user_info>  users_;

public:
    bool is_online(const std::string& username) const {
        boost::lock_guard<mutex_t> lock(users_mutex_);
        return users_.find(username) != users_.end();
    }

    std::string get_address(const std::string& username) const {
        boost::lock_guard<mutex_t> lock(users_mutex_);
        return users_.at(username).address;
    }

    void set_online(const std::string& username, user_info&& data) {
        boost::lock_guard<mutex_t> lock(users_mutex_);
        users_.emplace(username, std::move(data));
    }

    // Other methods:
    // ...
};
```

Unfortunately, our online service is somehow slow and the profilers show that the problem is in the `users_online` class. Any operation acquires a unique lock on the `mutex_` variable, so even getting resources results in waiting on a locked mutex. As some of the resources are hard to copy, the critical sections consume a lot of time, slowing down any operation on the `users_online` class.

Unfortunately, the project requirements do not allow us to redesign the class. Can we speed it up without interface changes?

Getting ready

Make sure that you feel comfortable with `boost::thread` or `std::thread` and know the basics of mutexes.

How to do it…

This will probably help:

Replace `boost::mutex` with `boost::shared_mutex`. Replace `boost::unique_locks` with `boost::shared_lock` for methods that do not modify data:

```
#include <boost/thread/shared_mutex.hpp>

class users_online {
    typedef boost::shared_mutex mutex_t;

    mutable mutex_t                             users_mutex_;
    std::unordered_map<std::string, user_info>  users_;

public:
    bool is_online(const std::string& username) const {
        boost::shared_lock<mutex_t> lock(users_mutex_);
        return users_.find(username) != users_.end();
    }

    std::string get_address(const std::string& username) const {
        boost::shared_guard<mutex_t> lock(users_mutex_);
        return users_.at(username).address;
    }

    void set_online(const std::string& username, user_info&& data) {
        boost::lock_guard<mutex_t> lock(users_mutex_);
        users_.emplace(username, std::move(data));
    }

    // Other methods:
    // ...
};
```

How it works…

We can allow getting the data from multiple threads simultaneously if those threads do not modify data. We need to uniquely own the mutex only if we are going to modify the data protected by it. In all other situations, simultaneous access to data is allowed. And that is what `boost::shared_mutex` was designed for. It allows shared locking (read locking), which allows multiple simultaneous access to resources.

When we do try to unique lock a resource that is shared locked, operations will be blocked till there are no read locks remaining, and only after that resource is unique locked, forcing new shared locks to wait until the unique lock is released. `boost::shared_lock` locking for reading and writing is much slower than the usual `boost::mutex` locking. Do not use `boost::shared_lock` unless you are sure that there’s no good way to redesign your code and you are sure that `boost::shared_lock` will speed things up.

Some readers may see the `mutable` keyword for the first time. This keyword can be applied to non-static and non-constant class members. The `mutable` data member can be modified in the constant member functions and is usually used for mutexes and other helper variables that are not directly related to the class logic.

There’s more…

When you do need only unique locks, do not use `boost::shared_mutex` because it is slower than a usual `boost::mutex` class.

Shared mutexes were not available in C++ until C++14. `shared_timed_mutex` and `shared_lock` are defined in the `<shared_mutex>` header in `std::` namespace. They have performance characteristics close to the Boost versions, so apply all the preceding performance notes.

C++17 has a `shared_mutex` that may be slightly faster than `shared_timed_mutex`, because it dose not provide the means for timed locking. This may save you a few precious nanoseconds.

See also

- There is also a `boost::upgrade_mutex` class, which may be useful for cases when shared lock needs promotion to unique lock. See the `Boost.Thread` documentation at http://boost.org/libs/thread for more information.
- Refer to http://herbsutter.com/2013/01/01/video-you-dont-know-const-and-mutable/ for more information about the mutable keyword.

Creating variables that are unique per thread

Let’s take a glance at the recipe *Creating work_queue class*. Each task there can be executed in one of the many threads and we do not know in which one. Imagine that we want to send the results of an executed task using some connection:

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

class connection: boost::noncopyable {
public:
    // Opening a connection is a slow operation
    void open();

    void send_result(int result);

    // Other methods
    // ...
};
```

We have the following solutions:

- Open a new connection when we need to send the data (which is very slow)
- Have a single connection for all the threads and wrap them in mutex (which is also slow)
- Have a pool of connections, get a connection from it in a thread-safe manner, and use it (a lot of coding is required, but this solution is fast)
- Have a single connection per thread (fast and simple to implement)

So, how can we implement the last solution?

Getting ready

Basic knowledge of threads is required.

How to do it…

It is time to make a thread local variable. Declare a function in a header file after the `connection` class definition:

```
connection& get_connection();
```

Make your source file look like this:

```
#include <boost/thread/tss.hpp>
boost::thread_specific_ptr<connection> connection_ptr;

connection& get_connection() {
    connection* p = connection_ptr.get();
    if (!p) {
        connection_ptr.reset(new connection);
        p = connection_ptr.get();
        p->open();
    }

    return *p;
}
```

Done. Using a thread-specific resource was never so easy:

```
void task() {
    int result;
    // Some computations go there.
    // ...

    // Sending the result:
    get_connection().send_result(result);
}
```

How it works…

The `boost::thread_specific_ptr` variable holds a separate pointer for each thread. Initially, this pointer is equal to `nullptr`; that is why we check for `!p` and open a connection if it is `nullptr`.

So, when we enter `get_connection()` from the thread that already initiated the pointer, `!p` return the value `false` and we return the already opened connection.

`delete` for the pointer stored inside the `connection_ptr` variable is called when the thread is exiting, so we do not need to worry about memory leaks.

There’s more…

You may provide your own cleanup function that will be called instead of `delete` at thread exit. A cleanup function must have the `void (*cleanup_function)(T*)` signature and must be passed during the `boost::thread_specific_ptr` construction.

C++11 has a special keyword, `thread_local`, to declare variables with thread local storage duration. C++11 has no `thread_specific_ptr` class, but you may use `thread_local T` or `thread_local std::unique_ptr<T>` to achieve the same behavior on compilers that support `thread_local`. `boost::thread_specific_ptr` works on pre-C++11 compilers, unlike `thread_local`.

C++17 has `inline` variables, and you may use `thread_local` with `inline` to declare thread local variables in header files.

See also

- The `Boost.Thread` documentation gives a lot of good examples on different cases; it can be found at http://boost.org/libs/thread
- Reading this topic at http://stackoverflow.com/questions/13106049/c11-gcc-4-8-thread-local-performance-penalty.html and about the GCCs `__thread` keyword at http://gcc.gnu.org/onlinedocs/gcc-3.3.1/gcc/Thread-Local.html may give you some ideas about how `thread_local` is implemented in compilers and how fast it is

Interrupting a thread

Sometimes, we need to kill a thread that has eaten too many resources or that is just executing for too long. For example, some parser works in a thread (and actively uses `Boost.Thread`), but we have already got the required amount of data from it, so parsing can be stopped. Here’s the stub:

```
int main() {
    boost::thread parser_thread(&do_parse);

    // ...

    if (stop_parsing) {
        // No more parsing required.
        // TODO: Stop the parser!
    }

    // ...

    parser_thread.join();
}
```

How can we do it?

Getting ready

Almost nothing is required for this recipe. You only need to have at least a basic knowledge of threads.

How to do it…

We can stop a thread by interrupting it:

```
if (stop_parsing) {
    // No more parsing required.
    parser_thread.interrupt();
}
```

How it works…

`Boost.Thread` provides some predefined **interruption points** in which the thread is checked for being interrupted via the `interrupt()` call. If the thread is interrupted, the exception `boost::thread_interrupted` is thrown. While the exception is propagated through the `do_parse()` internals, it calls destructors for all the resources, just like a typical exception does. `boost::thread_interrupted` exceptions are treated specially by the `Boost.Thread` library, and for that exception, it is allowed to leave the thread function (`do_parse()` in our example). When the exception leaves the thread function, it is caught by the `boost::thread` internals and treated as a request to cancel the thread.

> *`boost::thread_interrupted` is not derived from `std::exception`! Interruptions work well if you catch exceptions by their type or by references to `std::exception`. But if you catch an exception by `catch (...)` and do not rethrow it, the interruptions won't work.*

As we know from the first recipe in this chapter, if a function passed into a thread does not catch an exception and the exception leaves function bounds, the application terminates. `boost::thread_interrupted` is the only exception to that rule; it may leave function bounds and does not `std::terminate()` the application.

There’s more…

Interruption points of the `Boost.Thread` library are listed in official documentation. As a rule of a thumb everything that blocks checks for interruptions.

We may also manually add interruption points at any place. All we need is to call `boost::this_thread::interruption_point()`:

```
void do_parse() {
    while (not_end_of_parsing) {
        // If current thread was interrupted, the following
        // line will throw an boost::thread_interrupted.
        boost::this_thread::interruption_point();

        // Some parsing goes here.
        // ...
    }
}
```

If interruptions are not required for a project, defining `BOOST_THREAD_DONT_PROVIDE_INTERRUPTIONS` gives a small performance boost and totally disables thread interruptions.

C++11 has no thread interruptions, but you can partially emulate them using atomic operations:

- Create an atomic `bool` variable
- Check the atomic variable in the thread and throw an exception if it has changed
- Do not forget to catch that exception in the function passed to the thread (otherwise your application will terminate)

However, this won’t help you if the code is waiting somewhere in a conditional variable or in a sleep method.

See also

- The official documentation for `Boost.Thread` provides a list of predefined interruption points at http://www.boost.org/doc/libs/1_64_0/doc/html/thread/thread_management.html#thread.thread_management.tutorial.interruption.predefined_interruption_points
- As an exercise, see the other recipes from this chapter and think of where additional interruption points would improve the code
- Reading other parts of the `Boost.Thread` documentation may be useful; go to http://boost.org/libs/thread

Manipulating a group of threads

Those readers who were trying to repeat all the examples by themselves, or those who were experimenting with threads must already be bored with writing the following code to launch and join threads:

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

void some_function();

void sample() {
    boost::thread t1(&some_function);
    boost::thread t2(&some_function);
    boost::thread t3(&some_function);

    // ...

    t1.join();
    t2.join();
    t3.join();
}
```

Maybe there is a better way to do this?

Getting ready

Basic knowledge of threads will be more than enough for this recipe.

How to do it…

We may manipulate a group of threads using the `boost::thread_group` class.

1. Construct a `boost::thread_group` variable:

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

int main() {
    boost::thread_group threads;
```

2. Create threads into the preceding variable:

```
    // Launching 10 threads.
    for (unsigned i = 0; i < 10; ++i) {
        threads.create_thread(&some_function);
    }
```

3. Now, you may call functions for all the threads inside `boost::thread_group`:

```
    // Joining all threads.
    threads.join_all();

    // We can also interrupt all of them
    // by calling threads.interrupt_all();
}
```

How it works…

The `boost::thread_group` variable just holds all the threads constructed or moved to it and may send some calls to all the threads.

There’s more…

C++11 has no `thread_group` class; it’s Boost specific.

See also

The official documentation of `Boost.Thread` may surprise you with a lot of other useful classes that are not described in this chapter; go to http://boost.org/libs/thread.

Initializing a shared variable safely

Imagine that we are designing a safety-critical class that is used from multiple threads, receives answers from a server, postprocesses them, and outputs the response:

```
struct postprocessor {
    typedef std::vector<std::string> answer_t;

    // Concurrent calls on the same variable are safe.
    answer_t act(const std::string& in) const {
        if (in.empty()) {
            // Extremely rare condition.
            return read_defaults();
        }

        // ...
    }
};
```

Note the `return read_defaults();` line. There may be situations when server does not respond because of networking issues or some other problems. In those cases, we attempt to read defaults from file:

```
// Executes for a long time.
std::vector<std::string> read_defaults();
```

From the preceding code, we hit the problem: the server may be unreachable for some noticeable time, and for all that time we'll be rereading the file on each `act` call. This significantly affects performance.

We can attempt to fix it by storing `default_` inside the class:

```
struct postprocessor {
    typedef std::vector<std::string> answer_t;

private:
    answer_t default_;

public:
    postprocessor()
        : default_(read_defaults())
    {}

    // Concurrent calls on the same variable are safe.
    answer_t act(const std::string& in) const {
        if (in.empty()) {
            // Extremely rare condition.
            return default_;
        }

        // ...
    }
};
```

That's also not a perfect solution: we do not know how many instances of `postprocessor` class are constructed by the user and we are wasting memory on defaults that may not be required during the run.

So, we have to concurrent-safely read and store the data in the current instance on the first remote server failure and do not read it again on the next failures. There are many ways to do that, but let's look at the most right one.

Getting ready

Basic knowledge of threads is more than enough for this recipe.

How to do it…

1. We have to add variables for storing information that defaults were initialized and a variable for storing the defaults:

   ```
   #include <boost/thread/once.hpp>

   struct postprocessor {
       typedef std::vector<std::string> answer_t;

   private:
       mutable boost::once_flag default_flag_;
       mutable answer_t default_;
   ```

   Variables are `mutable` because we are going to modify them inside `const` member functions.

2. Let’s initialize our variables:

   ```
   public:
       postprocessor()
           : default_flag_(BOOST_ONCE_INIT)
           , default_()
       {}
   ```

3. Finally, let’s change the `act` function:

   ```
       // Concurrent calls on the same variable are safe.
       answer_t act(const std::string& in) const {
           answer_t ret;
           if (in.empty()) {
               // Extremely rare condition.
               boost::call_once(default_flag_, [this]() {
                   this->default_ = read_defaults();
               });
               return default_;
           }

           // ...
           return ret;
       }
   };
   ```

How it works…

In short, `boost::call_once` and `boost::once_flag` make sure that the function passed as a second parameter is executed only once.

The `boost::call_once` function synchronizes calls to the function *F* passed as a second argument. `boost::call_once` and `boost::once_flag` make sure that only one call to the function *F* progresses if there are two or more concurrent calls on the same `once_flag` and make sure that only once successful call to *F* is performed.

If the call to function *F* has not thrown exceptions that left the body of *F*, then `boost::call_once` assumes that the call was successful and stores that information inside the `boost::once_flag`. Any subsequent calls to `boost::call_once` with the same `boost::once_flag` do nothing.

Do not forget to initialize the `boost::once_flag` *with the* `BOOST_ONCE_INIT` *macro.*

There’s more..

The boost::call_once may pass parameters to the function to call:

```
#include <iostream>

void once_printer(int i) {
    static boost::once_flag flag = BOOST_ONCE_INIT;
    boost::call_once(
        flag,
        [](int v) { std::cout << "Print once " << v << '\n'; },
        i // <=== Passed to lambda from above.
    );

    // ...
}
```

Now, if we call once_printer function in a loop:

```
int main() {
    for (unsigned i = 0; i < 10; ++i) {
        once_printer(i);
    }
}
```

Only a single line will be in the output:

```
        Print once 0
```

C++11 has a std::call_once and std::once_flag in the <mutex> header. Unlike the Boost version, the standard library version of the once_flag does not require initialization via a macro, it has a constexpr constructor. As usual, Boost version is usable on pre-C++11 compilers, so use it if you have to support old compilers.

TIP *Visual Studio before 2015 was shipping a suboptimal std::call_once implementation more than ten times slower than the Boost’s version. Stick to the boost::call_once if you’re not using modern compilers.*

See also

The `Boost.Thread` documentation gives a lot of good examples on different cases. It can be found at http://boost.org/libs/thread.

Locking multiple mutexes

For the next few paragraphs, you'll be one of the people who write games. Congratulations, you can play at work!

You're developing a server and you have to write code for exchanging loot between two users:

```
class user {
    boost::mutex        loot_mutex_;
    std::vector<item_t> loot_;
public:
    // ...

    void exchange_loot(user& u);
};
```

Each user action could be concurrently processed by different threads on a server, so you have to guard the resources by mutexes. The junior developer tried to deal with the problem, but his solution does not work:

```
void user::exchange_loot(user& u) {
    // Terribly wrong!!! ABBA deadlocks.
    boost::lock_guard<boost::mutex> l0(loot_mutex_);
    boost::lock_guard<boost::mutex> l1(u.loot_mutex_);
    loot_.swap(u.loot_);
}
```

The issue in the preceding code is a well-known **ABBA deadlock** problem. Imagine that *thread 1* locks *mutex A* and *thread 2* locks *mutex B*. And now *thread 1* attempts to lock the already locked *mutex B* and *thread 2* attempts to lock the already locked *mutex A*. This results in two threads locked for infinity by each other, as they need a resource locked by other thread to proceed while the other thread waits for a resource owned by the current thread.

Now, if user1 and user2 call `exchange_loot` for each other concurrently, then we may end up with a situation that `user1.exchange_loot(user2)` calls locked `user1.loot_mutex_` and `user2.exchange_loot(user1)` calls locked `user2.loot_mutex_`. `user1.exchange_loot(user2)` waits for infinity in attempt to lock `user2.loot_mutex_` and `user2.exchange_loot(user1)` waits for infinity in an attempt to lock `user1.loot_mutex_`.

Getting ready

Basic knowledge of threads and mutexes is enough for this recipe.

How to do it…

There are two major out-of-the-box solutions to that problem:

1. The short one that requires variadic template support from the compiler:

```
#include <boost/thread/lock_factories.hpp>

void user::exchange_loot(user& u) {
    typedef boost::unique_lock<boost::mutex> lock_t;

    std::tuple<lock_t, lock_t> l = boost::make_unique_locks(
        loot_mutex_, u.loot_mutex_
    );

    loot_.swap(u.loot_);
}
```

   The same code using using `auto`:

```
#include <boost/thread/lock_factories.hpp>

void user::exchange_loot(user& u) {
    auto l = boost::make_unique_locks(
        loot_mutex_, u.loot_mutex_
    );

    loot_.swap(u.loot_);
}
```

2. The portable solution:

```
#include <boost/thread/locks.hpp>

void user::exchange_loot(user& u) {
    typedef boost::unique_lock<boost::mutex> lock_t;

    lock_t l0(loot_mutex_, boost::defer_lock);
    lock_t l1(u.loot_mutex_, boost::defer_lock);
    boost::lock(l0, l1);

    loot_.swap(u.loot_);
}
```

How it works…

The core idea is to order mutexes somehow and lock them always following that particular order. In that case, there’s no ABBA problem possible, as all the threads would always lock mutex *A* before *B*. Usually, other deadlock avoidance algorithms are used but for the simplicity of the example here, we assume that the ordering of mutexes is used.

In the first example, we used `boost::make_unique_locks` that always locks threads in some particular order and returns a tuple that holds the locks.

In the second example, we created the locks manually but have not locked them thanks to a passed `boost::defer_lock` parameter. The actual locking happened in the `boost::lock(l0, l1)` call, which locked the mutexes in some predefined order.

Now, if `user1` and `user2` call `exchange_loot` for each other concurrently, then both `user1.exchange_loot(user2)` and `user2.exchange_loot(user1)` calls will try to lock `user1.loot_mutex_` first or both will try to lock `user2.loot_mutex_` first. That depends on a runtime.

There’s more…

`boost::make_unique_locks` and `boost::lock` functions may accept more than 2 locks or mutexes, so you could use them in more advanced cases, where more than two mutexes must be locked simultaneously.

C++11 has a `std::lock` function defined in the header `<mutex>` that behaves exactly like the `boost::lock` function.

C++17 has a much more beautiful solution:

```
#include <mutex>

void user::exchange_loot(user& u) {
    std::scoped_lock l(loot_mutex_, u.loot_mutex_);
    loot_.swap(u.loot_);
}
```

In the preceding code, `std::scoped_lock` is a class that accepts a variadic amount of locks. It has variadic template parameters that are automatically deduced from the C++17 deduction guide. The actual type of the `std::scoped_lock` from the preceding example is:

```
std::scoped_lock<std::mutex, std::mutex>
```

The `std::scoped_lock` holds a lock to all the mutexes passed during construction and avoids deadlocks. In other words, it works like the first example, but looks slightly better.

See also

The official documentation of `Boost.Thread` may surprise you with a lot of other useful classes that were not described in this chapter; go to http://boost.org/libs/thread.

Manipulating Tasks

In this chapter, we will cover:

- Registering a task for an arbitrary data type processing
- Making timers and processing timer events as tasks
- Network communication as a task
- Accepting incoming connections
- Executing different tasks in parallel
- Pipeline tasks processing
- Making a nonblocking barrier
- Storing an exception and making a task from it
- Getting and processing system signals as tasks

Introduction

This chapter is all about tasks. We'll be calling the functional object a *task* because it is shorter and better reflects what it will do. The main idea of this chapter is that we can split all the processing, computations, and interactions to functors (tasks), and process each of those tasks almost independently. Moreover, we may not block on some slow operations such as receiving data from the socket or waiting for the time out, but instead provide a callback task and continue working with other tasks. Once the OS finishes, the slow operation, our callback is executed.

The best way to understand the example is to play with it by modifying, running, and extending it. The site, http://apolukhin.github.io/Boost-Cookbook/, *has all the examples from this chapter, and you can even play with some of them online.*

Before you start

This chapter requires at least a basic knowledge of the first, second, and fifth chapters. Basic knowledge on C++11 rvalue references and lambdas is required.

Registering a task for an arbitrary data type processing

First of all, let’s take care of the class that holds all the tasks and provides methods for their execution. We were already doing something like this in the Chapter 5, *Multithreading, Creating a work_queue class* recipe, but some of the following problems were not addressed:

- A `work_queue` class was only storing and returning tasks, but we also need to execute existing tasks.
- A task may throw an exception. We need to catch and process exceptions if they leave the task boundaries.
- A task may not notice a thread interruption. The next task on the same thread may get the interruption instead.
- We need a way to stop the processing of the tasks.

Getting ready

This recipe requires linking with the `boost_system` and `boost_thread` libraries. A basic knowledge of `Boost.Thread` is also required.

How to do it…

In this recipe, we use `boost::asio::io_service` instead of `work_queue` from the previous chapter. There is a reason for doing this, and we'll see it in the following recipes.

1. Let's start from the structure that wraps around a user task:

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

namespace detail {

template <class T>
struct task_wrapped {
private:
    T task_unwrapped_;

public:
    explicit task_wrapped(const T& f)
        : task_unwrapped_(f)
    {}

    void operator()() const {
        // Resetting interruption.
        try {
            boost::this_thread::interruption_point();
        } catch(const boost::thread_interrupted&){}

        try {
            // Executing task.
            task_unwrapped_();
        } catch (const std::exception& e) {
            std::cerr<< "Exception: " << e.what() << '\n';
        } catch (const boost::thread_interrupted&) {
            std::cerr<< "Thread interrupted\n";
        } catch (...) {
            std::cerr<< "Unknown exception\n";
        }
    }
};

} // namespace detail
```

2. For ease of use, we'll create a function that produces `task_wrapped` from the user's functor:

```
namespace detail {

template <class T>
task_wrapped<T> make_task_wrapped(const T& task_unwrapped) {
    return task_wrapped<T>(task_unwrapped);
}

} // namespace detail
```

3. Now, we are ready to write the `tasks_processor` class:

```
#include <boost/asio/io_service.hpp>

class tasks_processor: private boost::noncopyable {
protected:
    static boost::asio::io_service& get_ios() {
```

```
                static boost::asio::io_service ios;
                static boost::asio::io_service::work work(ios);

                return ios;
            }
```

4. Let’s add the `push_task` method:

```
        public:
            template <class T>
            static void push_task(const T& task_unwrapped) {
                get_ios().post(detail::make_task_wrapped(task_unwrapped));
            }
```

5. Let’s finish this class by adding the member functions for starting and stopping a task’s execution loop:

```
            static void start() {
                get_ios().run();
            }

            static void stop() {
                get_ios().stop();
            }
        }; // tasks_processor
```

Done! Now, it is time to test our class:

```
int func_test() {
    static int counter = 0;
    ++ counter;
    boost::this_thread::interruption_point();

    switch (counter) {
    case 3:
        throw std::logic_error("Just checking");

    case 10:
        // Emulation of thread interruption.
        // Caught inside task_wrapped and does not stop execution.
        throw boost::thread_interrupted();

    case 90:
        // Stopping the tasks_processor.
        tasks_processor::stop();
    }

    return counter;
}
```

The `main` function may look like this:

```
int main () {
    for (std::size_t i = 0; i < 100; ++i) {
        tasks_processor::push_task(&func_test);
    }

    // Processing was not started.
    assert(func_test() == 1);

    // We can also use lambda as a task.
    // Counting 2 + 2 asynchronously.
    int sum = 0;
    tasks_processor::push_task(
        [&sum]() { sum = 2 + 2; }
    );

    // Processing was not started.
    assert(sum == 0);

    // Does not throw, but blocks till
    // one of the tasks it is owning
    // calls tasks_processor::stop().
    tasks_processor::start();
    assert(func_test() == 91);
}
```

How it works…

The `boost::asio::io_service` variable can store and execute tasks posted to it. But we may not post a user's tasks to it directly, because they may receive an interruption addressed to other tasks or throw an exception. That is why we wrap a user's task in the `detail::task_wrapped` structure. It resets all the previous interruptions by calling:

```
try {
    boost::this_thread::interruption_point();
} catch(const boost::thread_interrupted&){}
```

`detail::task_wrapped` executes the task in the `try{ } catch()` block making sure that no exception leaves the `operator()` bounds.

Take a look at the `start()` function. The `boost::asio::io_service::run()` starts processing tasks posted to the `io_service` variable. If `boost::asio::io_service::run()` is not called, then posted tasks are not executed (this can be seen in the `main()` function). Task processing may be stopped via a call to `boost::asio::io_service::stop()`.

The `boost::asio::io_service` class returns from the `run()` function if there are no more tasks left, so we force it to continue with the execution using an instance of `boost::asio::io_service::work`:

```
static boost::asio::io_service& get_ios() {
    static boost::asio::io_service ios;
    static boost::asio::io_service::work work(ios);

    return ios;
}
```

The `iostream` classes and variables, such as `std::cerr` and `std::cout` are not thread safe on pre C++11 compilers and may produce interleaved characters on C++11 compatible compilers. In real projects, additional synchronization must be used to get readable output. For the simplicity of an example, we do not do that.

There’s more…

The C++17 standard library has no `io_service`. However, a big part of the `Boost.Asio` library is proposed as a Networking **Technical Specification** (**TS**) as an addition to C++.

See also

- The following recipes in this chapter will show you why we choose `boost::asio::io_service` instead of using our handwritten code from Chapter 5, *Multithreading*
- You may consider the documentation of `Boost.Asio` for getting some examples, tutorials, and class references at http://boost.org/libs/asio
- You may also read the *Boost.Asio C++ Network Programming* book, which gives a smoother introduction to `Boost.Asio` and covers some details that are not covered in this book

Making timers and processing timer events as tasks

It is a common task to check something with specified intervals. For example, we need to check some sessions for an activity once in every 5 seconds. There are popular solutions for such a problem:

- The bad solution creates a thread that does the checking and then sleeps for 5 seconds. This is a lame solution that eats a lot of system resources and scales badly.
- The right solution uses system specific APIs for manipulating timers asynchronously. This is a better solution, that requires some work and is not portable, unless you use `Boost.Asio`.

Getting ready

You must know how to use C++11 rvalue-references and `unique_ptr`.

This recipe is based on the code from the previous recipe. See the first recipe of this chapter to get information about the `boost::asio::io_service` and `task_queue` classes.

Link this recipe with the `boost_system` and `boost_thread` libraries. Define `BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS` to bypass restrictive library checks.

How to do it…

We just modify the `tasks_processor` class by adding new methods to run a task at some specified time.

1. Let’s add a method to our `tasks_processor` class for the delayed running of a task:

```
class tasks_processor {
    // ...
public:
    template <class Time, class Func>
    static void run_delayed(Time duration_or_time, const Func& f) {
        std::unique_ptr<boost::asio::deadline_timer> timer(
            new boost::asio::deadline_timer(
                get_ios(), duration_or_time
            )
        );

        timer_ref.async_wait(
            detail::timer_task<Func>(
                std::move(timer),
                f
            )
        );
    }
};
```

2. As a final step, we create a `timer_task` structure:

```
#include <boost/asio/io_service.hpp>
#include <boost/asio/deadline_timer.hpp>
#include <boost/system/error_code.hpp>
#include <memory>  // std::unique_ptr
#include <iostream>

namespace detail {

    template <class Functor>
    struct timer_task {
    private:
        std::unique_ptr<boost::asio::deadline_timer> timer_;
        task_wrapped<Functor> task_;

    public:
        explicit timer_task(
                std::unique_ptr<boost::asio::deadline_timer> timer,
                const Functor& task_unwrapped)
            : timer_(std::move(timer))
            , task_(task_unwrapped)
        {}

        void operator()(const boost::system::error_code& error) const {
            if (!error) {
                task_();
            } else {
                std::cerr << error << '\n';
            }
        }
    };

} // namespace detail
```

That’s how we could use the new functionality:

```
int main () {
    const int seconds_to_wait = 3;
    int i = 0;

    tasks_processor::run_delayed(
        boost::posix_time::seconds(seconds_to_wait),
        test_functor(i)
    );

    tasks_processor::run_delayed(
        boost::posix_time::from_time_t(time(NULL) + 1),
        &test_func1
    );

    assert(i == 0);

    // Blocks till one of the tasks
    // calls tasks_processor::stop().
    tasks_processor::start();
}
```

Where `test_functor` is structure with defined `operator()` and `test_func1` is a function:

```
struct test_functor {
    int& i_;

    explicit test_functor(int& i);

    void operator()() const {
        i_ = 1;
        tasks_processor::stop();
    }
};

void test_func1();
```

How it works…

In short, when a specified amount of time is passed, `boost::asio::deadline_timer` pushes the task to the instance of `boost::asio::io_service` class for execution.

All the nasty stuff is inside the `run_delayed` function:

```
template <class Time, class Functor>
static void run_delayed(Time duration_or_time, const Functor& f) {
    std::unique_ptr<boost::asio::deadline_timer>
    timer( /* ... */ );

    boost::asio::deadline_timer& timer_ref = *timer;

    timer_ref.async_wait(
        detail::timer_task<Functor>(
            std::move(timer),
            f
        )
    );
}
```

The `tasks_processor::run_delayed` function accepts a timeout and a functor to call after the timeout. In it, a unique pointer to `boost::asio::deadline_timer` is created. `boost::asio::deadline_timer` holds platform-specific stuff for asynchronous execution of a task.

`Boost.Asio` does not manage memory out of the box. The library user has to take care of managing resources usually by keeping them in the task. So if we need a timer and want some function to execute after the specified timeout, we have to move the timer's unique pointer into the task, get a reference to the timer, and pass a task to the timer.

We are getting a reference to the `deadline_timer` in this line:

```
boost::asio::deadline_timer& timer_ref = *timer;
```

Now, we create a `detail::timer_task` object that stores a functor and gets the ownership of the `unique_ptr<boost::asio::deadline_timer>` :

```
        detail::timer_task<Functor>(
            std::move(timer),
            f
        )
```

The `boost::asio::deadline_timer` must not be destroyed until it is triggered, and moving it into the `timer_task` functor guarantees that.

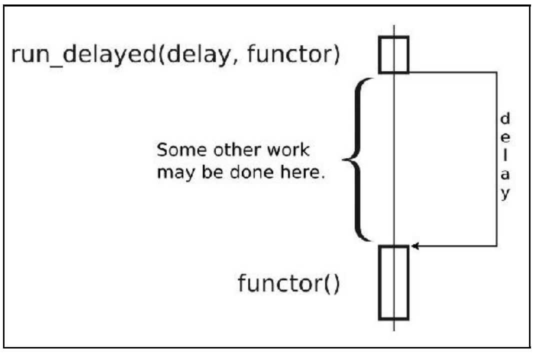
Finally, we instruct the `boost::asio::deadline_timer` to post the `timer_task` functor to the `io_service` when the requested amount of time elapses:

```
timer_ref.async_wait( /* timer_task */ )
```

Reference to the `io_service` variable is kept inside the `boost::asio::deadline_timer` variable. That's why its constructor requires a reference to `io_service` to store it and post the task to it as soon as the timeout elapses.

The `detail::timer_task::operator()` method accepts `boost::system::error_code`, which contains error description if something bad happened while waiting. If no error occurred, we call the user’s functor that is wrapped to catch exceptions (we re-use the `detail::task_wrapped` structure from the first recipe).

`boost::asio::deadline_timer::async_wait` does not consume CPU resources or thread of execution while waiting for the timeout. You may simply push some further into the `io_service` and they will start executing while the timeout is being maintained by OS:


run_delayed(delay, functor)
Some other work may be done here.
d e l a y
functor()

As a rule of thumb: all the resources that are used during the `async_*` calls must be stored in the task.

There’s more…

Some exotic/antique platforms have no APIs to implement timers in a good way, so the `Boost.Asio` library emulates the behavior of the asynchronous timer using an additional thread of execution per `io_service`. There’s just no other way to do it.

C++17 has no `Boost.Asio`-like classes in it; however, the Networking TS has `async_wait` and `timer` classes.

See also

- Reading the first recipe from this chapter will teach you the basics of `boost::asio::io_service`. The following recipes will provide you with more examples of `io_service` usage and will show you how to deal with network communications, signals, and other features using `Boost.Asio`.
- You may consider the documentation of `Boost.Asio` for getting some examples, tutorials, and class references at http://boost.org/libs/asio site.

Network communication as a task

Receiving or sending data by network is a slow operation. While packets are received by the machine, and while OS verifies them and copies the data to the user-specified buffer, multiple seconds may pass.

We may do a lot of work rather than waiting! Let’s modify our `tasks_processor` class so that it would be capable of sending and receiving data in an asynchronous manner. In nontechnical terms, we ask it to receive at least *N* bytes from the remote host and after that is done, call our functor. By the way, do not block on this call. Those readers who know about **libev**, **libevent**, or Node.js may find a lot of familiar things in this recipe.

Getting ready

This recipe is based on the previous two recipes. See the first recipe of this chapter to get information about the `boost::asio::io_service` and `task_queue` classes. See the second recipe review the basics of async processing.

Link this recipe with the `boost_system` and `boost_thread` libraries. Define `BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS` to bypass over restrictive library checks.

How to do it…

Let’s extend the code from the previous recipe by adding methods to create connections.

1. A connection would be represented by a `connection_with_data` class. This class is keeping socket to the remote host and a `std::string` for receiving and sending data:

```
#include <boost/asio/ip/tcp.hpp>
#include <boost/core/noncopyable.hpp>

struct connection_with_data: boost::noncopyable {
    boost::asio::ip::tcp::socket socket;
    std::string data;

    explicit connection_with_data(boost::asio::io_service& ios)
        : socket(ios)
    {}

    void shutdown() {
        if (!socket.is_open()) {
            return;
        }

        boost::system::error_code ignore;
        socket.shutdown(
            boost::asio::ip::tcp::socket::shutdown_both,
            ignore
        );
        socket.close(ignore);
    }

    ~connection_with_data() {
        shutdown();
    }
};
```

2. Just like in the previous recipe, class would be mostly used by unique pointer to it. Let’s add a `typedef` for simplicity:

```
#include <memory> // std::unique_ptr

typedef std::unique_ptr<connection_with_data> connection_ptr;
```

3. The `tasks_processor` class from previous recipe owns the `boost::asio::io_service` object. It seems reasonable to make it a factory for constructing connections:

```
class tasks_processor {
    // ...
public:
    static connection_ptr create_connection(
        const char* addr,
        unsigned short port_num)
    {
        connection_ptr c( new connection_with_data(get_ios()) );

        c->socket.connect(boost::asio::ip::tcp::endpoint(
            boost::asio::ip::address_v4::from_string(addr),
            port_num
        ));

        return c;
    }
};
```

4. The following is the methods for async writing data to remote host:

```
#include <boost/asio/write.hpp>

template <class T>
struct task_wrapped_with_connection;

template <class Functor>
void async_write_data(connection_ptr&& c, const Functor& f) {
    boost::asio::ip::tcp::socket& s = c->socket;
    std::string& d = c->data;

    boost::asio::async_write(
        s,
        boost::asio::buffer(d),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}
```

5. The following is the methods for async reading data from remote host:

```
#include <boost/asio/read.hpp>

template <class Functor>
void async_read_data(
    connection_ptr&& c,
    const Functor& f,
    std::size_t at_least_bytes)
{
    c->data.resize(at_least_bytes);
    c->data.resize(at_least_bytes);

    boost::asio::ip::tcp::socket& s = c->socket;
    std::string& d = c->data;
    char* p = (d.empty() ? 0 : &d[0]);

    boost::asio::async_read(
        s,
        boost::asio::buffer(p, d.size()),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}

template <class Functor>
void async_read_data_at_least(
    connection_ptr&& c,
    const Functor& f,
    std::size_t at_least_bytes,
    std::size_t at_most)
{
    std::string& d = c->data;
    d.resize(at_most);
    char* p = (at_most == 0 ? 0 : &d[0]);

    boost::asio::ip::tcp::socket& s = c->socket;

    boost::asio::async_read(
        s,
        boost::asio::buffer(p, at_most),
        boost::asio::transfer_at_least(at_least_bytes),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}
```

6. The final part is the `task_wrapped_with_connection` class definition:

```
template <class T>
struct task_wrapped_with_connection {
private:
```

```
            connection_ptr c_;
            T task_unwrapped_;

        public:
            explicit task_wrapped_with_connection
            (connection_ptr&& c, const T& f)
                : c_(std::move(c))
                , task_unwrapped_(f)
            {}

            void operator()(
                const boost::system::error_code& error,
                std::size_t bytes_count)
            {
                c_->data.resize(bytes_count);
                task_unwrapped_(std::move(c_), error);
            }
        };
```

Done! Now, the library user can use the preceding class like this to send the data:

```
void send_auth() {
    connection_ptr soc = tasks_processor::create_connection(
        "127.0.0.1", g_port_num
    );
    soc->data = "auth_name";

    async_write_data(
        std::move(soc),
        &on_send
    );
}
```

Users may also use it like this to receive data:

```
void receive_auth_response(
    connection_ptr&& soc,
    const boost::system::error_code& err)
{
    if (err) {
        std::cerr << "Error on sending data: "
        << err.message() << '\n';
        assert(false);
    }

    async_read_data(
        std::move(soc),
        &process_server_response,
        2
    );
}
```

This is how a library user may handle the received data:

```
void process_server_response(
        connection_ptr&& soc,
        const boost::system::error_code& err)
{
    if (err && err != boost::asio::error::eof) {
        std::cerr << "Client error on receive: "
        << err.message() << '\n';
        assert(false);
    }

    if (soc->data.size() != 2) {
        std::cerr << "Wrong bytes count\n";
        assert(false);
    }

    if (soc->data != "OK") {
        std::cerr << "Wrong response: " << soc->data << '\n';
        assert(false);
    }

    soc->shutdown();
    tasks_processor::stop();
}
```

How it works…

The `Boost.Asio` library does not manage resources and buffers out of the box. So, if we want some simple interface for reading and writing data, the simplest solution would be to tie together the socket and buffer for sending/receiving data. That's what the `connection_with_data` class does. It holds a `boost::asio::ip::tcp::socket`, which is a `Boost.Asio` wrapper around native sockets and a `std::string` variable that we use as a buffer.

A constructor of the `boost::asio::ip::tcp::socket` class accepts `boost::asio::io_service` as almost all the classes in `Boost.Asio`. After we create a socket, it must be connected to some remote endpoint:

```
        c->socket.connect(boost::asio::ip::tcp::endpoint(
            boost::asio::ip::address_v4::from_string(addr),
            port_num
        ));
```

Take a look at the writing function. It accepts a unique pointer to the `connection_with_data` class and functor `f`:

```
#include <boost/asio/write.hpp>

template <class Functor>
void async_write_data(connection_ptr&& c, const Functor& f) {
```

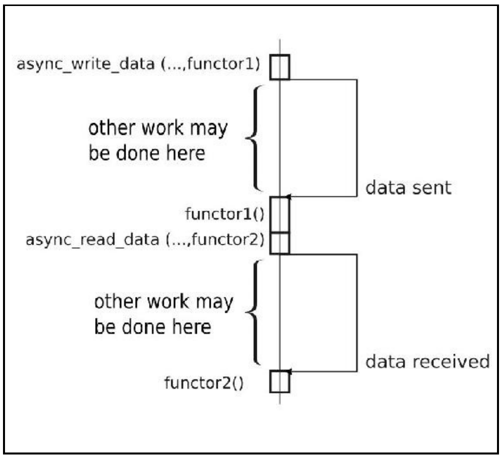
In it, we get references to socket and buffer:

```
        boost::asio::ip::tcp::socket& s = c->socket;
        std::string& d = c->data;
```

Then, we ask for asynchronous write:

```
    boost::asio::async_write(
        s,
        boost::asio::buffer(d),
        task_wrapped_with_connection<Functor>(std::move(c), f)
    );
}
```

All the interesting things happen in the `boost::asio::async_write` function. Just as in the case of timers, asynchronous call returns immediately without executing a function. It only tells to post the callback task to the `boost::asio::io_service` after some operation finishes (in our case, it's writing data to the socket). `boost::asio::io_service` executes our function in one of the threads that called the `io_service::run()` method. The following diagram illustrates this:


async_write_data (...,functor1)
other work may be done here
data sent
functor1()
async_read_data (...,functor2)
other work may be done here
data received
functor2()

Now, take a look at `task_wrapped_with_connection::operator()`. It accepts `const boost::system::error_code& error` and `std::size_t bytes_count`, because both `boost::asio::async_write` and `boost::asio::async_read` functions pass those parameters on async operation completion. A call to `c_->data.resize(bytes_count);` resizes the buffer to contain only the received/written data. Finally, we call the callback that was initially passed to an `async` function and stored as `task_unwrapped_`.

What was that all about? That was all about having a simple way for sending data! Now, we have an `async_write_data` function that asynchronously writes data from the buffer to the socket and executes a callback on operation completion:

```
void on_send(connection_ptr&& soc, const boost::system::
error_code& err);

void connect_and_send() {
    connection_ptr s = tasks_processor::create_connection
    ("127.0.0.1", 80);

    s->data = "data_to_send";
    async_write_data(
        std::move(s),
        &on_send
    );
}
```

`async_read_data` is pretty close to `async_write_data`. It resizes the buffer, creates a `task_wrapped_with_connection` function, and pushes it into `is_service` on async operation completion.

Note the `async_read_data_at_least` function. In its body, there's a slightly different call to `boost::asio::async_read`:

```
boost::asio::async_read(
    s,
    boost::asio::buffer(p, at_most),
    boost::asio::transfer_at_least(at_least_bytes),
    task_wrapped_with_connection<Functor>(std::move(c), f)
);
```

It has a `boost::asio::transfer_at_least(al_least_bytes)` in it. `Boost.Asio` has a lot of functors for customizing reads and writes. This one functor says, *transfer at least* `at_least_bytes` *bytes before calling the callback. More bytes are OK until they fit in buffer*.

Finally, let's take a look at one of the callbacks:

```
void process_server_response(
        connection_ptr&& soc,
        const boost::system::error_code& err);
```

In this example, callbacks must accept `connection_ptr` and a `boost::system::error_code` variable. A `boost::system::error_code` variable holds information about errors. It has an explicit conversion to `bool` operator, so the simple way to check for errors is just to write `if (err) { ... }`. If the remote ends transmission and closes the socket, `err` may contain `boost::asio::error::eof` error code. This is not always bad. In our example, we treat it as a non error behavior:

```
    if (err && err != boost::asio::error::eof) {
        std::cerr << "Client error on receive: "
        << err.message() << '\n';
        assert(false);
    }
```

Because we have tied together the socket and the buffer, you can get the received data from `soc->data`:

```
if (soc->data.size() != 2) {
    std::cerr << "Wrong bytes count\n";
    assert(false);
}

if (soc->data != "OK") {
    std::cerr << "Wrong response: " << soc->data << '\n';
    assert(false);
}
```

The `soc->shutdown()` call is optional, because when `soc` goes out of scope, the destructor for it is called. Destructor of `unique_ptr<connection_with_data>` calls `~connection_with_data` that has a `shutdown()` in its body.

There’s more…

Our `task_wrapped_with_connection::operator()` is not good enough! User provided `task_unwrapped_` callback my throw exceptions and may get interrupted by a `Boost.Thread` interruption that does not belong to that particular task. The fix would be to wrap the callback into the class from first recipe:

```
void operator()(
    const boost::system::error_code& error,
    std::size_t bytes_count)
{
    const auto lambda = [this, &error, bytes_count]() {
        this->c_->data.resize(bytes_count);
        this->task_unwrapped_(std::move(this->c_), error);
    };

    const auto task = detail::make_task_wrapped(lambda);
    task();
}
```

In `task_wrapped_with_connection::operator()`, we create a lambda function named `lambda`. On execution, `lambda` resizes the data inside the `connection_with_data` class to the `bytes_count` and calls an initially passed callback. Finally, we wrap the `lambda` into our safe for execution tasks from the first recipe and then execute it.

You may see a lot of `Boost.Asio` examples on the Internet. Many of those use `shared_ptr` instead of a `unique_ptr` for keeping the data around. Approach with `shared_ptr` is simpler to implement; however, it has two big drawbacks:

- Efficiency: `shared_ptr` has an atomic counter inside, and modifying it from different threads may significantly degrade performance. In one of the next recipes, you will see how to process tasks in multiple threads, and that’s the place where the differences may be noticeable in cases of high load.
- Explicitness: With `unique_ptr`, you always see that the ownership of the connection was transferred to somewhere (you see `std::move` in code). With `shared_ptr`, you can not understand from the interface whether the function grabs the ownership or if it just uses a reference to an object.

However, you may be forced to use `shared_ptr`, if according to the application’s logic, the ownership has to be shared across multiple tasks at the same time.

`Boost.Asio` is not a part of C++17, but it will be shipped as a Networking TS soon, and included into the one of the upcoming C++ standards.

See also

- See the official documentation of `Boost.Asio` for more examples, tutorials, full reference at http://boost.org/libs/asio, and an example of how to use the UDP or ICMP protocols.
- You may also read the *Boost.Asio C++ Network Programming* book, which describes `Boost.Asio` in more detail

Accepting incoming connections

A server-side working with a network often looks like a sequence where we first get the new connection, read data, then process it, and then send the result. Imagine that we are creating some kind of authorization server that must process huge amount of requests per second. In that case, we need to accept, receive, send asynchronously, and process tasks in multiple threads.

In this recipe, we'll see how to extend our `tasks_processor` class to accept and process incoming connections, and, in the next recipe, we'll see how to make it multithreaded.

Getting ready

This recipe requires a good knowledge of `boost::asio::io_service` basics as described in the first recipes of this chapter. Some knowledge about network communications will be of help to you. Knowledge of `boost::function` and information from at least two previous recipes is also required. Link this recipe with the `boost_system` and `boost_thread` libraries. Define `BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS` to bypass over restrictive library checks.

How to do it…

Just like in the previous recipes, we add new methods to our `tasks_processor` class.

1. We start with adding some `typedefs` to the `tasks_processor`:

```
class tasks_processor {
    typedef boost::asio::ip::tcp::acceptor acceptor_t;

    typedef boost::function<
        void(connection_ptr, const boost::system::error_code&)
    > on_accpet_func_t;
```

2. Let's add a class that ties together the socket for new incoming connections, socket to listen to, and a user provided callback for processing new connections:

```
private:
    struct tcp_listener {
        acceptor_t              acceptor_;
        const on_accpet_func_t  func_;
        connection_ptr          new_c_;

        template <class Functor>
        tcp_listener(
                boost::asio::io_service& io_service,
                unsigned short port,
                const Functor& task_unwrapped)
            : acceptor_(io_service, boost::asio::ip::tcp::endpoint(
                boost::asio::ip::tcp::v4(), port
            ))
            , func_(task_unwrapped)
        {}
    };

    typedef std::unique_ptr<tcp_listener> listener_ptr;
```

3. We need to add a function that starts listening on a specified port:

```
public:
   template <class Functor>
    static void add_listener(unsigned short port_num, const Functor& f) {
        std::unique_ptr<tcp_listener> listener(
            new tcp_listener(get_ios(), port_num, f)
        );

        start_accepting_connection(std::move(listener));
    }
```

4. Function that starts accepting incoming connections:

```
private:
   static void start_accepting_connection(listener_ptr&& listener) {
        if (!listener->acceptor_.is_open()) {
            return;
        }

        listener->new_c_.reset(new connection_with_data(
            listener->acceptor_.get_io_service()
        ));

        boost::asio::ip::tcp::socket& s = listener->new_c_->socket;
        acceptor_t& a = listener->acceptor_;
        a.async_accept(
            s,
            tasks_processor::handle_accept(std::move(listener))
        );
    }
```

5. We also need a functor that handles the new connection:

```
        private:
            struct handle_accept {
                listener_ptr listener;

                explicit handle_accept(listener_ptr&& l)
                    : listener(std::move(l))
                {}

                void operator()(const boost::system::error_code& error) {
                    task_wrapped_with_connection<on_accpet_func_t> task(
                        std::move(listener->new_c_), listener->func_
                    );

                    start_accepting_connection(std::move(listener));
                    task(error, 0);
                }
            };
```

Done! Now, we can accept connection in the following manner:

```
class authorizer {
public:
    static void on_connection_accpet(
        connection_ptr&& connection,
        const boost::system::error_code& error)
    {
        assert(!error);
        // ...
    }
};

int main() {
    tasks_processor::add_listener(80, &authorizer::on_connection_accpet);
    tasks_processor::start();
}
```

How it works…

The function `add_listener` constructs new `tcp_listener`, that keeps all the stuff required for accepting connections. Just as with any asynchronous operation, we need to keep resources alive while the operations executes. A unique pointer to `tcp_listener` does the job.

When we construct `boost::asio::ip::tcp::acceptor` specifying the endpoint (see *step 3*), it opens a socket at the specified address and gets ready for accepting connections.

In *step 4*, we create a new socket and call `async_accept` for that new socket. When a new connection comes, `listener->acceptor_` binds this connection to a socket and pushes the `tasks_processor::handle_accept` callback into `boost::asio::io_service`. As we understood from the previous recipe, all the `async_*` calls return immediately and `async_accept` is not a special case.

Let's take a closer look at our `handle_accept::operator()`. In it, we create a `task_wrapped_with_connection` functor from the previous recipe and move a new connection into it. Now, our `listener_ptr` does not have a socket in `new_c_`, as it is owned by the functor. We call the function `start_accepting_connection(std::move(listener))`, and it creates a new socket in `listener->new_c_` and starts an asynchronous accept. An async accept operation does not block, so the program continues execution, returns from the `start_accepting_connection(std::move(listener))` function, and executes the functor with the connection `task(error, 0)`.

> *You've made everything as shown in the example, but the performance of the server is not good enough. That's because the example is simplified and many optimizations left behind at the scene. The most significant one is to keep a separate small buffer in `connection_with_data` and use it for all the internal `Boost.Asio`'s callback related allocations. See Custom memory allocation example in the official documentation of the `Boost.Asio` library for more information on this optimization topic.*

When the destructor for the `boost::asio::io_service` is called, destructors for all the callbacks are called. This makes the destructor for `tcp_connection_ptr` to be called and frees the resources.

There’s more…

We did not use all the features of the `boost::asio::ip::tcp::acceptor` class. It can bind to a specific IPv6 or IPv4 address if we provide a specific `boost::asio::ip::tcp::endpoint`. You may also get a native socket via the `native_handle()` method and use some OS-specific calls for tuning the behavior. You may set up some options for `acceptor_` by calling `set_option`. For example, this is how you may force an `acceptor_` to reuse the address:

```
boost::asio::socket_base::reuse_address option(true);
acceptor_.set_option(option);
```

Reusing the address provides an ability to restart the server quickly after it was terminated without correct shutdown. After the server was terminated, a socket may be opened for some time, and you won’t be able to start the server on the same address without the `reuse_address` option.

C++17 has no classes from `Boost.Asio`, but Networking TS with most of the functionality is coming soon.

See also

- Starting this chapter from the beginning is a good idea to get much more information about `Boost.Asio`
- See the official documentation of `Boost.Asio` for more examples, tutorials, and a complete reference at http://boost.org/libs/asio

Executing different tasks in parallel

Now, it is time to make our `tasks_processor` process tasks in multiple threads. How hard can this be?

Getting started

You will need to read the first recipe from this chapter. Some knowledge of multithreading is also required, especially reading the *Manipulating a group of threads* recipe.

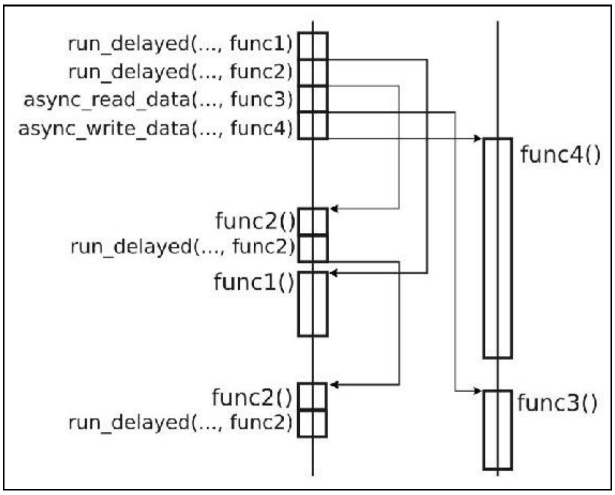
Link this recipe with the `boost_system` and `boost_thread` libraries. Define `BOOST_ASIO_DISABLE_HANDLER_TYPE_REQUIREMENTS` to bypass restrictive library checks.

How to do it…

All we need to do is to add the `start_multiple` method to our `tasks_processor` class:

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

class tasks_processor {
public:
    // Default value will attempt to guess optimal count of threads.
    static void start_multiple(std::size_t threads_count = 0) {
        if (!threads_count) {
            threads_count = (std::max)(static_cast<int>(
                boost::thread::hardware_concurrency()), 1
            );
        }

        // First thread is the current thread.
        -- threads_count;

        boost::asio::io_service& ios = get_ios();
        boost::thread_group tg;
        for (std::size_t i = 0; i < threads_count; ++i) {
            tg.create_thread([&ios]() { ios.run(); });
        }

        ios.run();
        tg.join_all();
    }
};
```

And now, we are able to do much more work as illustrated in the following diagram:



How it works…

The `boost::asio::io_service::run` method is thread safe. All we need to do is just run the `boost::asio::io_service::run` method from different threads.

If you are executing tasks that modify a common resource, you need to add mutexes around that resources, or organize your application in a way, that the common resource is not used simultaneously by different tasks. It is safe to use resource from different tasks without concurrent access to the resource because `boost::asio::io_service` takes care of additional synchronization between tasks and forces the modification results of one task to be seen by another task.

See the call to `boost::thread::hardware_concurrency()`. It returns the count of threads that can be run concurrently on current hardware. But, it is just a hint, and sometimes it may return a `0` value, which is why we are calling the `std::max` function for it. `std::max` ensures that `threads_count` stores at least the value `1`.

We wrapped `std::max` in parentheses because some popular compilers define the `min()` and `max()` macros, so we need additional tricks to work around this.

There's more…

The `boost::thread::hardware_concurrency()` function is a part of C++11; you may find it in the `<thread>` header of the `std::` namespace.

All the `boost::asio` classes are not part of C++17, but they will be available soon as a Networking TS.

See also

- See the `Boost.Asio` documentation for more examples and information about different classes at http://boost.org/libs/asio
- Recipes from Chapter 5, *Multithreading*, (especially the last recipe called *Manipulating a group of threads*) will give you information about the `Boost.Thread` usage
- See the `Boost.Thread` documentation for information about `boost::thread_group` and `boost::threads` at http://boost.org/libs/thread

Pipeline tasks processing

Sometimes, there is a requirement to process tasks in a specified time interval. Compared to previous recipes, where we were trying to process tasks in the order of their appearance in the queue, this is a big difference.

Consider the example where we are writing a program that connects two subsystems, one of which produces data packets and the other writes modified data to the disk (something like this can be seen in video cameras, sound recorders, and other devices). We need to process data packets one by one in the specified order, smoothly with a small jitter, and in multiple threads.

Naive approach does not work here:

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

subsystem1 subs1;
subsystem2 subs2;

void process_data() {
    while (!subs1.is_stopped()) {
        data_packet data = subs1.get_data();
        decoded_data d_decoded = decode_data(data);
        compressed_data c_data = compress_data(d_decoded);
        subs2.send_data(c_data);
    }
}

void run_in_multiple_threads() {
    boost::thread t(&process_data);
    process_data();

    t.join();
}
```

In a multithreaded environment, we can get *packet #1* in first thread and then *packet #2* in the second thread of execution. Because of different processing times, OS context switches and scheduling *packet #2* may be processed before *packet #1*. There’s no guarantee on packets, processing order. Let’s fix that!

Getting ready

The *Making a work_queue* recipe from Chapter 5, *Multithreading*, is required for understanding this example. The code must be linked against the `boost_thread` and `boost_system` libraries.

Basic knowledge of C++11, especially of lambda functions, is required.

How to do it…

This recipe is based on the code of the `work_queue` class from the *Making a work_queue* recipe of Chapter 5, *Multithreading*. We'll make some modifications and will be using a few instances of that class.

1. Let's start by creating separate queues for data decoding, data compressing, and data sending:

```
work_queue decoding_queue, compressing_queue, sending_queue;
```

2. Now, it is time to refactor the `process_data` and split it into multiple functions:

```
void start_data_accepting();
void do_decode(const data_packet& packet);
void do_compress(const decoded_data& packet);

void start_data_accepting() {
    while (!subs1.is_stopped()) {
        data_packet packet = subs1.get_data();

        decoding_queue.push_task(
            [packet]() {
                do_decode(packet);
            }
        );
    }
}

void do_decode(const data_packet& packet) {
    decoded_data d_decoded = decode_data(packet);

    compressing_queue.push_task(
        [d_decoded]() {
            do_compress(d_decoded);
        }
    );
}

void do_compress(const decoded_data& packet) {
    compressed_data c_data = compress_data(packet);

    sending_queue.push_task(
        [c_data]() {
            subs2.send_data(c_data);
        }
    );
}
```

3. Our `work_queue` class from Chapter 5, *Multithreading*, gets some interface changes for stopping and running tasks:

```
#include <deque>
#include <boost/function.hpp>
#include <boost/thread/mutex.hpp>
#include <boost/thread/locks.hpp>
#include <boost/thread/condition_variable.hpp>

class work_queue {
public:
    typedef boost::function<void()> task_type;

private:
    std::deque<task_type>       tasks_;
    boost::mutex                mutex_;
    boost::condition_variable   cond_;
    bool                        is_stopped_;

public:
    work_queue()
        : is_stopped_(false)
    {}

    void run();
    void stop();

    // Same as in Chapter 5, but with
    // rvalue references support.
    void push_task(task_type&& task);
};
```

4. The implementation of `work_queue`'s `stop()` and `run()` functions must look like this:

```
void work_queue::stop() {
    boost::lock_guard<boost::mutex> lock(mutex_);
    is_stopped_ = true;
    cond_.notify_all();
}

void work_queue::run() {
    while (1) {
        boost::unique_lock<boost::mutex> lock(mutex_);
        while (tasks_.empty()) {
            if (is_stopped_) {
                return;
            }
            cond_.wait(lock);
        }

        task_type t = std::move(tasks_.front());
        tasks_.pop_front();
        lock.unlock();

        t();
    }
}
```

5. That is all! Now, we only need to start the pipeline:

```
#include <boost/thread/thread.hpp>
int main() {
    boost::thread t_data_decoding(
        []() { decoding_queue.run(); }
    );
    boost::thread t_data_compressing(
        []() { compressing_queue.run(); }
    );
    boost::thread t_data_sending(
        []() { sending_queue.run(); }
    );

    start_data_accepting();
```

6. The pipeline can be stopped like this:

```
    decoding_queue.stop();
    t_data_decoding.join();

    compressing_queue.stop();
    t_data_compressing.join();

    sending_queue.stop();
    t_data_sending.join();
```

How it works…

The trick is to split the processing of a single data packet into some equally small subtasks and process them one by one in different `work_queues`. In this example, we can split the data process into data decoding, data compressing, and data sending.

Processing of six packets, ideally, would look like this:

Time	**Receiving**	**Decoding**	**Compressing**	**Sending**
Tick 1:	packet #1			
Tick 2:	packet #2	packet #1		
Tick 3:	packet #3	packet #2	packet #1	
Tick 4:	packet #4	packet #3	packet #2	packet #1
Tick 5:	packet #5	packet #4	packet #3	packet #2
Tick 6:	packet #6	packet #5	packet #4	packet #3
Tick 7:	-	packet #6	packet #5	packet #4
Tick 8:	-	-	packet #6	packet #5
Tick 9:	-	-	-	packet #6

However, our world is not ideal, so some tasks may finish faster than others. For example, receiving may work faster than decoding and, in that case, the decoding queue will be holding a set of tasks to be done. To avoid queue overflows, try hard to make each subsequent task slightly faster than the previous one.

We did not use `boost::asio::io_service` in our example, because it does not guarantee that posted tasks are executed in order of their postage.

There’s more…

All the tools used to create a pipeline in this example are available in C++11, so nothing would stop you from creating the same things without a Boost on a C++11 compatible compiler. However, Boost makes your code more portable and usable on the pre-C++11 compilers.

See also

- This technique is well known and used by processor developers. See http://en.wikipedia.org/wiki/Instruction_pipeline. Here, you may find a brief description of all the characteristics of the pipeline.
- *The Making a work_queue* from Chapter 5, *Multithreading* recipe will give you more information about methods used in this recipe.

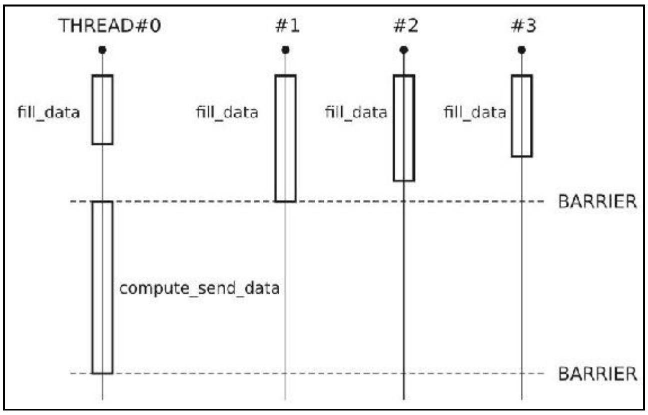
Making a nonblocking barrier

In multithreaded programming, there is an abstraction called **barrier**. It stops threads of execution that reach it until the requested number of threads are not blocked on it. After that, all the threads are released and they continue with their execution. Consider the following example of where it can be used.

We want to process different parts of data in different threads and then send the data:

```
#include <boost/array.hpp>
#include <boost/thread/barrier.hpp>
#include <boost/thread/thread.hpp>

typedef boost::array<std::size_t, 10000> vector_type;
typedef boost::array<vector_type, 4> data_t;

void fill_data(vector_type& data);
void compute_send_data(data_t& data);

void runner(std::size_t thread_index, boost::barrier& barrier, data_t& data) {
    for (std::size_t i = 0; i < 1000; ++ i) {
        fill_data(data.at(thread_index));
        barrier.wait();

        if (!thread_index) {
            compute_send_data(data);
        }
        barrier.wait();
    }
}

int main() {
    // Initing barrier.
    boost::barrier barrier(data_t::static_size);

    // Initing data.
    data_t data;

    // Run on 4 threads.
    boost::thread_group tg;
    for (std::size_t i = 0; i < data_t::static_size; ++i) {
        tg.create_thread([i, &barrier, &data] () {
            runner(i, barrier, data);
        });
    }

    tg.join_all();
}
```

The `data_barrier.wait()` method blocks until all the threads fill the data. After that, all the threads are released. The thread with the index `0` computes data to be sent using `compute_send_data(data)`, while other threads are again waiting at the barrier as shown in the following diagram:



Looks lame, doesn’t it?

Getting ready

This recipe requires knowledge of the first recipe of this chapter. Knowledge of `Boost.Thread` is also required. Code from this recipe requires linking against the `boost_thread` and `boost_system` libraries.

How to do it…

We do not need to block at all! Let’s take a closer look at the example. All we need to do is to post four `fill_data` tasks and make the last finished task call `compute_send_data(data)`.

1. We’ll need the `tasks_processor` class from the first recipe; no changes to it need to be done.
2. Instead of a barrier, we’ll be using the atomic variable:

```
#include <boost/atomic.hpp>
typedef boost::atomic<unsigned int> atomic_count_t;
```

3. Our new runner function will look like this:

```
void clever_runner(
        std::size_t thread_index,
        std::size_t iteration,
        atomic_count_t& counter,
        data_t& data)
{
    fill_data(data.at(thread_index));

    if (++counter != data_t::static_size) {
        return;
    }

    compute_send_data(data);

    if (++iteration == 1000) {
        // Exiting, because 1000 iterations are done.
        tasks_processor::stop();
        return;
    }

    counter = 0;
    for (std::size_t i = 0; i < data_t::static_size; ++ i) {
        tasks_processor::push_task([i, iteration, &counter, &data]() {
            clever_runner(
                i,
                iteration,
                counter,
                data
            );
        });
    }
}
```

4. The `main` function requires a minor change:

```
    // Initing counter.
    atomic_count_t counter(0);

    // Initing data.
    data_t data;

    // Run 4 tasks.
    for (std::size_t i = 0; i < data_t::static_size; ++i) {
        tasks_processor::push_task([i, &counter, &data]() {
            clever_runner(
                i,
                0, // first iteration
                counter,
                data
            );
        });
    }

    tasks_processor::start();
```

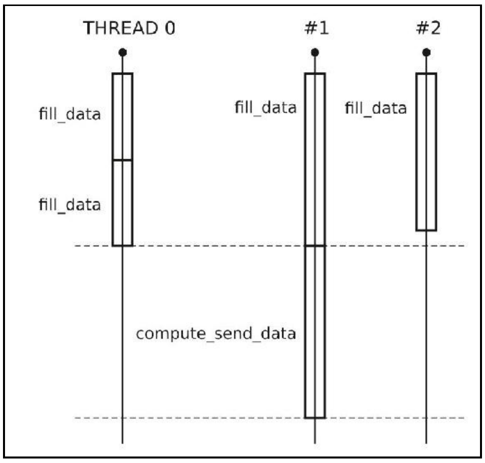
How it works...

We do not block at all. Instead of blocking, we count the tasks that finished filling the data. This is done by the `counter` atomic variable. The last remaining task will have a `counter` variable equal to `data_t::static_size`. Only that task must compute and send the data.

After that, we check for the exit condition (1000 iterations are done) and post the new data by pushing tasks to the queue.

There's more…

Is this a better solution? Well, first of all, it scales better:



This method can also be more effective for situations where a program does a lot of different work. Because no threads are waiting in barriers, free threads may execute some other tasks while one of the threads computes and sends the data.

This recipe could be implemented in C++11 without Boost libraries. You'll only need to replace `io_service` inside `tasks_processor` with `work_queue` from Chapter 5, *Multithreading*. But as always, Boost provides better portability, and it is possible to make this example run on a pre-C++11, compilers using Boost libraries. You'll only need to replace lambda functions with `boost::bind` and `boost::ref`.

See also

- Official documentation of `Boost.Asio` may give you more information about `io_service` usage at http://boost.org/libs/asio
- See all the `Boost.Function` related recipes from Chapter 2, *Managing Resources*, and the official documentation at http://boost.org/libs/function for an overview of how tasks work
- See the recipes from Chapter 1, *Starting to Write Your Application*, related to `Boost.Bind` for getting more information about what the `boost::bind` function does, or see the official documentation at http://boost.org/libs/bind

Storing an exception and making a task from it

Processing exceptions is not always trivial and may consume a lot of time. Consider the situation when an exception must be serialized and sent by the network. This may take milliseconds and a few thousands of lines of code. After the exception is caught, it is not always the best time and place to process it.

Can we store exceptions and delay their processing?

Getting ready

This recipe requires familiarity with `boost::asio::io_service`, which was described in the first recipe of this chapter.

This recipe requires linking with the `boost_system` and `boost_thread` libraries.

How to do it…

All we need is to have an ability to store exceptions and pass them between threads just like a usual variable.

1. Let’s start with the function that stores and processes exceptions:

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

struct process_exception {
    boost::exception_ptr exc_;

    explicit process_exception(const boost::exception_ptr& exc)
        : exc_(exc)
    {}

    void operator()() const;
};
```

2. The `operator()` of that functor just outputs the exception to the console:

```
#include <boost/lexical_cast.hpp>
void func_test2(); // Forward declaration.

void process_exception::operator()() const  {
    try {
        boost::rethrow_exception(exc_);
    } catch (const boost::bad_lexical_cast& /*e*/) {
        std::cout << "Lexical cast exception detected\n" << std::endl;

        // Pushing another task to execute.
        tasks_processor::push_task(&func_test2);
    } catch (...) {
        std::cout << "Can not handle such exceptions:\n"
            << boost::current_exception_diagnostic_information()
            << std::endl;

        // Stopping.
        tasks_processor::stop();
    }
}
```

3. Let’s write some functions to demonstrate how exceptions work:

```
#include <stdexcept>
void func_test1() {
    try {
        boost::lexical_cast<int>("oops!");
    } catch (...) {
        tasks_processor::push_task(
            process_exception(boost::current_exception())
        );
    }
}

void func_test2() {
    try {
        // ...
        BOOST_THROW_EXCEPTION(std::logic_error("Some fatal logic error"));
        // ...
    } catch (...) {
        tasks_processor::push_task(
            process_exception(boost::current_exception())
        );
    }
```

```
        }
```

Now, if we run the example like this:

```
  tasks_processor::get().push_task(&func_test1);
  tasks_processor::get().start();
```

We’ll get the following output:

```
Lexical cast exception detected

Can not handle such exceptions:
main.cpp(48): Throw in function void func_test2()
Dynamic exception type:
boost::exception_detail::clone_impl<boost::exception_detail::error_info_injector<std::logic_error>
 >
std::exception::what: Some fatal logic error
```

How it works…

The `Boost.Exception` library provides an ability to store and re-throw exceptions. The `boost::current_exception()` method must be called only from inside of the `catch()` block, and it returns an object of the type `boost::exception_ptr`.

In the preceding example in `func_test1()`, the `boost::bad_lexical_cast` exception is thrown. It is returned by `boost::current_exception()`; a `process_exception` task is created from that exception.

The only way to restore the exception type from `boost::exception_ptr` is to re-throw it using `boost::rethrow_exception(exc)` function. That's what the `process_exception` function does.

Throwing and catching exceptions is a heavy operation. Throwing may dynamically allocate memory, touch cold memory, lock mutex, compute a bunch of addresses, and do other stuff. Do not throw exception in performance critical paths without very good reasons to do so!

In `func_test2`, we are throwing a `std::logic_error` exception using the `BOOST_THROW_EXCEPTION` macro. This macro does a lot of useful work; it checks that our exception is derived from `std::exception`, adds information to our exception about the source filename, function name, and the code line number from where the exception was thrown. When our `std::logic_error` exception is re-thrown inside the `process_exception::operator()` it is caught by `catch(...)`. The `boost::current_exception_diagnostic_information()` outputs as much information about the thrown exception as possible.

There’s more…

Usually, `exception_ptr` is used to pass exceptions between threads. For example:

```
void run_throw(boost::exception_ptr& ptr) {
    try {
        // A lot of code goes here.
    } catch (...) {
        ptr = boost::current_exception();
    }
}

int main () {
    boost::exception_ptr ptr;

    // Do some work in parallel.
    boost::thread t(
        &run_throw,
        boost::ref(ptr)
    );

    // Some code goes here.
    // ...

    t.join();

    // Checking for exception.
    if (ptr) {
        // Exception occurred in thread.
        boost::rethrow_exception(ptr);
    }
}
```

The `boost::exception_ptr` class may allocate memory through heap multiple times, uses atomics, and implements some of the operations by re-throwing and catching exceptions. Try not to use it without an actual need.

C++11 has adopted `boost::current_exception`, `boost::rethrow_exception`, and `boost::exception_ptr`. You may find them in the `<exception>` in the `std::` namespace. The `BOOST_THROW_EXCEPTION` and `boost::current_exception_diagnostic_information()` functions are not in C++17.

See also

- Official documentation of `Boost.Exception` at http://boost.org/libs/exception contains a lot of useful information about implementation and restrictions. You may also find some information that is not covered in this recipe (for example, how to add additional information to an already thrown exception).
- The first recipe from this chapter gives you information about the `tasks_processor` class. The *Converting strings to numbers r*ecipe from Chapter 3, *Converting and Casting*, describes the `Boost.LexicalCast` library, that was used in this recipe.

Getting and processing system signals as tasks

When writing some server application (especially for Linux OS), catching and processing the signals is required. Usually, all the signal handlers are set up at server start and do not change during the application’s execution.

The goal of this recipe is to make our `tasks_processor` class capable of processing signals.

Getting ready

We will need code from the first recipe of this chapter. A sound knowledge of `Boost.Function` is also required.

This recipe requires linking with the `boost_system` and `boost_thread` libraries.

How to do it…

This recipe is similar to the recipes from *2* to *4* of this chapter: we have `async` signal waiting functions, some `async` signal handlers, and some support code.

1. Let's start by including the following headers:

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

2. Now, we add a member for signals processing to the `tasks_processor` class:

```
protected:
    static boost::asio::signal_set& signals() {
        static boost::asio::signal_set signals_(get_ios());
        return signals_;
    }

    static boost::function<void(int)>& signal_handler() {
        static boost::function<void(int)> users_signal_handler_;
        return users_signal_handler_;
    }
```

3. The function that will be called upon signal capture is as follows:

```
static void handle_signals(
        const boost::system::error_code& error,
        int signal_number)
{
    signals().async_wait(&tasks_processor::handle_signals);

    if (error) {
        std::cerr << "Error in signal handling: " << error << '\n';
    } else {
        boost::function<void(int)> h = signal_handler();
        h(signal_number);
    }

}
```

4. And now we need a function for registering the signals handler:

```
public:

    // This function is not thread safe!
    // Must be called before all the `start()` calls.
    // Function can be called only once.
    template <class Func>
    static void register_signals_handler(
            const Func& f,
            std::initializer_list<int> signals_to_wait)
    {
        // Making sure that this is the first call.
        assert(!signal_handler());

        signal_handler() = f;
        boost::asio::signal_set& sigs = signals();

        std::for_each(
            signals_to_wait.begin(),
            signals_to_wait.end(),
            [&sigs](int signal) { sigs.add(signal); }
        );
```

```
                sigs.async_wait(&tasks_processor::handle_signals);
            }
```

That’s all. Now, we are ready to process signals. The following is a test program:

```
void accept_3_signals_and_stop(int signal) {
    static int signals_count = 0;
    assert(signal == SIGINT);

    ++ signals_count;
    std::cout << "Captured " << signals_count << " SIGINT\n";
    if (signals_count == 3) {
        tasks_processor::stop();
    }
}


int main () {
    tasks_processor::register_signals_handler(
        &accept_3_signals_and_stop,
        { SIGINT, SIGSEGV }
    );

    tasks_processor::start();
}
```

This will give the following output:

```
Captured 1 SIGINT
Captured 2 SIGINT
Captured 3 SIGINT
Press any key to continue . . .
```

How it works…

Nothing is difficult here (compared to some previous recipes from this chapter). The `register_signals_handler` function adds the signal numbers that will be processed. It is done via a call to the `boost::asio::signal_set::add` function for each element of the `signals_to_wait`.

Next, the `sigs.async_wait` starts the `async` waiting for a signal and calls the `tasks_processor::handle_signals` function on the signal capture. The `tasks_processor::handle_signals` function immediately starts async wait for the next signal, checks for errors and, if there are none, it calls a callback providing a signal number.

There is more…

We can do better! We can wrap the user provided callback into our class from the first recipe to correctly process exceptions and do other good stuff from the first recipe:

```
boost::function<void(int)> h = signal_handler();

detail::make_task_wrapped([h, signal_number]() {
    h(signal_number);
})(); // make and run task_wrapped
```

When a thread-safe dynamic adding and removing of signals is required, we may modify this example to look like `detail::timer_task` from the *Making timers and processing timer events as tasks* recipe of this chapter . When multiple `boost::asio::signal_set` objects are registered for waiting on the same signals, a handler from each of `signal_set` is called on a single signal.

C++ has been capable of processing signals for a long time using the `signal` function from the `<csignal>` header. Networking TS probably will not have the `signal_set` functionality.

See also

- The *Storing any functional object in variable* recipe from Chapter 2, *Managing Resources*, provides information about `boost::function`
- See the official documentation of `Boost.Asio` for more information and examples on `boost::asio::signal_set` and other features of this great library at http://boost.org/libs/asio

Manipulating Strings

In this chapter, we will cover:

- Changing cases and case-insensitive comparison
- Matching strings using regular expressions
- Searching and replacing strings using regular expressions
- Formatting strings using safe printf-like functions
- Replacing and erasing strings
- Representing a string with two iterators
- Using a reference to string type

Introduction

This whole chapter is devoted to different aspects of changing, searching, and representing strings. We’ll see how some common string-related tasks can be easily done using Boost libraries. This chapter is easy enough; it addresses very common string manipulation tasks. So, let’s begin!

Changing cases and case-insensitive comparison

This is a pretty common task. We have two non-Unicode or ANSI character strings:

```
#include <string>
std::string str1 = "Thanks for reading me!";
std::string str2 = "Thanks for reading ME!";
```

We need to compare them in a case-insensitive manner. There are a lot of methods to do that, let’s take a look at Boost’s.

Getting ready

Basic knowledge of `std::string` is all we need here.

How to do it…

Here are different ways to do case-insensitive comparisons:

1. The most simple one is:

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

const bool solution_1 = (
     boost::iequals(str1, str2)
);
```

2. Using the Boost predicate and standard library method:

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

const bool solution_2 = (
    str1.size() == str2.size() && std::equal(
        str1.begin(),
        str1.end(),
        str2.begin(),
        boost::is_iequal()
    )
);
```

3. Making a lowercase copy of both the strings:

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

void solution_3() {
    std::string str1_low = boost::to_lower_copy(str1);
    std::string str2_low = boost::to_lower_copy(str2);
    assert(str1_low == str2_low);
}
```

4. Making an uppercase copy of the original strings:

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

void solution_4() {
    std::string str1_up = boost::to_upper_copy(str1);
    std::string str2_up = boost::to_upper_copy(str2);
    assert(str1_up == str2_up);
}
```

5. Converting the original strings to lowercase:

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

void solution_5() {
    boost::to_lower(str1);
    boost::to_lower(str2);
    assert(str1 == str2);
}
```

How it works…

The second method is not an obvious one. In the second method, we compare the length of strings. If they have the same length, we compare the strings character by character using an instance of the `boost::is_iequal` predicate. The `boost::is_iequal` predicate compares two characters in a case insensitive way.

The `Boost.StringAlgorithm` library uses `i` in the name of a method or class, if this method is case-insensitive. For example, `boost::is_iequal`, `boost::iequals`, `boost::is_iless`, and others.

There's more…

Each function and the functional object of the `Boost.StringAlgorithm` library that works with cases accept `std::locale`. By default (and in our examples), methods and classes use default constructed `std::locale`. If we work a lot with strings, it may be a good optimization to construct an `std::locale` variable once and pass it to all the methods. Another good optimization would be to use the *C* locale (if your application logic permits that) via `std::locale::classic()`:

```
// On some platforms std::locale::classic() works
// faster than std::locale().
boost::iequals(str1, str2, std::locale::classic());
```

Nobody forbids you to use both optimizations.

Unfortunately, C++17 has no string functions from `Boost.StringAlgorithm`. All the algorithms are fast and reliable, so do not be afraid to use them in your code.

See also

- Official documentation of the Boost String Algorithms Library can be found at http://boost.org/libs/algorithm/string
- See the *C++ Coding Standards* book by Andrei Alexandrescu and Herb Sutter for an example on how to make a case-insensitive string with a few lines of code

Matching strings using regular expressions

Let’s do something useful! It is a common case when the user’s input must be checked using some **regular expression**. The problem is that there are a lot of regular expression syntaxes, expressions written using one syntax are not treated well by the other syntaxes. Another problem is that long regexes are not so easy to write.

So in this recipe, we are going to write a program that supports different regular expression syntaxes and checks that the input strings match the specified regexes.

Getting started

This recipe requires basic knowledge of standard library. Knowledge of regular expression syntaxes can be helpful.

Linking examples against the `boost_regex` library is required.

How to do it…

This regex-matcher example consists of a few lines of code in the `main()` function:

1. To implement it, we need the following headers:

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

2. At the start of the program, we need to output the available regex syntaxes:

```
int main() {
    std::cout
        << "Available regex syntaxes:\n"
        << "\t[0] Perl\n"
        << "\t[1] Perl case insensitive\n"
        << "\t[2] POSIX extended\n"
        << "\t[3] POSIX extended case insensitive\n"
        << "\t[4] POSIX basic\n"
        << "\t[5] POSIX basic case insensitive\n\n"
        << "Choose regex syntax: ";
```

3. Now, correctly set up flags, according to the chosen syntax:

```
    boost::regex::flag_type flag;
    switch (std::cin.get())
    {
    case '0': flag = boost::regex::perl;
        break;

    case '1': flag = boost::regex::perl|boost::regex::icase;
        break;

    case '2': flag = boost::regex::extended;
        break;

    case '3': flag = boost::regex::extended|boost::regex::icase;
        break;

    case '4': flag = boost::regex::basic;
        break;

    case '5': flag = boost::regex::basic|boost::regex::icase;
        break;
    default:
        std::cout << "Incorrect number of regex syntax. Exiting...\n";
        return 1;
    }

    // Disabling exceptions.
    flag |= boost::regex::no_except;
```

4. We are now requesting regex patterns in a loop:

```
    // Restoring std::cin.
    std::cin.ignore();
    std::cin.clear();

    std::string regex, str;
    do {
        std::cout << "Input regex: ";
        if (!std::getline(std::cin, regex) || regex.empty()) {
            return 0;
        }
        // Without `boost::regex::no_except`flag this
        // constructor may throw.
        const boost::regex e(regex, flag);
        if (e.status()) {
            std::cout << "Incorrect regex pattern!\n";
            continue;
        }
```

5. Getting a `String to match:` in a loop:

```
        std::cout << "String to match: ";
        while (std::getline(std::cin, str) && !str.empty()) {
```

6. Applying regex to it and outputting the result:

```
            const bool matched = boost::regex_match(str, e);
            std::cout << (matched ? "MATCH\n" : "DOES NOT MATCH\n");
            std::cout << "String to match: ";
        } // end of `while (std::getline(std::cin, str))`
```

7. We will finish our example by restoring `std::cin` and requesting new regex patterns:

```
        // Restoring std::cin.
        std::cin.ignore();
        std::cin.clear();
    } while (1);
} // int main()
```

Now if we run the preceding example, we'll get the following output:

```
Available regex syntaxes:

[0] Perl
[1] Perl case insensitive
[2] POSIX extended
[3] POSIX extended case insensitive
[4] POSIX basic
[5] POSIX basic case insensitive
```

```
Choose regex syntax: 0
 Input regex: (\d{3}[#-]){2}
 String to match: 123-123#
 MATCH
 String to match: 312-321-
 MATCH
 String to match: 21-123-
 DOES NOT MATCH
 String to match: ^Z
 Input regex: \l{3,5}
 String to match: qwe
 MATCH
 String to match: qwert
 MATCH
 String to match: qwerty
 DOES NOT MATCH
 String to match: QWE
 DOES NOT MATCH
 String to match: ^Z

 Input regex: ^Z
 Press any key to continue . . .
```

How it works…

All the matching is done by the `boost::regex` class. It constructs an object that is capable of regex parsing and compilation. Additional configuration options are passed to the class using a `flag` input variable.

If the regular expression is incorrect, `boost::regex` throws an exception. If the `boost::regex::no_except` flag was passed, it reports an error returning a non-zero in the `status()` call (just like in our example):

```
if (e.status()) {
    std::cout << "Incorrect regex pattern!\n";
    continue;
}
```

This will result in:

```
Input regex: (incorrect regex(
Incorrect regex pattern!
```

Regular expressions matching is done by a call to the `boost::regex_match` function. It returns `true` in case of a successful match. Additional flags may be passed to `regex_match`, but we avoided their usage for brevity of the example.

There’s more…

C++11 contains almost all the `Boost.Regex` classes and flags. They can be found in the `<regex>` header of the `std::` namespace (instead of `boost::`). Official documentation provides information about differences of C++11 and `Boost.Regex`. It also contains some performance measures that tell that `Boost.Regex` is fast. Some standard libraries have performance issues, so choose wisely between Boost and Standard library versions.

See also

- The *Searching and replacing strings by regular expressions* recipe will give you more information about the `Boost.Regex` usage
- You may also consider official documentation to get more information about flags, performance measures, regular expression syntaxes, and C++11 conformance at http://boost.org/libs/regex

Searching and replacing strings using regular expressions

My wife enjoyed the *Matching strings by regular expressions* recipe very much. But, she wanted more and told me that I'll get no food until I promote the recipe to be able to replace parts of the input string according to a regex match.

Ok, here it comes. Each matched sub-expression (part of the regex in parenthesis) must get a unique number starting from 1; this number would be used to create a new string.

This is how an updated program should work like:

```
Available regex syntaxes:

        [0] Perl
        [1] Perl case insensitive
        [2] POSIX extended
        [3] POSIX extended case insensitive
        [4] POSIX basic
        [5] POSIX basic case insensitive


Choose regex syntax: 0
Input regex: (\d)(\d)
String to match: 00
MATCH: 0, 0,
Replace pattern: \1#\2
RESULT: 0#0
String to match: 42
MATCH: 4, 2,
Replace pattern: ###\1-\1-\2-\1-\1###
RESULT: ###4-4-2-4-4###
```

Getting ready

We’ll be reusing the code from the *Matching strings by regular expressions* recipe. It is recommended to read it before getting your hands on this one.

Linking an example against the `boost_regex` library is required.

How to do it…

This recipe is based on the code from the previous one. Let’s see what must be changed:

1. No additional headers should be included. However, we need an additional string to store the replace pattern:

```
std::string regex, str, replace_string;
```

2. We replace `boost::regex_match` with `boost::regex_find` and output matched results:

```
std::cout << "String to match: ";
while (std::getline(std::cin, str) && !str.empty()) {
    boost::smatch results;
    const bool matched = regex_search(str, results, e);
    if (matched)  {
        std::cout << "MATCH: ";
        std::copy(
            results.begin() + 1,
            results.end(),
            std::ostream_iterator<std::string>(std::cout, ", ")
        );
```

3. After that, we need to get the replace pattern and apply it:

```
        std::cout << "\nReplace pattern: ";
        if (
                std::getline(std::cin, replace_string)
                && !replace_string.empty())
        {
            std::cout << "RESULT: " <<
                boost::regex_replace(str, e, replace_string)
            ;
        } else {
            // Restoring std::cin.
            std::cin.ignore();
            std::cin.clear();
        }
    } else { // `if (matched) `
        std::cout << "DOES NOT MATCH";
    }
```

That’s it! Everyone’s happy and I’m fed.

How it works…

The `boost::regex_search` function doesn't only return a `true` or a `false` value (unlike the `boost::regex_match` function does), but also stores matched parts. We output matched parts using the following construction:

```
std::copy(
    results.begin() + 1,
    results.end(),
    std::ostream_iterator<std::string>( std::cout, ", ")
);
```

Note that we outputted the results by skipping the first result (`results.begin() + 1`), that is because `results.begin()` contains the whole regex match.

The `boost::regex_replace` function does all the replacing and returns the modified string.

There's more…

There are different variants of the `regex_*` functions, some of them receive bidirectional iterators instead of strings and some provide output to the iterator.

`boost::smatch` is a `typedef` for `boost::match_results<std::string::const_iterator>`. If you are using some other bidirectional iterators instead of `std::string::const_iterator`, you shall use the type of your bidirectional iterators as a template parameter for `boost::match_results`.

`match_results` has a format function, so we may tune our example with it, instead of:

```
std::cout << "RESULT: " << boost::regex_replace(str, e, replace_string);
```

We may use the following:

```
std::cout << "RESULT: " << results.format(replace_string);
```

By the way, `replace_string` supports multiple formats:

```
        Input regex: (\d)(\d)
         String to match: 12
         MATCH: 1, 2,
         Replace pattern: $1-$2---$&---$$
         RESULT: 1-2---12---$
```

All the classes and functions from this recipe exist in C++11 in the `std::` namespace of the `<regex>` header.

See also

Official documentation of `Boost.Regex` will give you more examples and information about performance, C++11 standard compatibility, and regular expression syntax at http://boost.org/libs/regex. The *Matching strings by regular expressions* recipe will tell you the basics of `Boost.Regex`.

Formatting strings using safe printf-like functions

The `printf` family of functions is a threat to security. It is a very bad design to allow users to put their own strings as a type and format the specifiers. So what do we do when user-defined format is required? How shall we implement the `std::string to_string(const std::string& format_specifier) const;` member function of the following class?

```
class i_hold_some_internals
{
    int i;
    std::string s;
    char c;
    // ...
};
```

Getting ready

Basic knowledge of standard library is more than enough for this recipe.

How to do it…

We wish to allow users to specify their own output format for a string:

1. To do that in a safe manner, we need the following header:

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

2. Now, we add some comments for the user:

```
            // `fmt` parameter may contain the following:
            // $1$ for outputting integer 'i'.
            // $2$ for outputting string 's'.
            // $3$ for outputting character 'c'.
            std::string to_string(const std::string& fmt) const {
```

3. It is time to make all the parts work:

```
                boost::format f(fmt);
                unsigned char flags = boost::io::all_error_bits;
                flags ^= boost::io::too_many_args_bit;
                f.exceptions(flags);
                return (f % i % s % c).str();
            }
```

That is all. Take a look at this code:

```
int main() {
    i_hold_some_internals class_instance;

    std::cout << class_instance.to_string(
        "Hello, dear %2%! "
        "Did you read the book for %1% %% %3%\n"
    );

    std::cout << class_instance.to_string(
        "%1% == %1% && %1%%% != %1%\n\n"
    );
}
```

Imagine that `class_instance` has a member `i` equal to `100`, `s` member equal to `"Reader"`, and a member `c` equal to `'!'`. Then, the program will output the following:

```
 Hello, dear Reader! Did you read the book for 100 % !
 100 == 100 && 100% != 100
```

How it works…

The `boost::format` class accepts the string that specifies the resulting string format. Arguments are passed to `boost::format` using `operator%`. Values `%1%`, `%2%`, `%3%`, `%4%`, and so on, in the format specifying string are replaced by arguments passed to `boost::format`.

We also disable the exceptions for cases when a format string contains fewer arguments than passed to `boost::format`:

```
boost::format f(format_specifier);
unsigned char flags = boost::io::all_error_bits;
flags ^= boost::io::too_many_args_bit;
```

This is done to allow some formats like this:

```
// Outputs 'Reader'.
std::cout << class_instance.to_string("%2%\n\n");
```

There's more…

What happens in case of an incorrect format?

Nothing awful, an exception is thrown:

```
    try {
        class_instance.to_string("%1% %2% %3% %4% %5%\n");
        assert(false);
    } catch (const std::exception& e) {
        // boost::io::too_few_args exception is catched.
        std::cout << e.what() << '\n';
    }
```

The following lines are outputted to the console by the previous code snippet:

```
boost::too_few_args: format-string referred to more arguments than
   were passed
```

C++17 has no `std::format`. The `Boost.Format` library is not a very fast library. Try not to use it in performance critical sections a lot.

See also

Official documentation contains more information about performance of the `Boost.Format` library. More examples and documentation on extended printf-like format is available at http://boost.org/libs/format.

Replacing and erasing strings

Situations where we need to erase something in a string, replace a part of the string, or erase the first or last occurrence of some sub-string are very common. Standard library allows us to do more parts of this, but it usually involves too much code writing.

We saw the `Boost.StringAlgorithm` library in action in the *Changing cases and case-insensitive comparison* recipe. Let’s see how it can be used to simplify our lives when we need to modify some strings:

```
#include <string>
const std::string str = "Hello, hello, dear Reader.";
```

Getting ready

Basic knowledge of C++ is required for this example.

How to do it…

This recipe shows how different string-erasing and replacing methods from the `Boost.StringAlgorithm` library work:

1. Erasing requires the `#include <boost/algorithm/string/erase.hpp>` header:

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

   void erasing_examples() {
       namespace ba = boost::algorithm;
       using std::cout;

       cout << "\n erase_all_copy :" << ba::erase_all_copy(str, ",");
       cout << "\n erase_first_copy:" << ba::erase_first_copy(str, ",");
       cout << "\n erase_last_copy :" << ba::erase_last_copy(str, ",");
       cout << "\n ierase_all_copy :" << ba::ierase_all_copy(str, "hello");
       cout << "\n ierase_nth_copy :" << ba::ierase_nth_copy(str, ",", 1);
   }
   ```

   This code outputs the following:

   ```
        erase_all_copy   :Hello hello dear Reader.
        erase_first_copy :Hello hello, dear Reader.
        erase_last_copy  :Hello, hello dear Reader.
        ierase_all_copy   :, , dear Reader.
        ierase_nth_copy  :Hello, hello dear Reader.
   ```

2. Replacing requires the `<boost/algorithm/string/replace.hpp>` header:

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

   void replacing_examples() {
       namespace ba = boost::algorithm;
       using std::cout;

       cout << "\n replace_all_copy :"
           << ba::replace_all_copy(str, ",", "!");

       cout << "\n replace_first_copy :"
           << ba::replace_first_copy(str, ",", "!");

       cout << "\n replace_head_copy :"
           << ba::replace_head_copy(str, 6, "Whaaaaaaa!");
   }
   ```

   This code outputs the following:

   ```
        replace_all_copy :Hello! hello! dear Reader.
        replace_first_copy :Hello! hello, dear Reader.
        replace_head_copy :Whaaaaaaa! hello, dear Reader.
   ```

How it works…

All the examples are self-documenting. The only one that is not obvious is the `replace_head_copy` function. It accepts count of bytes to replace as a second parameter and a replace string as the third parameter. So, in the preceding example, `Hello` gets replaced with `Whaaaaaaa!`.

There’s more…

There are also methods that modify strings in-place. They just do not end on `_copy` and return `void`. All the case-insensitive methods (the ones that start with `i`) accept `std::locale` as the last parameter and use a default constructed locale as a default parameter.

Do you use case insensitive methods a lot and need a better performance? Just create a `std::locale` variable holding `std::locale::classic()` and pass it to all the algorithms. On small strings most of the time is eaten by `std::locale` constructions, not by the algorithms:

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

void erasing_examples_locale() {
    namespace ba = boost::algorithm;

    const std::locale loc = std::locale::classic();

    const std::string r1
        = ba::ierase_all_copy(str, "hello", loc);

    const std::string r2
        = ba::ierase_nth_copy(str, ",", 1, loc);

    // ...
}
```

C++17 does not have `Boost.StringAlgorithm` methods and classes. However, it has a `std::string_view` class that can use substrings without memory allocations. You can find out more about `std::string_view` like classes in the next two recipes of this chapter.

See also

- Official documentation contains a lot of examples and a full reference of all the methods at http://boost.org/libs/algorithm/string
- See the *Changing cases and case-insensitive comparison* recipe from this chapter for more information about the `Boost.StringAlgorithm` library

Representing a string with two iterators

There are situations when we need to split some strings into substrings and do something with those substrings. In this recipe, we want to split string into sentences, count characters, and white-spaces and, of course, we want to use Boost and be as efficient as possible.

Getting ready

You’ll need some basic knowledge of standard library’s algorithms for this recipe.

How to do it…

That’s very easy to do with Boost:

1. First of all, include the right headers:

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

2. Now, let’s define our test string:

```
int main() {
    const char str[] =
        "This is a long long character array."
        "Please split this character array to sentences!"
        "Do you know, that sentences are separated using period, "
        "exclamation mark and question mark? :-)"
    ;
```

3. We make a `typedef` for our splitting iterator:

```
    typedef boost::split_iterator<const char*> split_iter_t;
```

4. Construct that iterator:

```
    split_iter_t sentences = boost::make_split_iterator(str,
        boost::algorithm::token_finder(boost::is_any_of("?!."))
    );
```

5. Now, we can iterate between matches:

```
    for (unsigned int i = 1; !sentences.eof(); ++sentences, ++i) {
        boost::iterator_range<const char*> range = *sentences;
        std::cout << "Sentence #" << i << " : \t" << range << '\n';
```

6. Count the number of characters:

```
        std::cout << range.size() << " characters.\n";
```

7. And count the white-spaces:

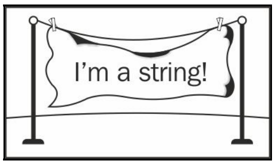
```
        std::cout
            << "Sentence has "
            << std::count(range.begin(), range.end(), ' ')
            << " whitespaces.\n\n";
    } // end of for(...) loop
} // end of main()
```

That’s it. Now if we run an example, it will output:

```
Sentence #1 : This is a long long character array
35 characters.
Sentence has 6 whitespaces.

Sentence #2 : Please split this character array to sentences
46 characters.
Sentence has 6 whitespaces.
Sentence #3 : Do you know, that sentences are separated using dot,
exclamation mark and question mark
90 characters.
Sentence has 13 whitespaces.

Sentence #4 : :-)
4 characters.
Sentence has 1 whitespaces.
```

How it works…

The main idea of this recipe is that we do not need to construct `std::string` from substrings. We even do not need to tokenize the whole string at once. All we need to do is find the first substring and return it as a pair of iterators to the beginning and to the end of the substring. If we need more substrings, find the next substring and return a pair of iterators for that substring.



Now, let's take a closer look at `boost::split_iterator`. We constructed one using the `boost::make_split_iterator` function that takes `range` as the first argument and a binary finder predicate (or binary predicate) as the second. When `split_iterator` is dereferenced, it returns the first substring as `boost::iterator_range<const char*>`, which just holds a pair of pointers and has a few methods to work with them. When we increment `split_iterator`, it tries to find the next substring, and if there is no substring found, `split_iterator::eof()` returns `true`.

Default constructed split iterator represents an `eof()`. So we could rewrite the loop condition from `!sentences.eof()` to `sentences != split_iter_t()`. You could also use the split iterators with algorithms, for example:

`std::for_each(sentences, split_iter_t(), [](auto range){ /**/ });`.

There’s more…

The `boost::iterator_range` class is widely used across all the Boost libraries. You may find it useful even for your own code in situations when a pair of iterators must be returned or when a function shall accept/work with a pair of iterators.

The `boost::split_iterator<>` and `boost::iterator_range<>` classes accept a forward iterator type as a template parameter. Because we were working with the character array in the preceding example, we provided `const char*` as iterators. If we were working with `std::wstring`, we would need to use the `boost::split_iterator<std::wstring::const_iterator>` and `boost::iterator_range<std::wstring::const_iterator>` types.

C++17 has neither `iterator_range` nor `split_iterator`. However, there are discussions going on for accepting `iterator_range` like class that would probably have the name `std::span`.

The `boost::iterator_range` class has no virtual functions and no dynamic memory allocations, it is fast and efficient. However, its output stream operator `<<` has no specific optimizations for character arrays, so streaming could be slow.

The `boost::split_iterator` class has a `boost::function` class in it, so constructing it for big functors could be slow. Iterating adds only tiny overheads that you won’t feel even in performance critical sections.

See also

- The next recipe will tell you about a nice replacement for `boost::iterator_range<const char*>`
- Official documentation for `Boost.StringAlgorithm` may provide you with more detailed information about classes and a whole bunch of examples at http://boost.org/libs/algorithm/string
- More information about `boost::iterator_range` can be found here: http://boost.org/libs/range; it is a part of the `Boost.Range` library that is not described in this book, but you may wish to study it by yourself

Using a reference to string type

This recipe is the most important recipe in this chapter! Let's take a look at a very common case, where we write some function that accepts a string and returns a part of the string between character values passed in the `starts` and `ends` arguments:

```
#include <string>
#include <algorithm>

std::string between_str(const std::string& input, char starts, char ends) {
    std::string::const_iterator pos_beg
        = std::find(input.begin(), input.end(), starts);
    if (pos_beg == input.end()) {
        return std::string();
    }
    ++ pos_beg;

    std::string::const_iterator pos_end
        = std::find(pos_beg, input.end(), ends);

    return std::string(pos_beg, pos_end);
}
```

Do you like this implementation? In my opinion, it is awful. Consider the following call to it:

```
between_str("Getting expression (between brackets)", '(', ')');
```

In this example, a temporary `std::string` variable is constructed from `"Getting expression (between brackets)"`. The character array is long enough, so there is a big chance that dynamic memory allocation is called inside the `std::string` constructor and the character array is copied into it. Then, somewhere inside the `between_str` function, new `std::string` is being constructed, which may also lead to another dynamic memory allocation and copying.

So, this simple function may, and in most cases will:

- Call dynamic memory allocation (two times)
- Copy string (two times)
- Deallocate memory (two times)

Can we do better?

Getting ready

This recipe requires basic knowledge of standard library and C++.

How to do it…

We do not really need a `std::string` class here, we only need some lightweight class that does not manage resources and only has a pointer to the character array and array's size. Boost has the `boost::string_view` class for that.

1. To use the `boost::string_view` class, include the following header:

```
    #include <boost/utility/string_view.hpp>
```

2. Change the method's signature:

```
    boost::string_view between(
        boost::string_view input,
        char starts,
        char ends)
```

3. Change `std::string` to `boost::string_view:` everywhere inside the function body:

```
    {
        boost::string_view::const_iterator pos_beg
            = std::find(input.cbegin(), input.cend(), starts);
        if (pos_beg == input.cend()) {
            return boost::string_view();
        }
        ++ pos_beg;

        boost::string_view::const_iterator pos_end
            = std::find(pos_beg, input.cend(), ends);
        // ...
```

4. The `boost::string_view` constructor accepts size as a second parameter, so we need to slightly change the code:

```
        if (pos_end == input.cend()) {
            return boost::string_view(pos_beg, input.end() - pos_beg);
        }

        return boost::string_view(pos_beg, pos_end - pos_beg);
    }
```

That's it! Now we may call `between("Getting expression (between brackets)", '(', ')')` and it will work without any dynamic memory allocation and characters copying. And we can still use it for `std::string`:

```
between(std::string("(expression)"), '(', ')')
```

How it works…

As already mentioned, `boost::string_view` contains only a pointer to the character array and size of data. It has a lot of constructors and may be initialized in different ways:

```
    boost::string_view r0("^_^");

    std::string O_O("O__O");
    boost::string_view r1 = O_O;

    std::vector<char> chars_vec(10, '#');
    boost::string_view r2(&chars_vec.front(), chars_vec.size());
```

The `boost::string_view` class has all the methods required by the `container` class, so it is usable with standard library algorithms and Boost algorithms:

```
#include <boost/algorithm/string/case_conv.hpp>
#include <boost/algorithm/string/replace.hpp>
#include <boost/lexical_cast.hpp>
#include <iterator>
#include <iostream>

void string_view_algorithms_examples() {
    boost::string_view r("O_O");
    // Finding single symbol.
    std::find(r.cbegin(), r.cend(), '_');

    // Will print 'o_o'.
    boost::to_lower_copy(std::ostream_iterator<char>(std::cout), r);
    std::cout << '\n';

    // Will print 'O_O'.
    std::cout << r << '\n';

    // Will print '^_^'.
    boost::replace_all_copy(
        std::ostream_iterator<char>(std::cout), r, "O", "^"
    );
    std::cout << '\n';

    r = "100";
    assert(boost::lexical_cast<int>(r) == 100);
}
```

The `boost::string_view` class does not really own string, so all its methods return constant iterators. Because of that, we cannot use it in methods that modify data, such as `boost::to_lower(r)`.

While working with `boost::string_view`, we must take additional care about data that it refers to; it must exist and be valid for the whole lifetime of the `boost::string_view` variable that references it.

Before Boost 1.61 there was no `boost::string_view` class, but the `boost::string_ref` class was used instead. Those classes are really close. `boost::string_view` closer follows the C++17 design and has better constexpr support. Since Boost 1.61, `boost::string_ref` is deprecated.

The `string_view` classes are fast and efficient, because they never allocate memory and have no virtual functions! Use them wherever it is possible. They are designed to be a drop-in replacement for `const std::string&` and `const char*` parameters. It means that you can replace the following three functions:

```
void foo(const std::string& s);
void foo(const char* s);
void foo(const char* s, std::size_t s_size);
```

With a single one:

```
void foo(boost::string_view s);
```

There’s more…

The `boost::string_view` class is a C++17 class. You can find it in the `<string_view>` header in the `std::` namespace if your compiler is C++17 compatible.

Boost’s and standard library’s version support constexpr usage of `string_view`s; however, `std::string_view` currently has more functions marked with constexpr.

Note that we have accepted the `string_view` variable by value instead of a constant reference. That’s the recommended way for passing `boost::string_view`s and `std::string_view`s because:

- `string_view` is a small class with trivial types inside. Passing it by value usually results in better performance because of less indirections and it allows the compiler to do more optimizations.
- In other cases, when there’s no performance difference writing `string_view val` is shorter than writing `const string_view& val`.

Just like the C++17’s `std::string_view`, the `boost::string_view` class is actually a `typedef`:

```
typedef basic_string_view<char, std::char_traits<char> > string_view;
```

You may also find useful the following typedefs for wide characters in the `boost::` and `std::` namespaces:

```
typedef basic_string_view<wchar_t,  std::char_traits<wchar_t> > wstring_view;

typedef basic_string_view<char16_t, std::char_traits<char16_t> > u16string_view;

typedef basic_string_view<char32_t, std::char_traits<char32_t> > u32string_view;
```

See also

Boost documentation for `string_ref` and `string_view` can be found at http://boost.org/libs/utility.

Metaprogramming

In this chapter, we will cover:

- Using type vector of types
- Manipulating vector of types
- Getting a function’s result type at compile time
- Making a higher-order metafunction
- Evaluating metafunctions lazily
- Converting all the tuple elements to string
- Splitting tuples
- Manipulating heterogeneous containers in C++14

Introduction

This chapter is devoted to some cool and hard to understand metaprogramming methods. These methods are not for everyday use, but they may be of real help for the development of generic libraries.

Chapter 4, *Compile-time Tricks*, already covered the basics of metaprogramming. Reading it is recommended for better understanding. In this chapter, we'll go deeper and see how multiple types can be packed in a single tuple like type. We'll make functions for manipulating collections of types, we'll see how types of compile-time collections may be changed, and how compile-time tricks can be mixed with runtime. All this is metaprogramming.

Fasten your seat belts and get ready, lets go…!

Using type vector of types

There are situations when it would be great to work with all the template parameters like they were in a container. Imagine that we are writing something, such as `Boost.Variant`:

```
#include <boost/mpl/aux_/na.hpp>

// boost::mpl::na == n.a. == not available
template <
    class T0 = boost::mpl::na,
    class T1 = boost::mpl::na,
    class T2 = boost::mpl::na,
    class T3 = boost::mpl::na,
    class T4 = boost::mpl::na,
    class T5 = boost::mpl::na,
    class T6 = boost::mpl::na,
    class T7 = boost::mpl::na,
    class T8 = boost::mpl::na,
    class T9 = boost::mpl::na
    >
struct variant;
```

The preceding code is the place where all the following interesting tasks start to happen:

- How can we remove constant and volatile qualifiers from all the types?
- How can we remove duplicate types?
- How can we get the sizes of all the types?
- How can we get the maximum size of the input parameters?

All these tasks can be easily solved using `Boost.MPL`.

Getting ready

A basic knowledge of Chapter 4, *Compile-time Tricks*, is required for this recipe. Accumulate some courage before reading—there will be a lot of metaprogramming in this recipe.

How to do it…

We already saw how a type can be manipulated at compile time. Why can't we go further and combine multiple types in an array and perform operations for each element of that array?

1. First of all, let's pack all the types in one of the `Boost.MPL` type's container:

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

template <
    class T0, class T1, class T2, class T3, class T4,
    class T5, class T6, class T7, class T8, class T9
>
struct variant {
    typedef boost::mpl::vector<
        T0, T1, T2, T3, T4, T5, T6, T7, T8, T9
    > types;
};
```

2. Let's make our example less abstract and see how it works if we specify types:

```
#include <string>
struct declared{ unsigned char data[4096]; };
struct non_declared;

typedef variant<
    volatile int,
    const int,
    const long,
    declared,
    non_declared,
    std::string
>::types types;
```

3. We may check everything at compile time. Let's assert that types is not empty:

```
#include <boost/static_assert.hpp>
#include <boost/mpl/empty.hpp>

BOOST_STATIC_ASSERT((!boost::mpl::empty<types>::value));
```

4. We may also check that, for example, the `non_declared` types is still at index `4` position:

```
#include <boost/mpl/at.hpp>
#include <boost/type_traits/is_same.hpp>

BOOST_STATIC_ASSERT((boost::is_same<
    non_declared,
    boost::mpl::at_c<types, 4>::type
>::value));
```

5. And that the last type is still `std::string`:

```
#include <boost/mpl/back.hpp>

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::back<types>::type,
    std::string
>::value));
```

6. We may carry out some transformations. Let's start with removing constant and volatile qualifiers:

```
#include <boost/mpl/transform.hpp>
#include <boost/type_traits/remove_cv.hpp>

typedef boost::mpl::transform<
    types,
    boost::remove_cv<boost::mpl::_1>
>::type noncv_types;
```

7. Here's how we can remove the duplicate types:

```
#include <boost/mpl/unique.hpp>

typedef boost::mpl::unique<
    noncv_types,
    boost::is_same<boost::mpl::_1, boost::mpl::_2>
>::type unique_types;
```

8. We may check that the vector contains only `5` types:

```
#include <boost/mpl/size.hpp>

BOOST_STATIC_ASSERT((boost::mpl::size<unique_types>::value == 5));
```

9. Here's how we can compute sizes of each element:

```
// Without this we'll get an error:
// "use of undefined type 'non_declared'"
struct non_declared{};

#include <boost/mpl/sizeof.hpp>
typedef boost::mpl::transform<
    unique_types,
    boost::mpl::sizeof_<boost::mpl::_1>
>::type sizes_types;
```

10. This is how to get the maximum size from the `sizes_type` type:

```
#include <boost/mpl/max_element.hpp>

typedef boost::mpl::max_element<sizes_types>::type max_size_type;
```

We may assert that the maximum size of the type is equal to the declared size of the structure, which must be the biggest one in our example:

```
BOOST_STATIC_ASSERT(max_size_type::type::value == sizeof(declared));
```

How it works…

The `boost::mpl::vector` class is a compile-time container that holds types. To be more precise, it is a type that holds types. We do not make instances of it; instead, we are just using it in `typedef`s.

Unlike the standard library containers, the `Boost.MPL` containers have no member methods. Instead, methods are declared in a separate header. So to use some methods, we need to:

1. Include the correct header.
2. Call that method, usually by specifying the container as a first parameter.

We already saw metafunctions in Chapter 4, *Compile-time Tricks*. We were using some metafunctions (such as `boost::is_same`) from the familiar `Boost.TypeTraits` library.

So, in *step 3*, *step 4*, and *step 5* we are just calling metafunctions for our container type.

The hardest part is coming up!

Placeholders are widely used by the `Boost.MPL` library for combining the metafunctions:

```
typedef boost::mpl::transform<
    types,
    boost::remove_cv<boost::mpl::_1>
>::type noncv_types;
```

Here, `boost::mpl::_1` is a placeholder and the whole expression means, for each type in `types`, do `boost::remove_cv<>::type` and push back that type to the resulting vector. Return the resulting vector via `::type`.

Let's move to *step 7*. Here, we specify a comparison metafunction for `boost::mpl::unique` using the `boost::is_same<boost::mpl::_1, boost::mpl::_2>` template parameter, where `boost::mpl::_1` and `boost::mpl::_2` are placeholders. You may find it similar to `boost::bind(std::equal_to(), _1, _2)`, and the whole expression in *step 7* is similar to the following pseudo code:

```
std::vector<type> t; // 't' stands for 'types'.
std::unique(t.begin(), t.end(), boost::bind(std::equal_to<type>(), _1, _2));
```

There is something interesting, which is required for better understanding, in *step 9*. In the preceding code, `sizes_types` is not a vector of values, but rather a vector of integral constants-types representing numbers. The `sizes_types typedef` is actually a following type:

```
struct boost::mpl::vector<
    struct boost::mpl::size_t<4>,
    struct boost::mpl::size_t<4>,
    struct boost::mpl::size_t<4096>,
    struct boost::mpl::size_t<1>,
    struct boost::mpl::size_t<32>
>
```

The final step must be clear now. It just gets the maximum element from the `sizes_types typedef`.

We may use the `Boost.MPL` metafunctions at any place where typedefs are *allowed.*

There’s more…

The `Boost.MPL` library usage results in longer compilation times, but gives you the ability to do everything you want with types. It does not add runtime overhead and won’t even add a single instruction to the resulting binary. C++17 has no `Boost.MPL` classes, and `Boost.MPL` does not use features of modern C++, such as the variadic templates. This makes the `Boost.MPL` compilation times not as short as possible on C++11 compilers, but makes the library usable on C++03 compilers.

See also

- See Chapter 4, *Compile-time Tricks* for information basics of metaprogramming
- The *Manipulating vector of types* recipe will give you even more information about metaprogramming and the `Boost.MPL` library
- See the official documentation of `Boost.MPL` for more examples and full reference at http://boost.org/libs/mpl

Manipulating a vector of types

The task of this recipe is to modify the content of one `boost::mpl::vector` function depending on the content of a second `boost::mpl::vector` function. We'll be calling the second vector as the vector of modifiers and each of those modifiers may have the following type:

```
// Make unsigned.
struct unsigne; // Not a typo: `unsigned` is a keyword, we can not use it.

// Make constant.
struct constant;

// Otherwise we do not change type.
struct no_change;
```

So, where shall we start from?

Getting ready

The basic knowledge of `Boost.MPL` is required. Reading the *Using type vector of types* recipe and Chapter 4, *Compile-time Tricks*, may help.

How to do it…

This recipe is similar to the previous one, but it also uses conditional compile-time statements. Get ready, it won’t be easy!

1. We shall start with headers:

```
// We'll need this at step 3.
#include <boost/mpl/size.hpp>
#include <boost/type_traits/is_same.hpp>
#include <boost/static_assert.hpp>

// We'll need this at step 4.
#include <boost/mpl/if.hpp>
#include <boost/type_traits/make_unsigned.hpp>
#include <boost/type_traits/add_const.hpp>

// We'll need this at step 5.
#include <boost/mpl/transform.hpp>
```

2. Now, let’s put all the metaprogramming magic inside the structure, for simpler reuse:

```
template <class Types, class Modifiers>
struct do_modifications {
```

3. It is a good thought to check that the passed vectors have the same size:

```
    BOOST_STATIC_ASSERT((boost::is_same<
        typename boost::mpl::size<Types>::type,
        typename boost::mpl::size<Modifiers>::type
    >::value));
```

4. Now, let’s take care of the modifying metafunction:

```
    typedef boost::mpl::if_<
        boost::is_same<boost::mpl::_2, unsigne>,
        boost::make_unsigned<boost::mpl::_1>,
        boost::mpl::if_<
            boost::is_same<boost::mpl::_2, constant>,
            boost::add_const<boost::mpl::_1>,
            boost::mpl::_1
        >
    > binary_operator_t;
```

5. And the final step:

```
    typedef typename boost::mpl::transform<
        Types,
        Modifiers,
        binary_operator_t
    >::type type;
};
```

We will now run some tests and make sure that our metafunction works great:

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/at.hpp>

typedef boost::mpl::vector<
    unsigne, no_change, constant, unsigne
> modifiers;
```

```
typedef boost::mpl::vector<
    int, char, short, long
> types;

typedef do_modifications<types, modifiers>::type result_type;

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 0>::type,
    unsigned int
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 1>::type,
    char
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 2>::type,
    const short
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<result_type, 3>::type,
    unsigned long
>::value));
```

How it works…

In step *3*, we assert that sizes are equal, but we do it in an unusual way. The `boost::mpl::size<Types>::type` metafunction actually returns an integral constant `struct boost::mpl::long_<4>`, so in a static assertion, we actually compare two types, not two numbers. This can be rewritten in a more familiar way:

```
    BOOST_STATIC_ASSERT((
        boost::mpl::size<Types>::type::value
        ==
        boost::mpl::size<Modifiers>::type::value
    ));
```

Notice the `typename` keyword we use. Without it, the compiler won't be able to decide if `::type` is actually a type or some variable. Previous recipes did not require it, because parameters for the metafunction were fully known at the point where we were using them. But in this recipe, parameter for the metafunction is a template.

We'll take a look at *step 5*, before taking care of *step 4*. In *step 5*, we provide the `Types`, `Modifiers`, and `binary_operator_t` parameters from *step 4* to the `boost::mpl::transform` metafunction. This metafunction is rather simple—for each passed vector, it takes an element and passes it to a third parameter—a binary metafunction. If we rewrite it in pseudo code, it will look like the following:

```
void boost_mpl_transform_pseoudo_code() {
    vector result;
    for (std::size_t i = 0; i < Types.size(); ++i) {
        result.push_back(
            binary_operator_t(Types[i], Modifiers[i])
        );
    }
    return result;
}
```

Step 4 may make someone's head hurt. At this step, we write a metafunction that is called for each pair of types from the `Types` and `Modifiers` vectors (see the preceding pseudo code):

```
    typedef boost::mpl::if_<
        boost::is_same<boost::mpl::_2, unsigne>,
        boost::make_unsigned<boost::mpl::_1>,
        boost::mpl::if_<
            boost::is_same<boost::mpl::_2, constant>,
            boost::add_const<boost::mpl::_1>,
            boost::mpl::_1
        >
    > binary_operator_t;
```

As we already know, `boost::mpl::_2` and `boost::mpl::_1` are placeholders. In this recipe, `_1` is a placeholder for a type from the `Types` vector and `_2` is a placeholder for a type from the `Modifiers` vector.

So, the whole metafunction works like this:

1. Compares the second parameter passed to it (via `_2`) with an `unsigned` type.
2. If types are equal, makes the first parameter passed to it (via `_1`) unsigned and returns that type.

3. Otherwise, it compares the second parameter passed to it (via `_2`) with a constant type.
4. If types are equal, it makes the first parameter passed to it (via `_1`) constant and, returns that type.
5. Otherwise, it returns the first parameter passed to it (via `_1`).

We need to be very careful while constructing this metafunction. Additional care should be taken as to not call `::type` at the end of it:

```
>::type binary_operator_t; // INCORRECT!
```

If we call `::type`, the compiler will attempt to evaluate the binary operator at this point, and this leads to a compilation error. In pseudo code, such an attempt would look like this:

```
binary_operator_t foo;
// Attempt to call binary_operator_t::operator() without parameters,
// when it has only two parameters overloads.
foo();
```

There’s more…

Working with metafunctions requires some practice. Even your humble servant cannot write some functions correctly from the first attempt (second and third attempts are also not good though). Do not be afraid or confused to experiment!

The `Boost.MPL` library is not a part of C++17 and does not use modern C++ features, but it can be used with C++11 variadic templates:

```
template <class... T>
struct vt_example {
    typedef typename boost::mpl::vector<T...> type;
};

BOOST_STATIC_ASSERT((boost::is_same<
    boost::mpl::at_c<vt_example<int, char, short>::type, 0>::type,
    int
>::value));
```

Just as always, metafunctions do not add a single instruction to the resulting binary file and do not make performance worse. However, using them you may make your code more tuned to a specific situation.

See also

- Read this chapter from the beginning to get more simple examples of the `Boost.MPL` usage
- See Chapter 4, *Compile-time Tricks*, especially the *Selecting an optimal operator for a template parameter* recipe, which contains code similar to the `binary_operator_t` metafunction
- Official documentation of `Boost.MPL` has more examples and a full table of contents at http://boost.org/libs/mpl

Getting a function’s result type at compile time

Many good features were added to C++11 to simplify the metaprogramming. One such feature is the alternative function syntax. It allows deducing the result type of a template function. Here is an example:

```
template <class T1, class T2>
auto my_function_cpp11(const T1& v1, const T2& v2)
    -> decltype(v1 + v2)
{
    return v1 + v2;
}
```

It allows us to write generic functions more easily:

```
#include <cassert>

struct s1 {};
struct s2 {};
struct s3 {};

inline s3 operator + (const s1& /*v1*/, const s2& /*v2*/) {
    return s3();
}

inline s3 operator + (const s2& /*v1*/, const s1& /*v2*/) {
    return s3();
}

int main() {
    s1 v1;
    s2 v2;

    s3 res0 = my_function_cpp11(v1, v2);
    assert(my_function_cpp11('\0', 1) == 1);
}
```

But, Boost has a lot of functions like these and it does not require C++11 to work. How is that possible and how can we make a C++03 version of the `my_function_cpp11` function?

Getting ready

Basic knowledge of C++ and templates is required for this recipe.

How to do it…

C++11 greatly simplifies metaprogramming. A lot of code must be written in C++03 to make something close to the alternative functions syntax:

1. We have to include the following header:

```
#include <boost/type_traits/common_type.hpp>
```

2. Now, let's make a metafunction in the `result_of` namespace for any types:

```
namespace result_of {

    template <class T1, class T2>
    struct my_function_cpp03 {
        typedef typename boost::common_type<T1, T2>::type type;
    };
```

3. And specialize it for types `s1` and `s2`:

```
    template <>
    struct my_function_cpp03<s1, s2> {
        typedef s3 type;
    };

    template <>
    struct my_function_cpp03<s2, s1> {
        typedef s3 type;
    };
} // namespace result_of
```

4. Now we are ready to write the `my_function_cpp03` function:

```
template <class T1, class T2>
typename result_of::my_function_cpp03<T1, T2>::type
    my_function_cpp03(const T1& v1, const T2& v2)
{
    return v1 + v2;
}
```

That's it! Now, we can use this function like a C++11 one:

```
int main() {
    s1 v1;
    s2 v2;

    s3 res1 = my_function_cpp03(v1, v2);
    assert(my_function_cpp03('\0', 1) == 1);
}
```

How it works…

The main idea of this recipe is that we may make a special metafunction that deduces the resulting type. Such a technique can be seen all around the Boost libraries, for example, in the `Boost.Variant`‘s implementation of `boost::get<>` or in almost any function from `Boost.Fusion`.

Now, let’s move step-by-step. The `result_of` namespace is just some kind of a tradition, but you may use your own and it does not matter. The `boost::common_type<>` metafunction deduces a type common of several types, so we use it for a general case. We also added two template specializations of the `result_of::my_function_cpp03` structures for the `s1` and `s2` types.

The disadvantage of writing metafunctions in C++03 is that sometimes we are required to write a lot. Compare the amount of code for `my_function_cpp11` and `my_function_cpp03` including the `result_of` namespace to feel the difference.

When the metafunction is ready, we may deduce the resulting type without C++11:

```
template <class T1, class T2>
typename result_of::my_function_cpp03<T1, T2>::type
    my_function_cpp03(const T1& v1, const T2& v2);
```

There’s more…

This technique does not add runtime overhead but it may slow down compilation a little bit. You may use it on modern C++ compilers as well.

See also

- The recipes *Enabling the usage of templated functions for integral types*, *Disabling templated function usage for real types*, and *Selecting an optimal operator for a template parameter* recipes from Chapter 4, *Compile-time Tricks*, will give you much more information about `Boost.TypeTraits` and metaprogramming
- Consider the official documentation of `Boost.TypeTraits` for more information about ready metafunctions at http://boost.org/libs/type_traits

Making a higher-order metafunction

Functions that accept other functions as an input parameter or functions that return other functions are called **higher-order** functions. For example, the following functions are higher order:

```
typedef void(*function_t)(int);

function_t higher_order_function1();
void higher_order_function2(function_t f);
function_t higher_order_function3(function_t f); f);
```

We already saw higher-order metafunctions in the recipes *Using type vector of types* and *Manipulating vector of types* recipes from this chapter, where we used `boost::mpl::transform`.

In this recipe, we'll try to make our own higher-order metafunction named `coalesce`, which accepts two types and two metafunctions. The `coalesce` metafunction applies the first type-parameter to the first metafunction and compares the resulting type with the `boost::mpl::false_` type. If the resulting type is the `boost::mpl::false_` type, it returns the result of applying the second type-parameter to the second metafunction, otherwise, it returns the first resulting type:

```
template <class Param1, class Param2, class Func1, class Func2>
struct coalesce;
```

Getting ready

This recipe (and chapter) is a tricky one. Reading this chapter from the beginning is highly recommended.

How to do it…

The `Boost.MPL` metafunctions are actually structures that can be easily passed as a template parameter. The hard part is to use it correctly:

1. We need the following headers to write a higher-order metafunction:

```
#include <boost/mpl/apply.hpp>
#include <boost/mpl/if.hpp>
#include <boost/type_traits/is_same.hpp>
```

2. The next step is to evaluate our functions:

```
template <class Param1, class Param2, class Func1, class Func2>
struct coalesce {
    typedef typename boost::mpl::apply<Func1, Param1>::type type1;
    typedef typename boost::mpl::apply<Func2, Param2>::type type2;
```

3. Now, we need to choose the correct result type:

```
    typedef typename boost::mpl::if_<
        boost::is_same< boost::mpl::false_, type1>,
        type2,
        type1
    >::type type;
};
```

That’s it! We have completed a higher-order metafunction! Now, we may use it just like that:

```
#include <boost/static_assert.hpp>
#include <boost/mpl/not.hpp>
#include <boost/mpl/next.hpp>

using boost::mpl::_1;
using boost::mpl::_2;

typedef coalesce<
    boost::mpl::true_,
    boost::mpl::int_<5>,
    boost::mpl::not_<_1>,
    boost::mpl::next<_1>
>::type res1_t;
BOOST_STATIC_ASSERT((res1_t::value == 6));

typedef coalesce<
    boost::mpl::false_,
    boost::mpl::int_<5>,
    boost::mpl::not_<_1>,
    boost::mpl::next<_1>
>::type res2_t;
BOOST_STATIC_ASSERT((res2_t::value));
```

How it works…

The main problem with writing the higher-order metafunctions is taking care of the placeholders. That's why we shall not call `Func1<Param1>::type` directly. Instead of that, we must use the `boost::mpl::apply` metafunction, which accepts one function and up to five parameters that shall be passed to this function.

You may configure `boost::mpl::apply` to accept even more parameters, defining the `BOOST_MPL_LIMIT_METAFUNCTION_ARITY` macro to the required amount of parameters, for example, to 6.

There's more…

C++11 has nothing close to the `Boost.MPL` library to apply a metafunction.

Modern C++ has a bunch of features that may help you achieve the `Boost.MPL` functionality. For example, C++11 has a `<type_traits>` header and **basic constexpr** support. C++14 has an **extended constexpr** support, in C++17 there's an `std::apply` function that works with tuples and is usable in constant expressions. Also, in C++17 lambdas are constexpr by default and there is an **if constexpr** (expr).

Writing your own solution would waste a lot of time and probably would not work on older compilers. So, `Boost.MPL` still remains one of the most suitable solutions for metaprogramming.

See also

See the official documentation, especially the *Tutorial* section, for more information about `Boost.MPL` at http://boost.org/libs/mpl.

Evaluating metafunctions lazily

Lazy evaluation means that the function is not called until we really need its result. Knowledge of this recipe is highly recommended for writing good metafunctions. The importance of lazy evaluation will be shown in the following example.

Imagine that we are writing some metafunction that accepts a function `Func`, a parameter `Param`, and a condition `Cond`. The resulting type of that function must be a `fallback` type if applying the `Cond` to `Param` returns `false`, otherwise the result must be a `Func` applied to `Param`:

```
struct fallback;

template <
        class Func,
        class Param,
        class Cond,
        class Fallback = fallback>
struct apply_if;
```

This metafunction is the place where we cannot live without lazy evaluation, because it may be impossible to apply `Func` to `Param` if the `Cond` is not met. Such attempts will always result in compilation failure and `Fallback` is never returned.

Getting ready

Reading Chapter 4, *Compile-time Tricks*, is highly recommended. However, a good knowledge of metaprogramming should be enough.

How to do it…

Keep an eye on the small details, like not calling `::type` in the example:

1. We’ll need the following headers:

```
        #include <boost/mpl/apply.hpp>
        #include <boost/mpl/eval_if.hpp>
        #include <boost/mpl/identity.hpp>
```

2. The beginning of the function is simple:

```
        template <class Func, class Param, class Cond, class Fallback>
        struct apply_if {
            typedef typename boost::mpl::apply<
                Cond, Param
            >::type condition_t;
```

3. We shall be careful here:

```
            typedef boost::mpl::apply<Func, Param> applied_type;
```

4. Additional care must be taken when evaluating an expression:

```
            typedef typename boost::mpl::eval_if_c<
                condition_t::value,
                applied_type,
                boost::mpl::identity<Fallback>
            >::type type;
        };
```

That’s it! Now we are free to use it like this:

```
#include <boost/static_assert.hpp>
#include <boost/type_traits/is_integral.hpp>
#include <boost/type_traits/make_unsigned.hpp>
#include <boost/type_traits/is_same.hpp>

using boost::mpl::_1;
using boost::mpl::_2;

typedef apply_if<
    boost::make_unsigned<_1>,
    int,
    boost::is_integral<_1>
>::type res1_t;

BOOST_STATIC_ASSERT((
    boost::is_same<res1_t, unsigned int>::value
));

typedef apply_if<
    boost::make_unsigned<_1>,
    float,
    boost::is_integral<_1>
>::type res2_t;

BOOST_STATIC_ASSERT((
    boost::is_same<res2_t, fallback>::value
));
```

How it works…

The main idea of this recipe is that we must not execute the metafunction if the condition is `false`, because when the condition is `false`, there is a chance that the metafunction for that type can't be applied:

```
// Will fail with static assertion somewhere deeply in the implementation
// of boost::make_unsigned<_1> if we do not evaluate the function lazily.
typedef apply_if<
    boost::make_unsigned<_1>,
    float,
    boost::is_integral<_1>
>::type res2_t;

BOOST_STATIC_ASSERT((
    boost::is_same<res2_t, fallback>::value
));
```

So, how do we evaluate a metafunction lazily?

The compiler does look inside the metafunction if there is no access to the metafunction's internal types or values. In other words, the compiler tries to compile the metafunction when we try to get one of its members via `::`. This can be a call to `::type` or `::value`. That is how an incorrect version of `apply_if` looks like:

```
template <class Func, class Param, class Cond, class Fallback>
struct apply_if {
    typedef typename boost::mpl::apply<
        Cond, Param
    >::type condition_t;

    // Incorrect: metafunction is evaluated when `::type` called.
    typedef typename boost::mpl::apply<Func, Param>::type applied_type;

    typedef typename boost::mpl::if_c<
        condition_t::value,
        applied_type,
        boost::mpl::identity<Fallback>
    >::type type;
};
```

This differs from our example, where we did not call `::type` at *step 3* and implemented *step 4* using `eval_if_c`, which calls `::type` only for one of its parameters. The `boost::mpl::eval_if_c` metafunction is implemented like this:

```
template<bool C, typename F1, typename F2>
struct eval_if_c {
    typedef typename if_c<C,F1,F2>::type f_;
    typedef typename f_::type type; // call `::type` only for one parameter
};
```

Because `boost::mpl::eval_if_c` calls `::type` for a succeeded condition and `fallback` has no `::type`, we were required to wrap `fallback` into the `boost::mpl::identity` class. This class is very simple, but useful structure that returns its template parameter via a `::type` call and does nothing more:

```
template <class T>
struct identity {
    typedef T type;
};
```

There’s more…

As we already mentioned, C++11 has no classes of `Boost.MPL`, but we may use `std::common_type<T>` with a single argument just like `boost::mpl::identity<T>`.

Just as always, metafunctions do not add a single line to the output binary file, you can use metafunctions as many times as you want. The more you do at compile time, the less remains for the runtime.

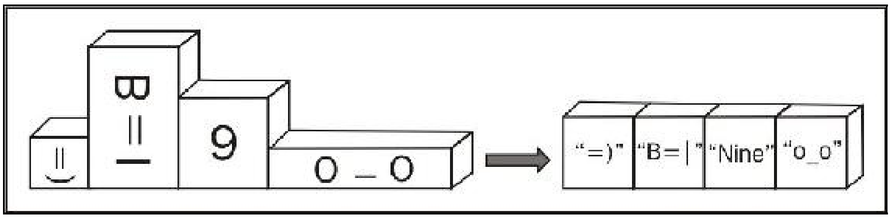
See also…

- The `boost::mpl::identity` type can be used to disable **Argument Dependent Lookup** (**ADL**) for template functions. See sources of `boost::implicit_cast` in the `<boost/implicit_cast.hpp>` header.
- Reading this chapter from the beginning and the official documentation of `Boost.MPL` at http://boost.org/libs/mpl may help.

Converting all the tuple elements to strings

This recipe and the next one are devoted to a mix of compile time and runtime features. We’ll be using the `Boost.Fusion` library and see what it can do.

Remember that we were talking about tuples and arrays in the first chapter? Now, we want to write a single function that can stream elements of tuples and arrays to strings.



Getting ready

You should be aware of the `boost::tuple` and `boost::array` classes and of the `boost::lexical_cast` function.

How to do it…

We already know almost all the functions and classes that will be used in this recipe. We just need to gather all of them together:

1. We need to write a functor that converts any type to string:

```
#include <boost/lexical_cast.hpp>
#include <boost/noncopyable.hpp>

struct stringize_functor: boost::noncopyable {
private:
    std::string& result;

public:
    explicit stringize_functor(std::string& res)
        : result(res)
    {}

    template <class T>
    void operator()(const T& v) const {
        result += boost::lexical_cast<std::string>(v);
    }
};
```

2. And now is the tricky part of the code:

```
#include <boost/fusion/include/for_each.hpp>

template <class Sequence>
std::string stringize(const Sequence& seq) {
    std::string result;
    boost::fusion::for_each(seq, stringize_functor(result));
    return result;
}
```

That’s all! Now, we may convert anything we want to string:

```
#include <iostream>
#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/adapted/boost_tuple.hpp>
#include <boost/fusion/adapted/std_pair.hpp>
#include <boost/fusion/adapted/boost_array.hpp>

struct cat{};

std::ostream& operator << (std::ostream& os, const cat& ) {
    return os << "Meow! ";
}

int main() {
    boost::fusion::vector<cat, int, std::string> tup1(cat(), 0, "_0");
    boost::tuple<cat, int, std::string> tup2(cat(), 0, "_0");
    std::pair<cat, cat> cats;
    boost::array<cat, 10> many_cats;

    std::cout << stringize(tup1) << '\n'
        << stringize(tup2) << '\n'
        << stringize(cats) << '\n'
        << stringize(many_cats) << '\n';
}
```

The preceding example outputs the following:

```
Meow! 0_0
Meow! 0_0
```

```
Meow! Meow!
Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow! Meow!
```

How it works…

The main problem of the `stringize` function is that neither `boost::tuple` nor `std::pair` have `begin()` or `end()` methods, so we cannot call `std::for_each`. This is where the `Boost.Fusion` steps in.

The `Boost.Fusion` library contains lots of terrific algorithms that may manipulate structures at compile time.

The `boost::fusion::for_each` function iterates through elements of sequence and applies a functor for each of the elements.

Note that we have included:

```
#include <boost/fusion/adapted/boost_tuple.hpp>
#include <boost/fusion/adapted/std_pair.hpp>
#include <boost/fusion/adapted/boost_array.hpp>
```

This is required because by default `Boost.Fusion` works only with its own classes. `Boost.Fusion` has its own tuple class, `boost::fusion::vector`, which is quite close to `boost::tuple`:

```
#include <string>
#include <cassert>

#include <boost/tuple/tuple.hpp>

#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/include/at_c.hpp>

void tuple_example() {
    boost::tuple<int, int, std::string> tup(1, 2, "Meow");
    assert(boost::get<0>(tup) == 1);
    assert(boost::get<2>(tup) == "Meow");
}

void fusion_tuple_example() {
    boost::fusion::vector<int, int, std::string> tup(1, 2, "Meow");
    assert(boost::fusion::at_c<0>(tup) == 1);
    assert(boost::fusion::at_c<2>(tup) == "Meow");
}
```

But `boost::fusion::vector` is not as simple as `boost::tuple`. We’ll see the difference in the *Splitting tuples* recipe.

There's more…

There is one fundamental difference between `boost::fusion::for_each` and `std::for_each`. The `std::for_each` function contains a loop inside it and determinates at runtime, how many iterations must be done. However, `boost::fusion::for_each()` knows iterations count at compile time and fully unrolls the loop. For the `boost::tuple<cat, int, std::string> tup2`, the `boost::fusion::for_each(tup2, functor)` call is equivalent to the following code:

```
    functor(boost::fusion::at_c<0>(tup2));
    functor(boost::fusion::at_c<1>(tup2));
    functor(boost::fusion::at_c<2>(tup2));
```

C++11 contains no `Boost.Fusion` classes. All the methods of `Boost.Fusion` are very effective. They do as much as possible at compile time and have some very advanced optimizations.

C++14 adds `std::integer_sequence` and `std::make_integer_sequence` to simplify for with variadic templates. Using those entities it is possible to hand write the `boost::fusion::for_each` functionality and implement the `stringize` function without `Boost.Fusion`:

```
#include <utility>
#include <tuple>

template <class Tuple, class Func, std::size_t... I>
void stringize_cpp11_impl(const Tuple& t, const Func& f, std::index_sequence<I...>) {
    // Oops. Requires C++17 fold expressions feature.
    // (f(std::get<I>(t)), ...);

    int tmp[] = { 0, (f(std::get<I>(t)), 0)... };
    (void)tmp; // Suppressing unused variable warnings.
}

template <class Tuple>
std::string stringize_cpp11(const Tuple& t) {
    std::string result;
    stringize_cpp11_impl(
        t,
        stringize_functor(result),
        std::make_index_sequence< std::tuple_size<Tuple>::value >()
    );
    return result;
}
```

As you may see a lot of code was written to do that and such code is not simple to read and understand.

Ideas on adding something close to a `constexpr for` to the C++20 Standard are discussed in the C++ Standardization Working Group. With that feature, some day we could write the following code (syntax may change!):

```
template <class Tuple>
std::string stringize_cpp20(const Tuple& t) {
    std::string result;
    for constexpr(const auto& v: t) {
        result += boost::lexical_cast<std::string>(v);
    }
    return result;
}
```

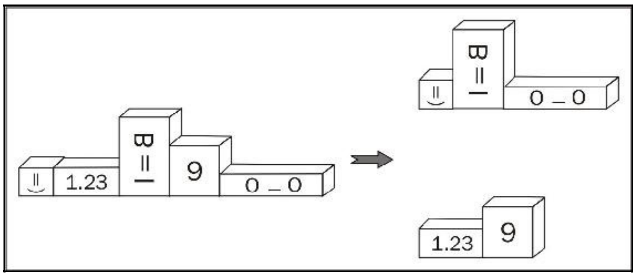
Until then, `Boost.Fusion` seems to be the most portable and simple solution.

See also

- The *Splitting tuples* recipe will give more information about the true power of `Boost.Fusion`
- The official documentation of `Boost.Fusion` contains some interesting examples and full references, which can be found at http://boost.org/libs/fusion

Splitting tuples

This recipe will show a tiny piece of the `Boost.Fusion` library's abilities. We'll be splitting a single tuple into two tuples, one with arithmetic types and the other with all other types.



Getting ready

This recipe requires knowledge of `Boost.MPL`, placeholders, and `Boost.Tuple`. Reading this chapter from the beginning is recommended.

How to do it…

This is possibly one of the hardest recipes in this chapter. The resulting types are determined at compile time and values for those types are filled at runtime:

1. To implement that mix, we need the following headers:

```
#include <boost/fusion/include/remove_if.hpp>
#include <boost/type_traits/is_arithmetic.hpp>
```

2. Now, we are ready to make a function that returns non-arithmetic types:

```
template <class Sequence>
typename boost::fusion::result_of::remove_if<
    const Sequence,
    boost::is_arithmetic<boost::mpl::_1>
>::type get_nonarithmetics(const Sequence& seq)
{
    return boost::fusion::remove_if<
        boost::is_arithmetic<boost::mpl::_1>
    >(seq);
}
```

3. And a function that returns arithmetic types:

```
template <class Sequence>
typename boost::fusion::result_of::remove_if<
    const Sequence,
    boost::mpl::not_< boost::is_arithmetic<boost::mpl::_1> >
>::type get_arithmetics(const Sequence& seq)
{
    return boost::fusion::remove_if<
        boost::mpl::not_< boost::is_arithmetic<boost::mpl::_1> >
    >(seq);
}
```

That’s it! Now, we are capable of doing the following tasks:

```
#include <boost/fusion/include/vector.hpp>
#include <cassert>
#include <boost/fusion/include/at_c.hpp>
#include <boost/blank.hpp>

int main() {
    typedef boost::fusion::vector<
        int, boost::blank, boost::blank, float
    > tup1_t;
    tup1_t tup1(8, boost::blank(), boost::blank(), 0.0);

    boost::fusion::vector<boost::blank, boost::blank> res_na
        = get_nonarithmetics(tup1);
    boost::fusion::vector<int, float> res_a = get_arithmetics(tup1);
    assert(boost::fusion::at_c<0>(res_a) == 8);
}
```

How it works…

The idea behind `Boost.Fusion` is that the compiler knows the structure layout at compile time and whatever the compiler knows at compile time, we may change at the same time. The `Boost.Fusion` allows us to modify different sequences, adding and removing fields, and changing field types. This is what we did in *step 2* and *step 3*; we removed the non-required fields from the tuple.

Now, let’s take a very close look at `get_nonarithmetics`. First of all, its result type is deduced using the following construction:

```
typename boost::fusion::result_of::remove_if<
    const Sequence,
    boost::is_arithmetic<boost::mpl::_1>
>::type
```

This must be familiar to us. We saw something like this in the *Getting function result type at compile-time* recipe in this chapter. The `Boost.MPL`‘s placeholder `boost::mpl::_1` works well with the `boost::fusion::result_of::remove_if` metafunction that returns a new sequence type.

Now, let’s move inside the function and we watch the following code:

```
    return boost::fusion::remove_if<
        boost::is_arithmetic<boost::mpl::_1>
    >(seq);
```

Remember that the compiler knows all the types of `seq` at compile time. This means that `Boost.Fusion` may apply metafunctions for different elements of `seq` and get the metafunction results for them. This also means that `Boost.Fusion` knows how to copy required fields from the old structure to the new one.

However, `Boost.Fusion` tries not to copy fields as long as possible.

The code in *step 3* is very similar to the code in *step 2*, but it has a negated predicate for removing non-required types.

Our functions can be used with any type supported by `Boost.Fusion` and not just with `boost::fusion::vector`.

There’s more…

You may use `Boost.MPL` functions for the `Boost.Fusion` containers. You just need to include `#include <boost/fusion/include/mpl.hpp>`:

```
#include <boost/fusion/include/mpl.hpp>
#include <boost/mpl/transform.hpp>
#include <boost/type_traits/remove_const.hpp>

template <class Sequence>
struct make_nonconst: boost::mpl::transform<
    Sequence,
    boost::remove_const<boost::mpl::_1>
> {};

typedef boost::fusion::vector<
    const int, const boost::blank, boost::blank
> type1;
typedef make_nonconst<type1>::type nc_type;

BOOST_STATIC_ASSERT((boost::is_same<
    boost::fusion::result_of::value_at_c<nc_type, 0>::type,
    int
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::fusion::result_of::value_at_c<nc_type, 1>::type,
    boost::blank
>::value));

BOOST_STATIC_ASSERT((boost::is_same<
    boost::fusion::result_of::value_at_c<nc_type, 2>::type,
    boost::blank
>::value));
```

We used `boost::fusion::result_of::value_at_c` instead of `boost::fusion::result_of::at_c` because `boost::fusion::result_of::at_c` returns the exact return type of the `boost::fusion::at_c` call, which is a reference. `boost::fusion::result_of::value_at_c` returns type without a reference.

The `Boost.Fusion` and `Boost.MPL` libraries are not a part of C++17. `Boost.Fusion` is extremely fast. It has many optimizations.

It is worth mentioning that we saw only a tiny part of the `Boost.Fusion` abilities. A separate book can be written about it.

See also

- Good tutorials and full documentation for `Boost.Fusion` is available at http://boost.org/libs/fusion
- You may also wish to see an official documentation for `Boost.MPL` at http://boost.org/libs/mpl

Manipulating heterogeneous containers in C++14

Most of the metaprogramming tricks that we saw in this chapter were invented long before C++11. Probably, you’ve already heard about some of that stuff.

How about something brand new? How about implementing the previous recipe in C++14 with a library that puts the metaprogramming upside down and makes your eyebrows go up? Fasten your seatbelts, we’re diving into the world of `Boost.Hana`.

Getting ready

This recipe requires knowledge of C++11 and C++14, especially lambdas. You will need a truly C++14 compatible compiler to compile the example.

How to do it…

Now, let’s make everything in the `Boost.Hana` way:

1. Start with including the header:

```
#include <boost/hana/traits.hpp>
```

2. We create an `is_arithmetic_` functional object:

```
constexpr auto is_arithmetic_ = [](const auto& v) {
    auto type = boost::hana::typeid_(v);
    return boost::hana::traits::is_arithmetic(type);
};
```

3. Now, we implement the `get_nonarithmetics` function:

```
#include <boost/hana/remove_if.hpp>

template <class Sequence>
auto get_nonarithmetics(const Sequence& seq)  {
    return boost::hana::remove_if(seq, [](const auto& v) {
        return is_arithmetic_(v);
    });
}
```

4. Let’s define `get_arithmetics` the other way around. Just for fun!

```
#include <boost/hana/filter.hpp>

constexpr auto get_arithmetics = [](const auto& seq) {
    return boost::hana::filter(seq, is_arithmetic_);
};
```

That’s it. Now, we can use these functions:

```
#include <boost/hana/tuple.hpp>
#include <boost/hana/integral_constant.hpp>
#include <boost/hana/equal.hpp>
#include <cassert>

struct foo {
    bool operator==(const foo&) const { return true; }
    bool operator!=(const foo&) const { return false; }
};

int main() {
    const auto tup1
        = boost::hana::make_tuple(8, foo{}, foo{}, 0.0);

    const auto res_na = get_nonarithmetics(tup1);
    const auto res_a = get_arithmetics(tup1);

    using boost::hana::literals::operator ""_c;
    assert(res_a[0_c] == 8);

    const auto res_na_expected = boost::hana::make_tuple(foo(), foo());
    assert(res_na == res_na_expected);
}
```

How it works…

The code may seem simple at first glance, but that’s not true. The `Boost.Hana` puts the metaprogramming the other way around! In the previous recipes, we were working with types directly, but `Boost.Hana` makes a variable that holds a type and works with variable most of the time.

Take a look at the `typeid_` call in *step 2*:

```
auto type = boost::hana::typeid_(v);
```

It actually returns a variable. Information about the type is now hidden inside the `type` variable and could be extracted by calling `decltype(type)::type`.

But let’s move line by line. In *step 2,* we store the generic lambda into the `is_arithmetic_` variable. From this point, we can use that variable as a functional object. Inside the lambda, we create a `type` variable that now holds information about the type of the `v`. The next line is a special wrapper around `std::is_arithmetic` that extracts information about the `v` type from the `type` variable and passes it to the `std::is_arithmetic` trait. Result of that call is a Boolean integral constant.

And now, the magic part! Lambda stored inside the `is_arithmetic_` variable is actually never called by `boost::hana::remove_if` and `boost::hana::filter` functions. All the `Boost.Hana`‘s functions that use it only need the result type of the lambda function, but not its body. We can safely change the definition and the whole example will continue to work well:

```
constexpr auto is_arithmetic_ = [] (const auto& v) {
    assert(false);
    auto type = boost::hana::typeid_(v);
    return boost::hana::traits::is_arithmetic(type);
};
```

In *steps 3* and *4,* we call `boost::hana::remove_if` and `boost::hana::filter` functions, respectively. In *step 3,* we used `is_arithmetic_` inside the lambda. In *step 4,* we used it directly. You can use any syntax you’d like, it’s just a matter of habit.

Finally in `main()`, we check that everything works as expected and that the element in tuple by index 0 is equal to `8`:

```
    using boost::hana::literals::operator ""_c;
    assert(res_a[0_c] == 8);
```

The best way to understand the `Boost.Hana` library is to experiment with it. You can do it online at http://apolukhin.github.io/Boost-Cookbook/.

There’s more…

There’s a small detail left undescribed. How does the tuple access by `operator[]` work? It is impossible to have a single function that returns different types!

This is very interesting if you meet this trick at first time. `Boost.Hana`‘s `operator ""_c` works with literals and constructs different types depending on the literal:

- If you write `0_c`, then `integral_constant<long long, 0>` is returned
- If you write `1_c`, then `integral_constant<long long, 1>` is returned
- If you write `2_c`, then `integral_constant<long long, 2>` is returned

The `boost::hana::tuple` class actually has many `operator[]` overloads, accepting different types of `integral_constant`. Depending on the value of integral constant the correct tuple element is returned. For example, if you write `some_tuple[1_c]` then `tuple::operator[](integral_constant<long long, 1>)` is called the the element by index `1` is returned.

`Boost.Hana` is not a part of C++17. However, the author of the library participates in the C++ Standardization meetings and proposes different interesting things for inclusion into the C++ Standard.

If you are expecting order of magnitude better compile times from `Boost.Hana` than from `Boost.MPL` then don’t. Currently compilers do not handle the `Boost.Hana` approach extremely well. This may change some day.

It’s worth looking at the source codes of the `Boost.Hana` library to discover new interesting ways of using C++14 features. All the Boost libraries could be found at GitHub https://github.com/boostorg.

See also

Official documentation has more examples, a full reference section, some more tutorials, and a compile-time performance section. Enjoy the `Boost.Hana` library at http://boost.org/libs/hana.

Containers

In this chapter, we will cover:

- Storing a few elements in the sequence container
- Storing most N elements in the sequence container
- Comparing strings in an ultra-fast manner
- Using an unordered set and map
- Making a map, where the value is also a key
- Using multi-index containers
- Getting benefits of a single linked list and memory pool
- Using flat associative containers

Introduction

This chapter is devoted to the Boost containers and the things directly connected with them. It provides information about the Boost classes that can be used in every day programming, and that will make your code much faster, and the development of new applications easier.

Containers differ not only by functionality, but also by the efficiency (complexity) of some of its members. The knowledge about complexities is essential for writing fast applications. This chapter doesn’t just introduce some new containers to you, it gives you tips on when and when not to use a specific type of container or its methods.

So, let’s begin!

Storing a few elements in a sequence container

For the past two decades, C++ programmers were using `std::vector` as a default sequence container. It is a fast container that does not do a lot of allocations, stores elements in a CPU cache friendly way and because container stores the elements contiguously `std::vector::data()` like functions allows to inter-operate with pure C functions.

But, we want more! There are cases when we do know the typical elements count to store in the vector, and we need to improve the performance of the vector by totally eliminating the memory allocations for that case.

Imagine that we are writing a high performance system for processing bank transactions. **Transaction** is a sequence of operations that must all succeed or fail if at least one of the operations failed. We know that the 99% of transactions consist of 8 or less operations and wish to speed up things:

```
#include <vector>

class operation;

template <class T>
void execute_operations(const T&);

bool has_operation();
operation get_operation();

void process_transaction_1() {
    std::vector<operation> ops;
    ops.reserve(8); // TODO: Memory allocation. Not good!

    while (has_operation()) {
        ops.push_back(get_operation());
    }

    execute_operations(ops);
    // ...
}
```

Getting ready

This recipe requires only the basic knowledge of standard library and C++.

How to do it…

This is going to be a simplest task of this book, thanks to the `Boost.Container` library:

1. Include the appropriate header:

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

2. Replace `std::vector` with `boost::container::small_vector` and drop the `reserve()` call:

```
void process_transaction_2() {
    boost::container::small_vector<operation, 8> ops;

    while (has_operation()) {
        ops.push_back(get_operation());
    }

    execute_operations(ops);
    // ...
}
```

How it works…

The `boost::container::small_vector`‘s second template parameter is the elements count to preallocate on a stack. So if most of the time we have to store 8 or less elements in the vector, we just put `8` as a second template parameter.

If we have to store more than 8 elements in the container, then the `small_vector` behaves exactly as `std::vector` and dynamically allocates a chunk of memory to store more than 8 elements. Just like `std::vector`, the `small_vector` is a sequence container with **random access iterators** that stores the elements consistently.

To sum up, `boost::container::small_vector` is a container that behaves exactly as `std::vector`, but allows to avoid memory allocations for a compile time specified amount of elements.

There’s more…

A drawback of using `small_vector` is that our elements count assumption leaks into the function signature that accepts `small_vector` as a parameter. So if we have three functions specialized for `4`, `8`, and `16` elements, respectively, and all those functions process transactions using `execute_operations` from the preceding example, we’ll end up with multiple instantiations of the `execute_operations` function:

```
void execute_operations(
    const boost::container::small_vector<operation, 4>&);

void execute_operations(
    const boost::container::small_vector<operation, 8>&);

void execute_operations(
    const boost::container::small_vector<operation, 16>&);
```

That’s not good! Now, we have multiple functions in our executable that do exactly the same thing and consist of almost exactly the same machine codes. This leads to bigger binaries, longer startup times of the executable, longer compile, and link times. Some compilers may eliminate the redundancy but chances are low.

However, the solution is very simple. `boost::container::small_vector` is derived from the `boost::container::small_vector_base` type that is independent from preallocated elements count:

```
void execute_operations(
    const boost::container::small_vector_base<operation>& ops
);
```

That’s it! Now, we may use the new `execute_operations` function with any `boost::container::small_vector` without a risk of bloating the binary size.

C++17 has no `small_vector` like class. There are proposals to include `small_vector` to the next C++ standard that will be out somewhere around year 2020.

See also

- The `Boost.Container` library has a full reference documentation for many interesting classes at http://boost.org/libs/container
- `small_vector` came to Boost from the **LLVM** project; you can read about the container at the origin site http://llvm.org/docs/ProgrammersManual.html#llvm-adt-smallvector-h

Storing at most N elements in the sequence container

Here’s a question: what container should we use to return the sequence from function if we know that the sequence never has more than *N* elements and *N* is not big. For example, how we must write the `get_events()` function that returns at most five events:

```
#include <vector>
std::vector<event> get_events();
```

The `std::vector<event>` allocates memory, so the code from earlier is not a good solution.

```
#include <boost/array.hpp>
boost::array<event, 5> get_events();
```

`boost::array<event, 5>` does not allocate memory, but it constructs all the five elements. There’s no way to return less than five elements.

```
#include <boost/container/small_vector.hpp>
boost::container::small_vector<event, 5> get_events();
```

The `boost::container::small_vector<event, 5>` does not allocate memory for five or less elements and allows us to return less than five elements. However, the solution is not perfect, because it is not obvious from the function interface that it never returns more than five elements.

Getting ready

This recipe requires only a basic knowledge of standard library and C++.

How to do it…

The `Boost.Container` has a container that perfectly satisfies our need:

```
#include <boost/container/static_vector.hpp>
boost::container::static_vector<event, 5> get_events();
```

How it works…

The `boost::container::static_vector<T, N>` is a container that does not allocate memory and can hold no more than a compile-time-specified amount of elements. Think of it as of `boost::container::small_vector<T, N>` that just cannot dynamically allocate memory and any attempt to store more than *N* elements results in a `std::bad_alloc` exception:

```
#include <cassert>

int main () {
    boost::container::static_vector<event, 5> ev = get_events();
    assert(ev.size() == 5);

    boost::container::static_vector<int, 2> ints;
    ints.push_back(1);
    ints.push_back(2);
    try {
        // The following line always throws:
        ints.push_back(3);
    } catch (const std::bad_alloc& ) {
        // ...
    }
}
```

Just like all the containers of the `Boost.Container` library, `static_vector` supports **move semantics** and emulates rvalue references using Boost.Move library if your compiler does not support rvalues.

There’s more…

The `std::vector` allocates bigger chunks of memory if a user inserts an element and it is impossible to fit the new value into the already allocated memory. In that case, `std::vector` moves elements from the old location to the new one if the elements are nothrow move constructible. Otherwise, `std::vector` copies elements to a new location and after that calls destructor for each element in the old location.

Because of that, behavior `std::vector` has amortized constant complexity for many member functions. The `static_vector` never allocates memory so that it does not have to move or copy elements from an old location to a new one. Because of that, operations that have **amortized O(1)** complexity for `std::vector` have true O(1) complexity for `boost::container::static_vector`. This may be handy for some real-time applications; though, beware of exceptions!

*Some people still prefer to pass output parameters by reference instead of returning them: `void get_events(static_vector<event, 5>& result_out)`. They think that this way, there’s a guarantee that no copying of result happens. Don’t do that, it makes things worse! C++ compilers have a whole bunch of optimizations, such as **Return Value Optimization** (**RVO**) and **Named Return Value Optimization** (**NRVO**); different platforms have agreements nailed down in ABI that code with `retun something;` does not result in an unnecessary copy and so forth. No copying happens already. However, when you pass a value, the reference compiler just does not see where the value came from and may assume that it aliases some other value in the scope. This may significantly degrade performance.*

C++17 has no `static_vector` class, and at this moment, there’s no plan to add it into C++20.

See also

The official documentation of `Boost.Container` has a detailed reference section that describes all the member functions of the `boost::container::static_vector` class. Refer to http://boost.org/libs/container.

Comparing strings in an ultra-fast manner

It is a common task to manipulate strings. Here, we’ll see how an operation of string comparison can be done quickly using some simple tricks. This recipe is a trampoline for the next one, where the techniques described here will be used to achieve constant time-complexity searches.

So, we need to make some class that is capable of quickly comparing strings for equality. We’ll make a template function to measure the speed of comparison:

```
#include <string>

template <class T>
std::size_t test_default() {
    // Constants
    const std::size_t ii_max = 200000;
    const std::string s(
        "Long long long string that "
        "will be used in tests to compare "
        "speed of equality comparisons."
    );

    // Making some data, that will be
    // used in comparisons.
    const T data1[] = {
        T(s),
        T(s + s),
        T(s + ". Whooohooo"),
        T(std::string(""))
    };

    const T data2[] = {
        T(s),
        T(s + s),
        T(s + ". Whooohooo"),
        T(std::string(""))
    };

    const std::size_t data_dimensions = sizeof(data1) / sizeof(data1[0]);

    std::size_t matches = 0u;
    for (std::size_t ii = 0; ii < ii_max; ++ii) {
        for (std::size_t i = 0; i < data_dimensions; ++i) {
            for (std::size_t j = 0; j < data_dimensions; ++j) {
                if (data1[i] == data2[j]) {
                    ++ matches;
                }
            }
        }
    }

    return matches;
}
```

Getting ready

This recipe requires only the basic knowledge of standard library and C++.

How to do it…

We’ll make `std::string` a public field in our own class and add all the comparison code to our class, without writing helper methods to work with the stored `std::string`, as shown in the following steps:

1. To do so, we’ll need the following header:

```
#include <boost/functional/hash.hpp>
```

2. Now, we can create our `fast comparison_` class:

```
struct string_hash_fast {
    typedef std::size_t comp_type;

    const comp_type     comparison_;
    const std::string   str_;

    explicit string_hash_fast(const std::string& s)
        : comparison_(
            boost::hash<std::string>()(s)
        )
        , str_(s)
    {}
};
```

3. Do not forget to define the `equality comparisons` operators:

```
inline bool operator == (
    const string_hash_fast& s1, const string_hash_fast& s2)
{
    return s1.comparison_ == s2.comparison_ && s1.str_ == s2.str_;
}

inline bool operator != (
    const string_hash_fast& s1, const string_hash_fast& s2)
{
    return !(s1 == s2);
}
```

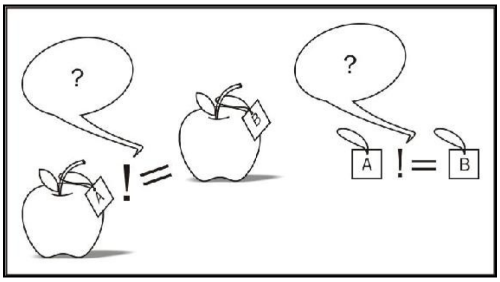
4. And, that’s it! Now, we can run our tests and see the result using the following code:

```
#include <iostream>
#include <iostream>
#include <cassert>

int main(int argc, char* argv[]) {
    if (argc < 2) {
        assert(
            test_default<string_hash_fast>()
            ==
            test_default<std::string>()
        );
        return 0;
    }

    switch (argv[1][0]) {
    case 'h':
        std::cout << "HASH matched: "
                  << test_default<string_hash_fast>();
        break;

    case 's':
            std::cout << "STD matched: "
                      << test_default<std::string>();
            break;

        default:
            return 2;
        }
    }
```

How it works…

The comparison of strings is slow because we are required to compare all the characters of the string one-by-one, if strings have equal length. Instead of doing that, we replace the comparison of strings with the comparison of integers. This is done via the `hash` function - the function that makes some short-fixed length representation of the string.

Let's talk about the `hash` values on apples. Imagine that you have two apples with labels, as shown in the following diagram, and you wish to check that the apples are of the same cultivar. The simplest way to compare those apples is by comparing them by labels. Otherwise, you'll lose a lot of time comparing the apples based on the color, size, form, and other parameters. Hash is something like a label that reflects the value of the object.



Now, let's move step bystep.

In *step 1*, we include the header file that contains definitions of the `hash` functions. In *step 2*, we declare our new `string` class that contains `str_`, which is the original value of string and `comparison_`, which is the computed `hash` value. Note the construction:

```
boost::hash<std::string>()(s)
```

Here, `boost::hash<std::string>` is a structure, a functional object just like `std::negate<>`. That is why we need the first parenthesis—we construct that functional object. The second parenthesis with `s` inside is a call to `std::size_t operator()(const std::string& s)`, which computes the `hash` value.

Now, take a look at *step 3*, where we define `operator==`:

```
return s1.comparison_ == s2.comparison_ && s1.str_ == s2.str_;
```

Take additional care about the second part of the expression. Hashing operation loses information, which means that there is possibly more than one string that produces exactly the same `hash` value. It means that if hashes mismatch, there is a 100% guarantee that the strings does not match; otherwise, we are required to compare strings using the traditional methods.

Well, it's time to compare numbers. If we measure the execution time using the default comparison method, it'll give us 819 milliseconds; however, our hashing comparison works almost two times faster and finishes in 475 milliseconds.

There’s more…

C++11 has the `hash` functional object; you may find it in the `<functional>` header in the `std::` namespace. Hashing in Boost and standard library is fast and reliable. It does not allocate additional memory and also does not have virtual functions.

You may specialize hashing for your own types. In Boost, it is done via specializing the `hash_value` function in the namespace of a custom type:

```
// Must be in the namespace of string_hash_fast class.
inline std::size_t hash_value(const string_hash_fast& v) {
    return v.comparison_;
}
```

This is different from standard library specialization of `std::hash`, where you are required to make template specialization of the `hash<>` structure in the `std::` namespace.

Hashing in Boost is defined for all the basic types (such as `int`, `float`, `double`, and `char`), for arrays, and for all the standard library containers, including `std::array`, `std::tuple`, and `std::type_index`. Some libraries also provide hash specializations, for example, the `Boost.Variant` library can hash any `boost::variant` classes.

See also

- Read the *Using unordered set and map* recipe in this chapter for more information about the hash functions' usage.
- The official documentation of `Boost.Functional/Hash` will tell you how to combine multiple hashes and provide more examples; read about it at http://boost.org/libs/functional/hash.

Using an unordered set and map

In the previous recipe, we saw how string comparison can be optimized using hashing. After reading it, the following question may arise: can we make a container that will cache hashed values to use faster comparison?

The answer is yes, and we can do much more. We may achieve almost constant search, insertion, and removal times for elements.

Getting ready

Basic knowledge about C++ and STL containers are required. Reading the previous recipe will also help.

How to do it…

This will be the simplest of all recipes:

1. All you need to do is just include the `<boost/unordered_map.hpp>` header, if you wish to use maps. If we wish to use sets, include the `<boost/unordered_set.hpp>` header.
2. Now, you are free to use `boost::unordered_map` instead of `std::map` and `boost::unordered_set` instead of `std::set`:

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

void example() {
    boost::unordered_set<std::string> strings;

    strings.insert("This");
    strings.insert("is");
    strings.insert("an");
    strings.insert("example");

    assert(strings.find("is") != strings.cend());
}
```

How it works…

Unordered containers store values and remember the hash of each value. Now if you wish to find a value in them, they will compute the hash of that value and search for that hash in the container. After hash is found, containers check for equality of the found value and the searched value. Then, the iterator to the value or to the end of container is returned.

Because the container may search for the constant width integral hash value, it may use some optimizations and algorithms suitable only for integers. Those algorithms guarantee constant search complexity O(1), when traditional `std::set` and `std::map` provide worse complexity O(log(N)), where *N* is the number of elements in the container. This leads us to a situation where the more the elements in traditional `std::set` or `std::map`, the slower it works. However, the performance of unordered containers does not depend on the element count.

Such good performance never comes free of cost. In unordered containers, values are unordered (you are not surprised, are you?). It means that we'll be elements of containers from `begin()` to `end()` will be the output, as follows:

```
template <class T>
void output_example() {
    T strings;

    strings.insert("CZ");
    strings.insert("CD");
    strings.insert("A");
    strings.insert("B");

    std::copy(
        strings.begin(),
        strings.end(),
        std::ostream_iterator<std::string>(std::cout, "  ")
    );
}
```

We'll get the following output for `std::set` and `boost::unordered_set`:

```
 boost::unordered_set<std::string> : B A CD CZ
 std::set<std::string> : A B CD CZ
```

So, how much does the performance differ? Usually, it depends on the implementation quality. I've got the following numbers:

```
For 100 elements:
Boost: map is 1.69954 slower than unordered map
Std: map is 1.54316 slower than unordered map

For 1000 elements:
Boost: map is 4.13714 slower than unordered map
Std: map is 2.12495 slower than unordered map

For 10000 elements:
Boost: map is 2.04475 slower than unordered map
Std: map is 2.23285 slower than unordered map

For 100000 elements:
Boost: map is 1.67128 slower than unordered map
Std: map is 1.68169 slower than unordered map
```

The performance was measured using for the following code block:

```
    T map;

    for (std::size_t ii = 0; ii < ii_max; ++ii) {
        map[s + boost::lexical_cast<std::string>(ii)] = ii;
    }

    // Asserting.
    for (std::size_t ii = 0; ii < ii_max; ++ii) {
        assert(map[s + boost::lexical_cast<std::string>(ii)] == ii);
    }
```

The code contains a lot of string constructions, so it is not 100% correct to measure the speedup using this test. It is here to show that unordered containers are usually faster than ordered ones.

Sometimes, a task might arise where we need to use a user-defined type in unordered containers:

```
struct my_type {
    int         val1_;
    std::string val2_;
};
```

To do that, we need to write a comparison operator for that type:

```
inline bool operator == (const my_type& v1, const my_type& v2) {
    return v1.val1_ == v2.val1_ && v1.val2_ == v2.val2_;
}
```

We also need to specialize the hashing function for that type. If the type consists of multiple fields, we usually just need to combine hashes of all the fields that participate in `equality comparisons`:

```
std::size_t hash_value(const my_type& v) {
    std::size_t ret = 0u;

    boost::hash_combine(ret, v.val1_);
    boost::hash_combine(ret, v.val2_);
    return ret;
}
```

It is highly recommended to combine hashes using the `boost::hash_combine` function.

There's more…

Multiversions of containers are also available, `boost::unordered_multiset` is defined in the `<boost/unordered_set.hpp>` header, and `boost::unordered_multimap` is defined in the `<boost/unordered_map.hpp>` header. Just like in the case of a standard library, multiversions of containers are capable of storing multiple equal key values.

All the unordered containers allow you to specify your own hashing functor, instead of the default `boost::hash`. They also allow you to specialize your own equal comparison functor, instead of the default `std::equal_to`.

C++11 has all the unordered containers of the Boost library. You may find them in the headers: `<unordered_set>` and `<unordered_map>`, in the `std::` namespace, instead of `boost::`. The Boost and the standard library versions may differ in performance, but must work in the same way. However, Boost's unordered containers are available even on C++03/C++98 compilers and make use of the rvalue reference emulation of `Boost.Move`, so you may use those containers for the move-only classes even on pre-C++11 compilers.

C++11 has no `hash_combine` function, so you have to write your own:

```
template <class T>
inline void hash_combine(std::size_t& seed, const T& v)
{
    std::hash<T> hasher;
    seed ^= hasher(v) + 0x9e3779b9 + (seed<<6) + (seed>>2);
}
```

Or just use `boost::hash_combine`.

Since Boost 1.64, unordered containers in Boost have the C++17's functionality for extracting and inserting nodes.

See also

- The recipe *Using C++11 move emulation* in Chapter 1, *Starting to Write Your Application*, for more details on rvalue reference emulation of `Boost.Move`
- More information about the unordered containers is available on the official site at http://boost.org/libs/unordered
- More information about combining hashes and computing hashes for ranges are available at http://boost.org/libs/functional/hash

Making a map, where value is also a key

Several times in a year, we need something that may store and index a pair of values. Moreover, we need to get the first part of the pair using the second, and get the second part using the first. Confused? Let me show you an example. We create a vocabulary class. When the users put values into it, the class must return identifiers, and when the users put identifiers into it, the class must return values.

To be more practical, users are putting login names in our vocabulary and wish to get the unique identifier out of it. They also wish to get all the logins for an identifier.

Let’s see how it can be implemented using Boost.

Getting ready

Basic knowledge about standard library and templates is required for this recipe.

How to do it…

This recipe is about the abilities of the `Boost.Bimap` library. Let's see how it can be used to implement this task:

1. We need the following includes:

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

2. Now, we are ready to make our vocabulary structure:

```
int main() {
    typedef boost::bimap<
        std::string,
        boost::bimaps::multiset_of<std::size_t>
    > name_id_type;

    name_id_type name_id;
```

3. It can be filled using the following syntax:

```
    // Inserting keys <-> values
    name_id.insert(name_id_type::value_type(
        "John Snow", 1
    ));

    name_id.insert(name_id_type::value_type(
        "Vasya Pupkin", 2
    ));

    name_id.insert(name_id_type::value_type(
        "Antony Polukhin", 3
    ));

    // Same person as "Antony Polukhin"
    name_id.insert(name_id_type::value_type(
        "Anton Polukhin", 3
    ));
```

4. We can work with the left part of it just like with a map:

```
    std::cout << "Left:\n";

    typedef name_id_type::left_const_iterator left_const_iterator;
    const left_const_iterator lend = name_id.left.end();

    for (left_const_iterator it = name_id.left.begin();
         it!= lend;
         ++it)
    {
        std::cout << it->first << " <=> " << it->second << '\n';
    }
```

5. The right part is almost the same as the left:

```
    std::cout << "\nRight:\n";

    typedef name_id_type::right_const_iterator right_const_iterator;
    const right_const_iterator rend = name_id.right.end();
```

```
            for (right_const_iterator it = name_id.right.begin();
                 it!= rend;
                 ++it)
            {
                std::cout << it->first << " <=> " << it->second << '\n';
            }
```

6. We also need to ensure that there is such a person in vocabulary:

```
            assert(
                name_id.find(name_id_type::value_type(
                    "Anton Polukhin", 3
                )) != name_id.end()
            );
        } /* end of main() */
```

That's it, now if we put all the code (except includes) inside `int main()`, we'll get the following output:

```
Left:
Anton Polukhin <=> 3
Antony Polukhin <=> 3
John Snow <=> 1
Vasya Pupkin <=> 2

Right:
1 <=> John Snow
2 <=> Vasya Pupkin
3 <=> Antony Polukhin
3 <=> Anton Polukhin
```

How it works…

In *step 2*, we define the `bimap` type:

```
    typedef boost::bimap<
        std::string,
        boost::bimaps::multiset_of<std::size_t>
    > name_id_type;
```

The first template parameter tells that the first key must have type `std::string`, and should work as `std::set`. The second template parameter tells that the second key must have type `std::size_t`. Multiple first keys may have a single second key value, just like in `std::multimap`.

We may specify the underlying behavior of `bimap` using classes from the `boost::bimaps::` namespace. We may use hash map as an underlying type for the first key:

```
#include <boost/bimap/unordered_set_of.hpp>
#include <boost/bimap/unordered_multiset_of.hpp>

typedef boost::bimap<
    boost::bimaps::unordered_set_of<std::string>,
    boost::bimaps::unordered_multiset_of<std::size_t>
> hash_name_id_type;
```

When we do not specify the behavior of the key and just specify its type, `Boost.Bimap` uses `boost::bimaps::set_of` as a default behavior. Just like in our example, we may try to express the following code using standard library:

```
#include <boost/bimap/set_of.hpp>

typedef boost::bimap<
    boost::bimaps::set_of<std::string>,
    boost::bimaps::multiset_of<std::size_t>
> name_id_type;
```

Using standard library it would look like a combination of the following two variables:

```
    std::map<std::string, std::size_t> key1;      // == name_id.left
    std::multimap<std::size_t, std::string> key2; // == name_id.right
```

As we can see from the preceding comments, a call to `name_id.left` (in *step 4*) returns a reference to something with an interface close to `std::map<std::string, std::size_t>`. A call to `name_id.right` from *step 5* returns something with an interface close to `std::multimap<std::size_t, std::string>`.

In *step 6*, we work with a whole `bimap`, searching for a pair of keys and making sure that they are in the container.

There’s more…

Unfortunately, C++17 has nothing close to `Boost.Bimap`. Here are some other bad news:

`Boost.Bimap` does not support rvalue references, and on some compilers, insane amount of warnings is shown. Refer to your compilers’ documentation to get the information about suppressing specific warnings.

Good news is that `Boost.Bimap` usually uses less memory than two standard library containers and makes searches as fast as standard library containers. It has no virtual function calls inside, but uses dynamic allocations.

See also

- The next recipe, *Using multi-index containers*, will give you more information about multi-indexing, and about the Boost library that can be used instead of `Boost.Bimap`
- Read the official documentation for more examples and information about `bimap` at http://boost.org/libs/bimap

Using multi-index containers

In the previous recipe, we made some kind of vocabulary, which is good when we need to work with pairs. But, what if we need much more advanced indexing? Let’s make a program that indexes persons:

```
struct person {
    std::size_t     id_;
    std::string     name_;
    unsigned int    height_;
    unsigned int    weight_;

    person(std::size_t id, const std::string& name,
                unsigned int height, unsigned int weight)
        : id_(id)
        , name_(name)
        , height_(height)
        , weight_(weight)
    {}
};

inline bool operator < (const person& p1, const person& p2) {
    return p1.name_ < p2.name_;
}
```

We will need a lot of indexes, for example, by name, ID, height, and weight.

Getting ready

Basic knowledge about standard library containers and unordered maps is required.

How to do it…

All the indexes can be constructed and managed by a single `Boost.Multiindex` container.

1. To do so, we need a lot of includes:

```
#include <iostream>
#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/member.hpp>
```

2. The hardest part is to construct `multi-index` type:

```
void example_main() {
    typedef boost::multi_index::multi_index_container<
        person,
        boost::multi_index::indexed_by<
            // names are unique
            boost::multi_index::ordered_unique<
                boost::multi_index::identity<person>
            >,

            // IDs are not unique, but we do not need them ordered
            boost::multi_index::hashed_non_unique<
                boost::multi_index::member<
                    person, std::size_t, &person::id_
                >
            >,

            // Height may not be unique, but must be sorted
            boost::multi_index::ordered_non_unique<
                boost::multi_index::member<
                    person, unsigned int, &person::height_
                >
            >,

            // Weight may not be unique, but must be sorted
            boost::multi_index::ordered_non_unique<
                boost::multi_index::member<
                    person, unsigned int, &person::weight_
                >
            >
        > // closing for `boost::multi_index::indexed_by<`
    > indexes_t;
```

3. Now, we may insert values into our `multi-index`:

```
    indexes_t persons;

    // Inserting values:
    persons.insert(person(1, "John Snow", 185, 80));
    persons.insert(person(2, "Vasya Pupkin", 165, 60));
    persons.insert(person(3, "Antony Polukhin", 183, 70));
    // Same person as "Antony Polukhin".
    persons.insert(person(3, "Anton Polukhin", 182, 70));
```

4. Let’s construct a function for printing the index content:

```
template <std::size_t IndexNo, class Indexes>
void print(const Indexes& persons) {
    std::cout << IndexNo << ":\n";
```

```
        typedef typename Indexes::template nth_index<
                IndexNo
        >::type::const_iterator const_iterator_t;

        for (const_iterator_t it = persons.template get<IndexNo>().begin(),
             iend = persons.template get<IndexNo>().end();
             it != iend;
             ++it)
        {
            const person& v = *it;
            std::cout
                << v.name_ << ", "
                << v.id_ << ", "
                << v.height_ << ", "
                << v.weight_ << '\n'
            ;
        }

        std::cout << '\n';
    }
```

5. Print all the indexes as follows:

```
        print<0>(persons);
        print<1>(persons);
        print<2>(persons);
        print<3>(persons);
```

6. Some code from the previous recipe can also be used:

```
        assert(persons.get<1>().find(2)->name_ == "Vasya Pupkin");
        assert(
            persons.find(person(
                77, "Anton Polukhin", 0, 0
            )) != persons.end()
        );

        // Won't compile:
        //assert(persons.get<0>().find("John Snow")->id_ == 1);
```

Now if we run our example, it will output the content of indexes:

```
0:
Anton Polukhin, 3, 182, 70
Antony Polukhin, 3, 183, 70
John Snow, 1, 185, 80
Vasya Pupkin, 2, 165, 60

1:
John Snow, 1, 185, 80
Vasya Pupkin, 2, 165, 60
Anton Polukhin, 3, 182, 70
Antony Polukhin, 3, 183, 70

2:
Vasya Pupkin, 2, 165, 60
Anton Polukhin, 3, 182, 70
Antony Polukhin, 3, 183, 70
John Snow, 1, 185, 80

3:
Vasya Pupkin, 2, 165, 60
Antony Polukhin, 3, 183, 70
Anton Polukhin, 3, 182, 70
John Snow, 1, 185, 80
```

How it works...

The hardest part here is a construction of multi-index type using `boost::multi_index::multi_index_container`. The first template parameter is a class that we are going to index. In our case, it is `person`. The second parameter is a type `boost::multi_index::indexed_by`, all the indexes must be described as a template parameter of that class.

Now, let's take a look at the first index description:

```
            boost::multi_index::ordered_unique<
                boost::multi_index::identity<person>
            >
```

The usage of the `boost::multi_index::ordered_unique` class means that the index must work like `std::set` and have all of its members. The `boost::multi_index::identity<person>` class means that the index must use the `operator <` of a `person` class for orderings.

The next table shows the relation between the `Boost.MultiIndex` types and the **STL containers**:

The `Boost.MultiIndex` types	STL containers
`boost::multi_index::ordered_unique`	`std::set`
`boost::multi_index::ordered_non_unique`	`std::multiset`
`boost::multi_index::hashed_unique`	`std::unordered_set`
`boost::multi_index::hashed_non_unique`	`std::unordered_mutiset`
`boost::multi_index::sequenced`	`std::list`

Take a look at the second index:

```
            boost::multi_index::hashed_non_unique<
                boost::multi_index::member<
                    person, std::size_t, &person::id_
                >
            >
```

The `boost::multi_index::hashed_non_unique` type means that the index works like `std::set`, and `boost::multi_index::member<person, std::size_t, &person::id_>` means that the index must apply the hash function only to a single member field of the person structure to `person::id_`.

The remaining indexes won't be a trouble now; so let's take a look at the usage of indexes in the `print` function instead. Getting the type of iterator for a specific index is done using the following code:

```
    typedef typename Indexes::template nth_index<
            IndexNo
    >::type::const_iterator const_iterator_t;
```

This looks slightly overcomplicated because `Indexes` is a template parameter. The example would be simpler, if we could write this code in the scope of `indexes_t`:

```
typedef indexes_t::nth_index<0>::type::const_iterator const_iterator_t;
```

The `nth_index` member metafunction takes a zero-based number of index to use. In our example, index 1 is index of IDs, index 2 is index of heights, and so on.

Now, let’s take a look at how to use `const_iterator_t`:

```
for (const_iterator_t it = persons.template get<IndexNo>().begin(),
         iend = persons.template get<IndexNo>().end();
         it != iend;
         ++it)
    {
        const person& v = *it;
        // ...
```

This can also be simplified for `indexes_t` being in scope:

```
    for (const_iterator_t it = persons.get<0>().begin(),
         iend = persons.get<0>().end();
         it != iend;
         ++it)
    {
        const person& v = *it;
        // ...
```

The function `get<indexNo>()` returns index. We may use that index almost like an STL container.

There's more…

C++17 has no multi-index library. The `Boost.MultiIndex` is a fast library that uses no virtual functions. The official documentation of `Boost.MultiIndex` contains performance and memory usage measures, showing that this library in most cases uses less memory than standard library based handwritten code. Unfortunately, `boost::multi_index::multi_index_container` does not support C++11 features and also has no rvalue references emulation using `Boost.Move`.

See also

The official documentation of `Boost.MultiIndex` contains tutorials, performance measures, examples, and other `Boost.Multiindex` libraries' description of useful features. Read about it at http://boost.org/libs/multi_index.

Getting benefits of a single linked list and memory pool

Nowadays, we usually use `std::vector` when we need nonassociative and nonordered containers. This is recommended by *Andrei Alexandrescu* and *Herb Sutter* in the book *C++ Coding Standards*. Even those users who did not read the book usually use `std::vector`. Why? Well, `std::list` is slower and uses much more resources than `std::vector`. The `std::deque` container is very close to `std::vector`, but does not store values continuously.

If we need a container where erasing and inserting elements does not invalidate iterators, then we are forced to choose a slow `std::list`.

But wait, we may assemble a better solution using Boost!

Getting ready

Good knowledge about standard library containers is required to understand the introduction part. After that, only basic knowledge of C++ and standard library containers is required.

How to do it…

In this recipe, we'll be using two Boost libraries at the same time: `Boost.Pool` and a single linked list from `Boost.Container`.

1. We need the following headers:

```
#include <boost/pool/pool_alloc.hpp>
#include <boost/container/slist.hpp>
#include <cassert>
```

2. Now, we need to describe the type of our list. This can be done as shown in the following code:

```
typedef boost::fast_pool_allocator<int> allocator_t;
typedef boost::container::slist<int, allocator_t> slist_t;
```

3. We can work with our single linked list like with `std::list`:

```
template <class ListT>
void test_lists() {
    typedef ListT list_t;

    // Inserting 1000000 zeros.
    list_t  list(1000000, 0);

    for (int i = 0; i < 1000; ++i) {
        list.insert(list.begin(), i);
    }

    // Searching for some value.
    typedef typename list_t::iterator iterator;
    iterator it = std::find(list.begin(), list.end(), 777);
    assert(it != list.end());

    // Erasing some values.
    for (int i = 0; i < 100; ++i) {
        list.pop_front();
    }

    // Iterator is still valid and points to the same value.
    assert(it != list.end());
    assert(*it == 777);

    // Inserting more values
    for (int i = -100; i < 10; ++i) {
        list.insert(list.begin(), i);
    }

    // Iterator is still valid and points to the same value
    assert(it != list.end());
    assert(*it == 777);
}

void test_slist() {
    test_lists<slist_t>();
}

void test_list() {
    test_lists<std::list<int> >();
}
```

4. Some list-specific functions:

```
        void list_specific(slist_t& list, slist_t::iterator it) {
            typedef slist_t::iterator iterator;

            // Erasing element 776
            assert( *(++iterator(it)) == 776);
            assert(*it == 777);

            list.erase_after(it);

            assert(*it == 777);
            assert( *(++iterator(it)) == 775);
```

5. Memory must be freed using the following code:

```
            // Freeing memory: slist rebinds allocator_t and allocates
            // nodes of the slist, not just ints.

            boost::singleton_pool<
                boost::fast_pool_allocator_tag,
                sizeof(slist_t::stored_allocator_type::value_type)
            >::release_memory();
        } // end of list_specific function
```

How it works…

When we are using `std::list`, we may notice a slowdown because each node of the list needs separate allocation. It means that usually when we insert 10 elements into `std::list`, the container calls `new` 10 times. Also, the allocated nodes usually are located randomly in memory, which is not CPU cache friendly.

That is why we used Boost `::fast_pool_allocator<int>` from `Boost.Pool`. This allocator tries to allocate bigger blocks of memory, so that at a later stage, multiple nodes could be constructed without multiple calls to `new`.

The `Boost.Pool` library has a drawback—it uses memory for internal needs. Usually, additional `sizeof(void*)` is used per element. To workaround that issue, we are using a single linked list from `Boost.Containers`.

The `boost::container::slist` class is more compact, but its iterators can iterate only forward. *Step 3* is simple for those readers who are aware of standard library containers, so we move to *step 4* to see some `boost::container::slist` specific features. Since a single linked list iterator could iterate only forward, traditional algorithms of insertion and deletion take linear time O(N). That's because when we are erasing or inserting, the previous element of the list must be modified. To workaround that issue, the single linked list has methods `erase_after` and `insert_after` that work for a constant time O(1). These methods insert or erase elements right after the current position of the iterator.

> *However, erasing and inserting values at the beginning of a single linked lists makes no big difference.*

Take a careful look at the following code:

```
boost::singleton_pool<
    boost::fast_pool_allocator_tag,
    sizeof(slist_t::stored_allocator_type::value_type)
>::release_memory();
```

It is required because `boost::fast_pool_allocator` does not free memory, so we must do it by hand. The *Doing something at scope exit* recipe from Chapter 2, *Managing Resources*, may be a help in freeing `Boost.Pool`.

Let's take a look at the execution times to feel the difference:

```
$ TIME="Runtime=%E RAM=%MKB" time ./07_slist_and_pool l
std::list: Runtime=0:00.08 RAM=34224KB

$ TIME="Runtime=%E RAM=%MKB" time ./07_slist_and_pool s
slist_t:   Runtime=0:00.04 RAM=19640KB
```

As we may see, `slist_t` uses half the memory, and is twice as fast compared to the `std::list` class.

There's more…

The `Boost.Container` library actually has an out-of-the-box solution, called `boost::container::stable_vector`. The latter allows random access to the elements, has random access iterators, but has most of the performance and memory usage drawbacks of `std::list`.

C++11 has `std::forward_list`, which is very close to `boost::containers::slist`. It also has the `*_after` methods, but has no `size()` method. C++11 and Boost versions of single linked list have the same performance and neither of them have virtual functions. However, the Boosts version is also usable on C++03 compilers, and even has support for rvalue references emulation via `Boost.Move`.

The `boost::fast_pool_allocator` is not in C++17. However, C++17 has a better solution! The header `<memory_resource>` contains useful stuff to work with polymorphic allocator, and in there you can find `std::pmr::synchronized_pool_resource`, `std::pmr::unsynchronized_pool_resource`, and `std::pmr::monotonic_buffer_resource`. Experiment with those to achieve even better performance.

Guessing why `boost::fast_pool_allocator` does not free the memory by itself? That's because C++03 has no stateful allocators, so the containers are not copying and storing allocators. That makes it impossible to implement a `boost::fast_pool_allocator` function that deallocates memory by itself.

See also

- The official documentation of `Boost.Pool` contains more examples and classes to work with memory pools. Follow the link http://boost.org/libs/pool to read about it.
- The *Using flat associative containers* recipe will introduce you to some more classes from `Boost.Container`. You can also read the official documentation of `Boost.Container` at http://boost.org/libs/container to study that library by yourself or to get full reference documentation of its classes.
- *Vector vs List*, and other interesting topics from *Bjarne Stroustrup*, the inventor of C++ programming language, can be found at http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style site.

Using flat associative containers

After reading the previous recipe, some of the readers may start using fast pool allocators everywhere; especially, for `std::set` and `std::map`. Well, I'm not going to stop you from doing that, but at least let's take a look at an alternative: flat associative containers. These containers are implemented on top of the traditional vector container and store the values ordered.

Getting ready

The basic knowledge about standard library associative containers is required.

How to do it…

The flat containers are part of the `Boost.Container` library. We already saw how to use some of its containers in the previous recipes. In this recipe, we'll be using a `flat_set` associative container:

1. We'll need to include only a single header file:

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

2. After that, we are free to construct the flat container and experiment with it:

```
#include <algorithm>
#include <cassert>

int main() {
    boost::container::flat_set<int> set;
```

3. Reserving space for elements:

```
    set.reserve(4096);
```

4. Filling the container:

```
    for (int i = 0; i < 4000; ++i) {
        set.insert(i);
    }
```

5. Now, we can work with it just like with `std::set`:

```
    // 5.1
    assert(set.lower_bound(500) - set.lower_bound(100) == 400);

    // 5.2
    set.erase(0);

    // 5.3
    set.erase(5000);

    // 5.4
    assert(std::lower_bound(set.cbegin(), set.cend(), 900000) == set.cend());

    // 5.5
    assert(
        set.lower_bound(100) + 400
        ==
        set.find(500)
    );
} // end of main() function
```

How it works…

Steps 1 and *2* are simple, but *step 3* requires attention. It is one of the most important steps while working with flat associative containers and `std::vector`.

The `boost::container::flat_set` class stores its values ordered in vector, which means that any insertion or deletion of elements not on the end on the container takes liner time O(N), just like in case of `std::vector`. This is a necessary evil. But for that, we gain almost three times less memory usage per element, more processor cache friendly storage, and random access iterators. Take a look at *step 5*, `5.1`, where we were getting the distance between two iterators returned by calls to the `lower_bound` member functions. Getting distance with flat set takes constant time O(1), while the same operation on iterators of `std::set` takes linear time O(N). In case of `5.1`, getting the distance using `std::set` would be 400 times slower than getting the distance for flat set containers.

Back to *step 3*. Without reserving memory, insertion of elements may become at times slower and less memory efficient. The `std::vector` class allocates the required chunk of memory and then in-place construct elements on that chunk. When we insert some element without reserving the memory, there is a chance that there is no free space remaining on the preallocated chunk of memory, so `std::vector` allocates a bigger chunk of memory . After that, `std::vector` copies or moves elements from the first chunk to the second, deletes elements of the first chunk, and deallocates the first chunk. Only after that, insertion occurs. Such copying and deallocation may occur multiple times during insertions, dramatically reducing the speed.

> *If you know the count of elements that `std::vector` or any flat container must store, reserve the space for those elements before insertion. This speeds up the program in most cases!*

Step 4 is simple, we are inserting elements here. Note that we are inserting ordered elements. This is not required, but recommended to speed up insertion. Inserting elements at the end of `std::vector` is much more cheaper than in the middle or at the beginning.

In *step 5*, `5.2` and `5.3` do not differ much, except in their execution speed. Rules for erasing elements are pretty much the same as for inserting them. See the preceding paragraph for explanations.

> *May be I'm telling you simple things about containers, but I saw some very popular products that use features of C++11, have insane amount of optimizations and lame usage of standard library containers, especially `std::vector`.*

In *step 5*, `5.4` shows you that the `std::lower_bound` function works faster with `boost::container::flat_set` than with `std::set`, because of random access iterators.

In *step 5*, `5.5` also shows you the benefit of random access iterators.

> *We did not use the `std::find` function here. This is because that function takes liner time O(N), while the member `find` functions take logarithmic time*

$O(log(N))$.

There’s more…

When should we use flat containers, and when should we use usual ones? Well, it’s up to you, but here is a list of differences from the official documentation of `Boost.Container` that will help you to decide:

- A faster lookup than standard associative containers
- Much faster iteration than standard associative containers
- Less memory consumption for small objects (and for big objects if `shrink_to_fit` is used)
- Improved cache performance (data is stored in contiguous memory)
- Nonstable iterators (iterators are invalidated when inserting and erasing elements)
- Non-copyable and non-movable value types can’t be stored
- Weaker exception safety than standard associative containers (copy/move constructors can throw an exception when shifting values in erasures and insertions)
- Slower insertion and erasure than standard associative containers (specially for non-movable types)

C++17 unfortunately has no flat containers. Flat containers from Boost are fast, have a lot of optimizations, and do not use virtual functions. Classes from `Boost.Containers` have support of rvalue reference emulation via `Boost.Move`, so you are free to use them even on C++03 compilers.

See also

- Refer to the *Getting benefits of single linked list and memory pool* recipe for more information about `Boost.Container`.
- The recipe *Using C++11 move emulation* in Chapter 1, *Starting to Write Your Application*, will give you the basics of emulation rvalue references on C++03 compatible compilers.
- The official documentation of `Boost.Container` contains a lot of useful information about `Boost.Container` and full reference of each class. Read about it at http://boost.org/libs/container.

Gathering Platform and Compiler Information

In this chapter, we will cover:

- Detecting an OS and compiler
- Detecting int128 support
- Detecting and bypassing disabled RTTI
- Writing metafunctions using simpler methods
- Reducing code size and increasing the performance of user-defined types (UDTs) in C++11
- The portable way to export and import functions and classes
- Detecting the Boost version and getting latest features

Introduction

Different projects and companies have different coding requirements. Some of them forbid exceptions or RTTI, while some forbid C++11. If you are willing to write portable code that can be used by a wide range of projects, this chapter is for you.

Want to make your code as fast as possible and use the latest C++ features? You’ll definitely need a tool for detecting compiler features.

Some compilers have unique features that may greatly simplify your life. If you are targeting a single compiler, you can save many hours and use those features. No need to implement their analogs from scratch!

This chapter is devoted to different helper macros used to detect compiler, platform, and Boost features. These macro are widely used across Boost libraries and are essential for writing portable code that is able to work with any compiler flags.

Detecting an OS and compiler

I’m guessing you’ve seen a bunch of ugly macros to detect the compiler on which the code is compiled. Something like this is a typical practice in C word:

```
#include <something_that_defines_macros>
#if !defined(__clang__) \
    && !defined(__ICC) \
    && !defined(__INTEL_COMPILER) \
    && (defined(__GNUC__) || defined(__GNUG__))

// GCC specific

#endif
```

Now, try to come up with a good macro to detect the GCC compiler. Try to make that macro usage as short as possible.

Take a look at the following recipe to verify your guess.

Getting ready

Only basic knowledge of C++ is required.

How to do it…

The recipe is simple and consists of a single a header and a single macro.

1. The header:

```
#include <boost/predef/compiler.h>
```

2. The macro:

```
#if BOOST_COMP_GNUC

// GCC specific

#endif
```

How it works…

The header `<boost/predef/compiler.h>` knows all the possible compilers and has a macro for each of those. So if the current compiler is GCC, then macro `BOOST_COMP_GNUC` is defined to `1` and all the other macros for other compilers are defined to `0`. If we are not on a GCC compiler, then the `BOOST_COMP_GNUC` macro is defined to `0`.

Thanks to this approach, you do not need to check for the macro itself being defined:

```
#if defined(BOOST_COMP_GNUC) // Wrong!

// GCC specific

#endif
```

Macros of the `Boost.Predef` library are always defined, and that saves you from typing `defined()` or `def` in `#ifdef`.

There’s more…

The `Boost.Predef` library also has macros for detecting OS, architecture, standard library implementation, and some hardware abilities. Approach with macros that are always defined; this allows you to write complex expressions much shorter:

```
#include <boost/predef/os.h>
#include <boost/predef/compiler.h>

#if BOOST_COMP_GNUC && BOOST_OS_LINUX && !BOOST_OS_ANDROID

// Do something for non Android Linux.

#endif
```

Now, the best part. The `Boost.Predef` library is usable on C, C++, and Objective-C compilers. If you like it, use it in your non C++ projects.

C++17 has no `Boost.Predef` library functionality.

See also

- Read the official documentation of `Boost.Predef` for more information about its abilities at http://boost.org/libs/predef
- The next recipe will introduce you to the `Boost.Config` library, that is much order, slightly less beautiful, but much more functional

Detecting int128 support

Some compilers have support for extended arithmetic types such as 128-bit floats or integers. Let’s take a quick glance at how to use them using Boost.

We’ll be creating a method that accepts three parameters and returns the multiplied value of those methods. If compiler supports 128-bit integers, then we use them. If compiler supports `long long`, then we use it; otherwise, we need to issue a compile-time error.

Getting ready

Only the basic knowledge of C++ is required.

How to do it…

What do we need to work with 128-bit integers? Macros that show that they are available and a few `typedefs` to have portable type names across platforms.

1. Include a header:

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

2. Now, we need to detect int128 support:

```
#ifdef BOOST_HAS_INT128
```

3. Add some `typedefs` and implement the method as follows:

```
typedef boost::int128_type int_t;
typedef boost::uint128_type uint_t;

inline int_t mul(int_t v1, int_t v2, int_t v3) {
    return v1 * v2 * v3;
}
```

4. For compilers that do not support the int128 type and have no `long long`, we may produce compile time error:

```
#else // #ifdef BOOST_HAS_INT128

#ifdef BOOST_NO_LONG_LONG
#error "This code requires at least int64_t support"
#endif
```

5. Now, we need to provide some implementation for compilers without int128 support using `int64`:

```
struct int_t { boost::long_long_type hi, lo; };
struct uint_t { boost::ulong_long_type hi, lo; };

inline int_t mul(int_t v1, int_t v2, int_t v3) {
    // Some hand written math.
    // ...
}

#endif // #ifdef BOOST_HAS_INT128
```

How it works…

The header `<boost/config.hpp>` contains a lot of macros to describe compiler and platform features. In this example, we used `BOOST_HAS_INT128` to detect support of 128-bit integers and `BOOST_NO_LONG_LONG` to detect support of 64-bit integers.

As we may see from the example, Boost has `typedefs` for 64-bit signed and unsigned integers:

```
boost::long_long_type
boost::ulong_long_type
```

It also has `typedefs` for 128-bit signed and unsigned integers:

```
boost::int128_type
boost::uint128_type
```

There’s more…

C++11 has support of 64-bit types via the `long long int` and `unsigned long long int` built-in types. Unfortunately, not all compilers support C++11, so `BOOST_NO_LONG_LONG` may be useful for you.

128-bit integers are not a part of C++17, so `typedefs` and macros from Boost are one of the ways to write portable code.

There’s an ongoing work in C++ standardization committee on adding integers of compile-time specified width. When that work will be finished, you would be able to create 128-bit, 512-bit, and even 8388608-bit (1 MB big) integers.

See also

- Read the recipe *Detecting and bypassing disabled RTTI* for more information about `Boost.Config`.
- Read the official documentation of `Boost.Config` at http://boost.org/libs/config for more information about its abilities.
- There is a library in Boost that allows constructing types of unlimited precision. Follow the link http://boost.org/libs/multiprecision and take a look at the `Boost.Multiprecision` library.

Detecting and bypassing disabled RTTI

Some companies and libraries have specific requirements for their C++ code, such as successful compilation without RTTI.

In this small recipe, we'll not just detect disabled RTTI, but also write a Boost like library from scratch that stores information about types, and compares types at runtime, even without `typeid`.

Getting ready

Basic knowledge of C++ RTTI usage is required for this recipe.

How to do it…

Detecting disabled RTTI, storing information about types, and comparing types at runtime are tricks that are widely used across Boost libraries.

1. To do this, we first need to include the following header:

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

2. Let's first look at the situation where RTTI is enabled and the C++11 `std::type_index` class is available:

```
#if !defined(BOOST_NO_RTTI) \
    && !defined(BOOST_NO_CXX11_HDR_TYPEINDEX)

#include <typeindex>
using std::type_index;

template <class T>
type_index type_id() {
    return typeid(T);
}
```

3. Otherwise, we need to construct our own `type_index` class:

```
#else

#include <cstring>
#include <iosfwd> // std::basic_ostream
#include <boost/current_function.hpp>

struct type_index {
    const char * name_;

    explicit type_index(const char* name)
        : name_(name)
    {}

    const char* name() const { return name_; }
};

inline bool operator == (type_index v1, type_index v2) {
    return !std::strcmp(v1.name_, v2.name_);
}

inline bool operator != (type_index v1, type_index v2) {
    return !(v1 == v2);
}
```

4. The final step is to define the `type_id` function:

```
template <class T>
inline type_index type_id() {
    return type_index(BOOST_CURRENT_FUNCTION);
}

#endif
```

5. Now, we can compare types:

```
#include <cassert>
```

```
int main() {
    assert(type_id<unsigned int>() == type_id<unsigned>());
    assert(type_id<double>() != type_id<long double>());
}
```

How it works…

The macro `BOOST_NO_RTTI` is be defined if RTTI is disabled, and the macro `BOOST_NO_CXX11_HDR_TYPEINDEX` is be defined when the compiler has no `<typeindex>` header and no `std::type_index` class.

The handwritten `type_index` structure from *step 3* of the previous section only holds the pointer to some string; nothing really interesting here.

Take a look at the `BOOST_CURRENT_FUNCTION` macro. It returns the full name of the current function, including template parameters, arguments, and the return type.
For example, `type_id<double>()` is represented as follows:

```
type_index type_id() [with T = double]
```

So, for any other type, `BOOST_CURRENT_FUNCTION` returns a different string, and that's why the `type_index` variable from the example does not compare equal to it.

Congratulations! We've just reinvented most of the `Boost.TypeIndex` library functionality. Remove all the code from *steps 1 to 4* and slightly change the code in *step 5* to use the `Boost.TypeIndex` library:

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

void test() {
    using boost::typeindex::type_id;

    assert(type_id<unsigned int>() == type_id<unsigned>());
    assert(type_id<double>() != type_id<long double>());
}
```

There’s more…

Of course `Boost.TypeIndex` is slightly more than that; it allows you to get human readable type name in a platform independent way, works around platform-related issues, allows to invent your own RTTI implementation, to have a constexpr RTTI, and other stuff.

Different compilers have different macros for getting a full function name. Using macros from Boost is the most portable solution. The `BOOST_CURRENT_FUNCTION` macro returns the name at compile time, so it implies minimal runtime penalty.

C++11 has a `__func__` magic identifier that is evaluated to the name of the current function. However, result of `__func__` is only the function name, while `BOOST_CURRENT_FUNCTION` tries hard to also show function parameters, including template ones.

See also

- Read the upcoming recipes for more information on `Boost.Config`
- Browse to http://github.com/boostorg/type_index to view the source codes of the `Boost.TypeIndex` library
- Read the official documentation of `Boost.Config` at http://boost.org/libs/config
- Read the official documentation of the `Boost.TypeIndex` library at http://boost.org/libs/type_index
- Recipe *Getting human readable type name* in Chapter 01, *Starting to Write Your Application* will introduce you to some of the other capabilities of the `Boost.TypeIndex`

Writing metafunctions using simpler methods

Chapter 4, *Compile-time Tricks*, and Chapter 8, *Metaprogramming*, were devoted to metaprogramming. If you were trying to use techniques from those chapters, you may have noticed that writing a metafunction can take a lot of time. So, it may be a good idea to experiment with metafunctions using more user-friendly methods, such as C++11 `constexpr`, before writing a portable implementation.

In this recipe, we'll take a look at how to detect `constexpr` support.

Getting ready

The `constexpr` functions are functions that can be evaluated at compile-time. That is all we need to know for this recipe.

How to do it…

Let’s see how we can detect compiler support for the `constexpr` feature:

1. Just like in other recipes from this chapter, we start with the following header:

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

2. Write the `constexpr` function:

```
#if !defined(BOOST_NO_CXX11_CONSTEXPR) \
    && !defined(BOOST_NO_CXX11_HDR_ARRAY)

template <class T>
constexpr int get_size(const T& val) {
    return val.size() * sizeof(typename T::value_type);
}
```

3. Let’s print an error if C++11 features are missing:

```
#else
#error "This code requires C++11 constexpr and std::array"
#endif
```

4. That’s it. Now, we are free to write code such as the following:

```
#include <array>

int main() {
    std::array<short, 5> arr;
    static_assert(get_size(arr) == 5 * sizeof(short), "");

    unsigned char data[get_size(arr)];
}
```

How it works…

The `BOOST_NO_CXX11_CONSTEXPR` macro is defined when C++11 `constexpr` is available.

The `constexpr` keyword tells the compiler that the function can be evaluated at compile time if all the inputs for that function are compile-time constants. C++11 imposes a lot of limitations on what a `constexpr` function can do. C++14 removed some of the limitations.

The `BOOST_NO_CXX11_HDR_ARRAY` macro is defined when the C++11 `std::array` class and the `<array>` header are available.

There's more…

However, there are other usable and interesting macros for `constexpr` too, as follows:

- The `BOOST_CONSTEXPR` macro expands to `constexpr` or does not expand
- The `BOOST_CONSTEXPR_OR_CONST` macro expands to `constexpr` or `const`
- The `BOOST_STATIC_CONSTEXPR` macro is the same as `static BOOST_CONSTEXPR_OR_CONST`

Using those macros, it is possible to write code that takes advantage of C++11 constant expression features if they are available:

```
template <class T, T Value>
struct integral_constant {
    BOOST_STATIC_CONSTEXPR T value = Value;

    BOOST_CONSTEXPR operator T() const {
        return this->value;
    }
};
```

Now, we can use `integral_constant` as shown in the following code:

```
char array[integral_constant<int, 10>()];
```

In the example, `BOOST_CONSTEXPR operator T()` is called to get the array size.

The C++11 constant expressions may improve compilation speed and diagnostic information in case of error. It's a good feature to use. If your function requires **relaxed constexpr** from C++14, then you may use `BOOST_CXX14_CONSTEXPR` macro. It expands to `constexpr` only if relaxed constexpr is available and expands to nothing otherwise.

See also

- More information about `constexpr` usage can be read at http://en.cppreference.com/w/cpp/language/constexpr
- Read the official documentation of `Boost.Config` for more information about macros at http://boost.org/libs/config

Reducing code size and increasing performance of user-defined types (UDTs) in C++11

C++11 has very specific logic when **user-defined types** (**UDTs**) are used in standard library containers. Some containers use move assignment and move construction only if the move constructor does not throw exceptions or there is no copy constructor.

Let’s see how we can ensure the compiler that the out class `move_nothrow` has a non-throwing `move` assignment operator and a non-throwing `move` constructor.

Getting ready

Basic knowledge of C++11 rvalue references is required for this recipe. Knowledge of standard library containers will also serve you well.

How to do it…

Let's take a look at how we can improve our C++ classes using Boost.

1. All we need to do is mark the `move_nothrow` assignment operator and `move_nothrow` constructor with the `BOOST_NOEXCEPT` macro:

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

class move_nothrow {
    // Some class class members go here.
    // ...

public:
    move_nothrow() BOOST_NOEXCEPT;
    move_nothrow(move_nothrow&&) BOOST_NOEXCEPT
        // Members initialization goes here.
        // ...
    {}

    move_nothrow& operator=(move_nothrow&&) BOOST_NOEXCEPT {
        // Implementation goes here.
        // ...
        return *this;
    }

    move_nothrow(const move_nothrow&);
    move_nothrow& operator=(const move_nothrow&);
};
```

2. Now, we may use the class with `std::vector` in C++11 without any modifications:

```
#include <vector>

int main() {
    std::vector<move_nothrow> v(10);
    v.push_back(move_nothrow());
}
```

3. If we remove `BOOST_NOEXCEPT` from the `move` constructor, we'll get the following error because we provided no definition for the copy constructor:

```
undefined reference to `move_nothrow::move_nothrow(move_nothrow
 const&)
```

How it works…

The `BOOST_NOEXCEPT` macro expands to `noexcept` on compilers that support it. The standard library containers use type traits to detect if the constructor throws an exception or not. Type traits make their decision mainly based on `noexcept` specifiers.

Why do we get an error without `BOOST_NOEXCEPT`? Compiler’s type trait returns that `move_nothrow` throws, so `std::vector` tries to use the copy constructor of `move_nothrow`, which is not defined.

There’s more…

The BOOST_NOEXCEPT macro also reduces binary size irrespective of whether the definition of the noexcept function or method is in a separate source file or not.

```
// In header file.
int foo() BOOST_NOEXCEPT;

// In source file.
int foo() BOOST_NOEXCEPT {
    return 0;
}
```

That’s because in the latter case, the compiler knows that the function does not throw exceptions and so there is no need to generate code that handles them.

If a function marked as noexcept *does throw an exception, your program will terminate without calling destructors for the constructed objects.*

See also

- A document describing why `move` constructors are allowed to throw exceptions and how containers must move objects is available at http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3050.html
- Read the official documentation of `Boost.Config` for more examples of `BOOST_NOEXCEPT` such as macros existing in Boost at http://boost.org/libs/config

The portable way to export and import functions and classes

Almost all modern languages have the ability to make libraries, a collection of classes, and methods that have a well-defined interface. C++ is no exception to this rule. We have two types of libraries: runtime (also called shared or dynamic) and static. But, writing libraries is not a simple task in C++. Different platforms have different methods for describing which symbols must be exported from the shared library.

Let’s take a look at how to manage symbol visibility in a portable way using Boost.

Getting ready

Experience in creating dynamic and static libraries may be useful in this recipe.

How to do it…

The code for this recipe consists of two parts. The first part is the library itself. The second part is the code that uses that library. Both parts use the same header, in which the library methods are declared. Managing symbol visibility in a portable way using Boost is simple and can be done using the following steps:

1. In the header file, we need definitions from the following header file:

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

2. The following code must also be added to the header file:

```
#if defined(MY_LIBRARY_LINK_DYNAMIC)
#   if defined(MY_LIBRARY_COMPILATION)
#       define MY_LIBRARY_API BOOST_SYMBOL_EXPORT
#   else
#       define MY_LIBRARY_API BOOST_SYMBOL_IMPORT
#   endif
#else
#   define MY_LIBRARY_API
#endif
```

3. Now, all the declarations must use the `MY_LIBRARY_API` macro:

```
int MY_LIBRARY_API foo();

class MY_LIBRARY_API bar {
public:
    /* ... */
    int meow() const;
};
```

4. Exceptions must be declared with `BOOST_SYMBOL_VISIBLE`; otherwise, they can be caught only using `catch(...)` in the code that uses the library:

```
#include <stdexcept>

struct BOOST_SYMBOL_VISIBLE bar_exception
    : public std::exception
{};
```

5. Library source files must include the header file:

```
#define MY_LIBRARY_COMPILATION
#include "my_library.hpp"
```

6. Definitions of methods must also be in the source files of the library:

```
int MY_LIBRARY_API foo() {
    // Implementation goes here.
    // ...
    return 0;
}

int bar::meow() const {
    throw bar_exception();
}
```

7. Now, we can use the library as shown in the following code:

```
#include "../06_A_my_library/my_library.hpp"
#include <cassert>

int main() {
    assert(foo() == 0);
    bar b;
    try {
        b.meow();
        assert(false);
    } catch (const bar_exception&) {}
}
```

How it works…

All the work is done in *step 2*. There, we are defining the macro `MY_LIBRARY_API`, which we apply to classes and methods that we wish to export from our library. In *step 2*, we check for `MY_LIBRARY_LINK_DYNAMIC`. If it is not defined, we are building a static library and there is no need to define `MY_LIBRARY_API`.

> *The developer must take care of `MY_LIBRARY_LINK_DYNAMIC`! It will not define itself. If we are making a dynamic library, we need to make our build system to define it,*

If `MY_LIBRARY_LINK_DYNAMIC` is defined, we are building a runtime library, and that's where the workarounds start. You, as the developer, must tell the compiler that we are now exporting function to the user. The user must tell the compiler that he/she is importing methods from the library. To have a single header file for both the import and export library, we use the following code:

```
#if defined(MY_LIBRARY_COMPILATION)
#    define MY_LIBRARY_API BOOST_SYMBOL_EXPORT
#else
#    define MY_LIBRARY_API BOOST_SYMBOL_IMPORT
#endif
```

When exporting the library (or, in other words, compiling it), we must define `MY_LIBRARY_COMPILATION`. This leads to `MY_LIBRARY_API` being defined to `BOOST_SYMBOL_EXPORT`. For example, see *step 5*, where we defined `MY_LIBRARY_COMPILATION` before including `my_library.hpp`. If `MY_LIBRARY_COMPILATION` is not defined, the header is included by the user, who doesn't know anything about that macro. And, if the header is included by the user, the symbols must be imported from the library.

The `BOOST_SYMBOL_VISIBLE` macro must be used only for those classes that are not exported but are used by RTTI. Examples of such classes are exceptions and classes being cast using `dynamic_cast`.

There’s more…

Some compilers export all the symbols by default but provide flags to disable such behavior. For example, GCC and Clang on Linux provide `-fvisibility=hidden`. It is highly recommended to use those flags because it leads to smaller binary size, faster loading of dynamic libraries, and better logical structuring of binary. Some inter-procedural optimizations can perform better when fewer symbols are exported. C++17 has no standard way for describing visibilities. Hopefully someday, a portable way to work with visibility will appear in C++, but until then, we have to use macros from Boost.

See also

- Read this chapter from the beginning to get more examples of `Boost.Config` usage
- Consider reading the official documentation of `Boost.Config` for the full list of the `Boost.Config` macro and their description at http://boost.org/libs/config

Detecting the Boost version and getting latest features

Boost is being actively developed, so each release contains new features and libraries. Some people wish to have libraries that compile for different versions of Boost and also want to use some of the features of the new versions.

Let’s take a look at the `boost::lexical_cast` change log. According to it, Boost 1.53 has a `lexical_cast(const CharType* chars, std::size_t count)` function overload. Our task for this recipe will be to use that function overload for new versions of Boost and work around that missing function overload for older versions.

Getting ready

Only basic knowledge of C++ and the `Boost.LexicalCast` library is required.

How to do it…

Well, all we need to do is get info about the version of Boost and use it to write optimal code. This can be done as shown in the following steps:

1. We need to include the headers containing the Boost version and `boost::lexical_cast`:

```
#include <boost/version.hpp>
#include <boost/lexical_cast.hpp>
```

2. We use the new feature of `Boost.LexicalCast` if it is available:

```
#if (BOOST_VERSION >= 105200)

int to_int(const char* str, std::size_t length) {
    return boost::lexical_cast<int>(str, length);
}
```

3. Otherwise, we are required to copy data to `std::string` first:

```
#else

int to_int(const char* str, std::size_t length) {
    return boost::lexical_cast<int>(
        std::string(str, length)
    );
}

#endif
```

4. Now, we can use the code as shown here:

```
#include <cassert>

int main() {
    assert(to_int("10000000", 3) == 100);
}
```

How it works…

The BOOST_VERSION macro contains the Boost version written in the following format: a single number for the major version, followed by three numbers for the minor version, and then two numbers for the patch level. For example, Boost 1.73.1 will contain the 107301 number in the BOOST_VERSION macro.

So, we check the Boost version in *step 2* and choose the correct implementation of the to_int function according to the abilities of Boost.LexicalCast.

There’s more…

Having a version macro is a common practice for big libraries. Some of the Boost libraries allow you to specify the version of the library to use; see `Boost.Thread` and its `BOOST_THREAD_VERSION` macro for an example.

By the way, C++ has a version macro too. Value of the `__cplusplus` macro allows you to distinguish pre-C++11 from C++11, C++11 from C++14, or C++17. Currently it can be defined to one of the following values: `199711L`, `201103L`, `201402L`, or `201703L`. Macro value stands for year and month when the committee approved the standard.

See also

- Read the recipe *Creating an execution thread* in Chapter 5, *Multithreading*, for more information about `BOOST_THREAD_VERSION` and how it affects the `Boost.Thread` library, or read the documentation at http://boost.org/libs/thread
- Read this chapter from the beginning or consider reading the official documentation of `Boost.Config` at http://boost.org/libs/config

Working with the System

In this chapter, we will cover:

- Listing files in a directory
- Erasing and creating files and directories
- Writing and using plugins
- Getting backtrace – current call sequence
- Passing data quickly from one process to another
- Syncing interprocess communications
- Using pointers in shared memory
- The fastest way to read files
- Coroutines - saving the state and postponing the execution

Introduction

Each operating system has many system calls. These calls differ from one operating system to another, while doing very close things. Boost provides portable and safe wrappers around those calls. Knowledge of wrappers is essential for writing good programs.

This chapter is devoted to working with the operating system. We already saw how to deal with network communications and signals in Chapter 6, *Manipulating Tasks*. In this chapter, we'll take a closer look at the filesystem, creating, and deleting files. We'll see how data can be passed between different system processes, how to read files at maximum speed, and how to perform other tricks.

Listing files in a directory

There are standard library functions and classes to read and write data to files. But before C++17, there were no functions to list files in a directory, get the type of a file, or get access rights for a file.

Let’s see how such iniquities can be fixed using Boost. We’ll be doing a program that lists names, write accesses, and types of files in the current directory.

Getting ready

Some basics of C++ would be more than enough for using this recipe.

This recipe requires linking against the `boost_system` and `boost_filesystem` libraries.

How to do it…

This recipe and the next one are about portable wrappers for working with a filesystem:

1. We need to include the following two headers:

```
#include <boost/filesystem/operations.hpp>
#include <iostream>
```

2. Now, we need to specify a directory:

```
int main() {
    boost::filesystem::directory_iterator begin("./");
```

3. After specifying the directory, loop through its content:

```
    boost::filesystem::directory_iterator end;
    for (; begin != end; ++ begin) {
```

4. The next step is getting the file info:

```
        boost::filesystem::file_status fs =
            boost::filesystem::status(*begin);
```

5. Now, output the file info:

```
        switch (fs.type()) {
        case boost::filesystem::regular_file:
            std::cout << "FILE       ";
            break;
        case boost::filesystem::symlink_file:
            std::cout << "SYMLINK    ";
            break;
        case boost::filesystem::directory_file:
            std::cout << "DIRECTORY  ";
            break;
        default:
            std::cout << "OTHER      ";
            break;
        }
        if (fs.permissions() & boost::filesystem::owner_write) {
            std::cout << "W ";
        } else {
            std::cout << "  ";
        }
```

6. The final step would be to output the filename:

```
        std::cout << *begin << '\n';
    } /*for*/
} /*main*/
```

That’s it; now if we run the program, it will output something like this:

```
FILE W "./main.o"
FILE W "./listing_files"
DIRECTORY W "./some_directory"
FILE W "./Makefile"
```

How it works…

Functions and classes of `Boost.Filesystem` just wrap around system-specific functions to work with files.

Note the usage of `/` in *step 2*. POSIX systems use a slash to specify paths; Windows, by default, uses backslashes. However, Windows understands forward slashes too, so `./` will work on all the popular operating systems, and it means the current directory.

Take a look at *step 3*, where we are default constructing the `boost::filesystem::directory_iterator` class. It works just like a `std::istream_iterator` class, which acts as an `end` iterator when default constructed.

Step 4 is a tricky one, not because this function is hard to understand, but because lots of conversions are happening. Dereferencing the `begin` iterator returns `boost::filesystem::directory_entry`, which is implicitly converted to `boost::filesystem::path`, which is used as a parameter for the `boost::filesystem::status` function. Actually, we may do much better:

```
boost::filesystem::file_status fs = begin->status();
```

Read the reference documentation carefully to avoid unrequired implicit conversions.

Step 5 is obvious, so we are moving to *step 6* where implicit conversion to the path happens again. A better solution would be the following:

```
std::cout << begin->path() << '\n';
```

Here, `begin->path()` returns a const reference to the `boost::filesystem::path` variable that is contained inside `boost::filesystem::directory_entry`.

There’s more…

The ;`Boost.Filesystem` is a part of C++17. All the stuff in C++17 is located in a single header file `<filesystem>` in namespace `std::filesystem`. The standard library version of filesystem differs slightly from the Boost version, mostly by using scoped enumerations (`enum class`) where `Boost.Filesystem` was using just unscoped `enum`.

There is a class ; `directory_entry`. That class provides caching of filesystem information, so if you work a lot with filesystem and query different information, try using `directory_entry` for a better performance.

Just like in the case of other Boost libraries, `Boost.Filesystem` works on pre-C++17 compilers and even on a pre-C++11 compilers.

See also

- The *Erasing and creating files and directories* recipe will show another example of the usage of `Boost.Filesystem`
- Read Boost's official documentation of `Boost.Filesystem` to get more info about its abilities; it is available at the following link: http://boost.org/libs/filesystem
- You can find the C++17 draft at http://www.open- std.org/jtc1/sc22/wg21/docs/papers/2017/n4659.pdf

Erasing and creating files and directories

Let’s consider the following lines of code:

```
std::ofstream ofs("dir/subdir/file.txt");
ofs << "Boost.Filesystem is fun!";
```

In these lines, we attempt to write something to `file.txt` in the `dir/subdir` directory. This attempt will fail if there is no such directory. The ability to work with filesystems is necessary for writing a good working code.

In this recipe, we’ll construct a directory and a subdirectory, write some data to a file, and try to create `symlink`. If the symbolic link’s creation fails, erase the created entities. We should also avoid using exceptions as a mechanism of error reporting, preferring some kind of return codes.

Let’s see how that can be done in an elegant way using Boost.

Getting ready

Basic knowledge of C++ and the `std::ofstream` class is required for this recipe.

The ;`Boost.Filesystem` is not a header-only library, so code in this recipe requires linking against the `boost_system` and `boost_filesystem` libraries.

How to do it…

We continue to deal with portable wrappers for a filesystem, and, in this recipe, we'll see how to modify the directory content:

1. As always, we need to include some headers:

```
#include <boost/filesystem/operations.hpp>
#include <cassert>
#include <fstream>
```

2. Now, we need a variable to store errors (if any):

```
int main() {
    boost::system::error_code error;
```

3. We will also create directories, if required, as follows:

```
    boost::filesystem::create_directories("dir/subdir", error);
    assert(!error);
```

4. Then, we will write data to the file:

```
    std::ofstream ofs("dir/subdir/file.txt");
    ofs << "Boost.Filesystem is fun!";
    assert(ofs);
    ofs.close();
```

5. We need to attempt to create `symlink`:

```
    boost::filesystem::create_symlink(
        "dir/subdir/file.txt", "symlink", error);
```

6. Then, we need to check that the file is accessible through `symlink`:

```
    if (!error) {
        std::cerr << "Symlink created\n";
        assert(boost::filesystem::exists("symlink"));
```

7. We'll remove the created file, if the `symlink` creation fails:

```
    } else {
        std::cerr << "Failed to create a symlink\n";

        boost::filesystem::remove_all("dir", error);
        assert(!error);

        boost::filesystem::remove("symlink", error);
        assert(!error);
    } /*if (!error)*/
} /*main*/
```

How it works…

We saw `boost::system::error_code` in action in almost all the recipes in Chapter 6, *Manipulating Tasks*. It can store information about errors and is widely used all around Boost libraries.

> *If you do not provide an instance of `boost::system::error_code` to the `Boost.Filesystem` functions, the code will compile well. In that case, when an error occurs, a `boost::filesystem::filesystem_error` exception is thrown.*

Take a careful look at *step 3*. We used the `boost::filesystem::create_directories` function, not `boost::filesystem::create_directory`, because the latter one cannot create sub-directories. It is the same story with `boost::filesystem::remove_all` and `boost::filesystem::remove`. The first can remove non empty directories, that contain files and sub-directories. The second one removes a single file.

The remaining steps are simple to understand and should not cause any trouble.

There’s more…

The `boost::system::error_code` class is a part of C++11 and can be found in the `<system_error>` header in the `std::` namespace. The classes of `Boost.Filesystem` are part of C++17.

Finally, here is a small recommendation for those who are going to use `Boost.Filesystem`. When the errors occur during filesystem, operations are routine or application require high responsibility/performance, for this, use `boost::system::error_codes`. Otherwise, catching exceptions are more preferable and reliable.

See also

The *Listing files in a directory* recipe also contains information about `Boost.Filesystem`. Read Boost's official documentation at http://boost.org/libs/filesystem to get more information and examples.

Writing and using plugins

Here’s a tricky question: we want to allow users to write extensions to the functionality of our program, but we do not want to give them the source codes. In other words we’d like to say, “*Write a function X and pack it into a shared library. We may use your function along with functions of some other users!*”

> *You meet this technique in everyday life: your browser uses it to allow third-party plugins, your text editor may use it for syntax highlighting, games use* ***dynamic library loading*** *for* ***downloadable content*** *(**DLC**s) and for adding gamer’s content, web pages are returned by servers that use modules/plugins for encryption/authentication and so forth.*

What are the requirements for a user’s function and how can we use that function at some point without linking it to the shared library?

Getting ready

Basic knowledge of C++ is required for this recipe. Reading the *The portable way to export and import functions and classes* from Chapter 10 is a requirement.

How to do it…

First of all, you have to make an agreement with your users:

1. Document the requirement for the plugin interface. For example, you may say that all the plugins must export a function with name `greet` and that the function must accept `const std::string&` and return `std::string`.

2. After that, users may start writing plugins/shared library in the following way:

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

#define API extern "C" BOOST_SYMBOL_EXPORT

API std::string greeter(const std::string& name) {
    return "Good to meet you, " + name + ".";
}
```

3. Your program code for loading a shared library must include the header from `Boost.DLL`:

```
#include <boost/dll/shared_library.hpp>
```

4. The code for loading a library must be the following:

```
int main() {
    boost::filesystem::path plugin_path = /* path-to-pligin */;

    boost::dll::shared_library plugin(
        plugin_path,
        boost::dll::load_mode::append_decorations
    );
```

5. Getting the user's function must look like this:

```
    auto greeter = plugin.get<std::string(const std::string&)>("greeter");
```

6. Done. Now, you can use that function:

```
    std::cout << greeter("Sally Sparrow");
}
```

Depending on the loaded plugin, you'll have different results:

`plugin_hello`:

```
Good to meet you, Sally Sparrow.
```

`plugin_do_not`:

```
They are fast. Faster than you can believe. Don't turn

your back, don't look away, and don't blink. Good luck, Sally Sparrow.
```

How it works…

There is a small trick in *step 2*. When you declare a function as `extern "C"`, it means that the compiler must not **mangle** (change) the function name. In other words, in *step 2* we just create a function that has a name `greet` and is exported with that exact name from the shared library.

In *step 4*, we create a `boost::dll::shared_library` variable with name `plugin`. The constructor of that variable loads the shared library by a specified path into the address space of the current executable. In *step 5*, we search for the function with name `greet` in the `plugin`. We also specify that the function has the signature `std::string(const std::string&)` and store a pointer to that function in the variable `greet`.

That's it! From now on, we can use the `greet` variable as a function, as long as the `plugin` variable and all its copies are not destroyed.

You can export multiple functions from shared library; you can even export variables.

TIP

> *Be careful! Always link C and C++ libraries dynamically to the plugin and your main executable, because otherwise your application will crash. Always use the same or ABI compatible versions of C and C++ libraries in your plugins and in your application. Otherwise your application will crash. Read the docs for typical missuses!*

There’s more…

The `Boost.DLL` is a new library; it appeared in Boost 1.61. My favorite part of the library is an ability to add platform-specific decorations to the shared library name. For example, the following code, depending on the platform, will try to load `"./some/path/libplugin_name.so"`, `"./some/path/plugin_name.dll"`, or `"./some/path/libplugin_name.dll"`:

```
boost::dll::shared_library lib(
    "./some/path/plugin_name",
    boost::dll::load_mode::append_decorations
);
```

C++17 has no `boost::dll::shared_library`-like classes. But, work is ongoing and some day we may see it in the C++ standard.

See also

Official documentation contains multiple examples and, what is more important, typical problems/missuses of the library http://boost.org/libs/dll site.

Getting backtrace – current call sequence

When reporting errors or failures, it is more important to report the steps that lead to the error rather than the error itself. Consider the naive trading simulator:

```
int main() {
    int money = 1000;
    start_trading(money);
}
```

All it reports is a line:

```
            Sorry, you're bankrupt!
```

That’s a no go. We want to know how did it happened, what were the steps that led to bankruptcy!

Okay. Let’s fix the following function and make it report the steps that led to bankruptcy:

```
void report_bankruptcy() {
    std::cout << "Sorry, you're bankrupt!\n";

    std::exit(0);
}
```

Getting started

You will need a Boost 1.65 or newer for this recipe. Basic knowledge of C++ is also a requirement.

How to do it…

For this recipe, we will need only to construct a single class and output it:

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

void report_bankruptcy() {
    std::cout << "Sorry, you're bankrupt!\n";
    std::cout << "Here's how it happened:\n"
        << boost::stacktrace::stacktrace();

    std::exit(0);
}
```

Done. Now the `report_bankruptcy()` outputs something close to the following (read it from the bottom up):

```
Sorry, you're bankrupt!
Here's how it happened:
 0# report_bankruptcy()
 1# loose(int)
 2# go_to_casino(int)
 3# go_to_bar(int)
 4# win(int)
 5# go_to_casino(int)
 6# go_to_bar(int)
 7# win(int)
 8# make_a_bet(int)
 9# loose(int)
10# make_a_bet(int)
11# loose(int)
12# make_a_bet(int)
13# start_trading(int)
14# main
15# 0x00007F79D4C48F45 in /lib/x86_64-linux-

gnu/libc.so.6
16# 0x0000000000401F39 in ./04_stacktrace
```

How it works…

All the magic is within the `boost::stacktrace::stacktrace` class. On construction, it quickly stores the current call stack in itself. `boost::stacktrace::stacktrace` is copyable and movable, so a stored a call sequence can be passed to other functions, copied into the exception classes, and even stored in some file. Do whatever you like with it!

Instances of `boost::stacktrace::stacktrace` on the output, decode the stored call sequence and attempt to get human readable function names. That's what you've seen in the example from earlier: call sequence that leads to the `report_bankruptcy()` function call.

The `boost::stacktrace::stacktrace` you to iterate over stored addresses, decode individual addresses into human readable names. If you do not like the default output format of the trace, you can write your own function that does the output in a way you prefer.

Note that backtrace usefulness depends on multiple factors. Release builds of your program may contain inline functions, resulting in less readable traces:

```
0# report_bankruptcy()
1# go_to_casino(int)
2# win(int)
3# make_a_bet(int)
4# make_a_bet(int)
5# make_a_bet(int)
6# main
```

Building your executable without debug symbols may produce a trace without many function names.

Read the Configuration and Build section of the official documentation for more information about different compilation flags and macros that may affect trace readability.

There’s more…

A `Boost.Stacktrace` library has a very neat feature for big projects. You can disable all the tracing while linking your program. It means that you do not need to rebuild all your source files. Just define `BOOST_STACKTRACE_LINK` macro for a whole project. Now, if you link with the `boost_stacktrace_noop` library, empty traces will be collected. Link with `boost_stacktrace_windbg`/`boost_stacktrace_windbg_cached`/`boost_stacktrace_backtrace`/ `... libraries` to get traces of different readability.

`Boost.Stacktrace` is a new library; it appeared in Boost in 1.65.

`boost::stacktrace::stacktrace` collects current call sequences pretty fast; it just dynamically allocates a chunk of memory and copies a bunch of addresses into it. Decoding addresses is much slower; it uses multiple platform-specific calls, may fork processes, and may initialize and use **COM**.

C++17 does not have `Boost.Stacktrace` functionality. Work is going on to add it to one of the next C++ standards.

See also

Official documentation at http://boost.org/libs/stacktrace/ has some examples on async signal safe stack tracing and detailed description of all the `Boost.Stacktrace` abilities.

Passing data quickly from one process to another

Sometimes, we write programs that communicate with each other a lot. When programs are run on different machines, using sockets is the most common technique for communication. But if multiple processes run on a single machine, we can do much better!

Let’s take a look at how to make a single memory fragment available from different processes using the `Boost.Interprocess` library.

Getting ready

Basic knowledge of C++ is required for this recipe. Knowledge of atomic variables is also required (take a look at the *See also* section for more information about atomics). Some platforms require linking against the runtime library `rt`.

How to do it…

In this example, we will be sharing a single atomic variable between processes, making it increment when a new process starts and decrement when the process terminates:

1. We need to include the following header for interprocess communications:

```
#include <boost/interprocess/managed_shared_memory.hpp>
```

2. Following the header, `typedef`, and a check will help us make sure that atomics are usable for this example:

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

typedef boost::atomic<int> atomic_t;
#if (BOOST_ATOMIC_INT_LOCK_FREE != 2)
#error "This code requires lock-free boost::atomic<int>"
#endif
```

3. Create, or get, a shared segment of memory:

```
int main() {
    boost::interprocess::managed_shared_memory
        segment(boost::interprocess::open_or_create, "shm1-cache", 1024);
```

4. Get, or construct, an `atomic` variable:

```
    atomic_t& atomic
        = *segment.find_or_construct<atomic_t> // 1
            ("shm1-counter")                   // 2
            (0)                                // 3
    ;
```

5. Work with the `atomic` variable in a usual way:

```
    std::cout << "I have index " << ++ atomic
        << ". Press any key...\n";
    std::cin.get();
```

6. Destroy the `atomic` variable:

```
    const int snapshot = --atomic;
    if (!snapshot) {
        segment.destroy<atomic_t>("shm1-counter");
        boost::interprocess::shared_memory_object
                ::remove("shm1-cache");
    }
} /*main*/
```

That's all! Now if we run multiple instances of this program simultaneously, we'll see that each new instance increments its index value:

```
I have index 1. Press any key...
I have index 2.

Press any key...
I have index 3. Press any key...
I have index 4. Press any key...
I have index 5.
```

```
        Press any key...
```

How it works…

The main idea of this recipe is to get a segment of memory that is visible to all processes and place some data in it. Let's take a look at *step 3*, where we retrieve such a segment of memory. Here, `shm1- cache` is the name of the segment (different segments differ in names). You may give any names to the segments. The first parameter is `boost::interprocess::open_or_create`, which tells that `boost::interprocess::managed_shared_memory` must open an existing segment with the name `shm1- cache` or construct it. The last parameter is the size of the segment.

The size of the segment must be big enough to fit the `Boost.Interprocess` library-specific data in it. That's why we used `1024` and not `sizeof(atomic_t)`. But actually, the operating system rounds this value to the nearest bigger supported value, which is usually equal to or bigger than 4 kilobytes.

Step 4 is a tricky one, as we are performing multiple tasks at the same time here. In part `2` of this step, we find or construct a variable with the name `shm1-counter` in the segment. In part `3` of *step 4*, we provide a parameter, which is used for the initialization of a variable, if it has not been found in *step 2*. This parameter is used only if the variable is not found and must be constructed, otherwise it is ignored. Take a closer look at the second line (part `1`). See the call to the dereference operator `*`. We are doing it because `segment.find_or_construct<atomic_t>` returns a pointer to `atomic_t`, and working with bare pointers in C++ is a bad style.

We are using atomic variables in shared memory! This is required, because two or more processes may simultaneously work with the same `shm1-counter` atomic variable.

You must be very careful when working with objects in shared memory; do not forget to destroy them! In *step 6*, we are destroying the object and segment using their names.

There's more…

Take a closer look at *step 2,* where we are checking for `BOOST_ATOMIC_INT_LOCK_FREE != 2`. We are checking that `atomic_t` does not use mutexes. This is very important, because the usual mutexes do not work in shared memory. So if `BOOST_ATOMIC_INT_LOCK_FREE` is not equal to `2`, we get an undefined behavior.

Unfortunately, C++11 has no interprocess classes, and, as far as I know, `Boost.Interprocess` is not proposed for inclusion in C++20.

Once a managed segment is created, it cannot increase in size automatically! Make sure that you are creating segments big enough for your needs, or take a look at the See also section for information about increasing managed segments.

Shared memory is the fastest way for processes to communicate, but works for processes that may share memory. That usually means that processes must run on the same host or on a **symmetric multiprocessing** (**SMP**) cluster.

See also

- The *Syncing interprocess communications* recipe will tell you more about shared memory, interprocess communications, and syncing access to resources in shred memory
- The *Quick access to a common resource using atomics* recipe for more information about atomics
- Boost's official documentation of `Boost.Interprocess` may also help; it is available at http://boost.org/libs/interprocess
- How to increase managed segments is described in *Growing managed segments* at http://boost.org/libs/interprocess

Syncing interprocess communications

In the previous recipe, we saw how to create shared memory and how to place some objects in it. Now, it’s time to do something useful. Let’s take an example from the *Making a work_queue* recipe in Chapter 5, *Multithreading*, and make it work for multiple processes. At the end of this example, we’ll get a class that may store different tasks and pass them between processes.

Getting ready

This recipe uses techniques from the previous one. You will also need to read the *Making a work_queue* recipe in Chapter 5, *Multithreading*, and get its main idea. The example requires linking against the runtime library `rt` on some platforms.

How to do it…

It is considered that spawning separate sub-processes instead of threads makes a program more reliable, because termination of a sub-process does not terminate the main process. We won't argue with that assumption here, and just see how data sharing between processes can be implemented.

1. A lot of headers are required for this recipe:

```
#include <boost/interprocess/managed_shared_memory.hpp>
#include <boost/interprocess/containers/deque.hpp>
#include <boost/interprocess/allocators/allocator.hpp>
#include <boost/interprocess/sync/interprocess_mutex.hpp>
#include <boost/interprocess/sync/interprocess_condition.hpp>
#include <boost/interprocess/sync/scoped_lock.hpp>

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

2. Now, we need to define our structure, `task_structure`, which will be used to store tasks:

```
struct task_structure {
    // ...
};
```

3. Let's start writing the `work_queue` class:

```
class work_queue {
public:
    typedef boost::interprocess::managed_shared_memory
            managed_shared_memory_t;

    typedef task_structure task_type;
    typedef boost::interprocess::allocator<
        task_type,
        boost::interprocess::managed_shared_memory::segment_manager
    > allocator_t;
```

4. Write the members of `work_queue` as follows:

```
private:
    managed_shared_memory_t segment_;
    const allocator_t       allocator_;

    typedef boost::interprocess::deque<task_type, allocator_t> deque_t;
    deque_t&        tasks_;

    typedef boost::interprocess::interprocess_mutex mutex_t;
    mutex_t&        mutex_;

    typedef boost::interprocess::interprocess_condition condition_t;
    condition_t&    cond_;

    typedef boost::interprocess::scoped_lock<mutex_t> scoped_lock_t;
```

5. Initialization of members must look like the following:

```
public:
    explicit work_queue()
        : segment_(
              boost::interprocess::open_or_create,
              "work-queue",
              1024 * 1024 * 32
        )
        , allocator_(segment_.get_segment_manager())
        , tasks_(
            *segment_.find_or_construct<deque_t>
              ("work-queue:deque")(allocator_)
        )
        , mutex_(
            *segment_.find_or_construct<mutex_t>
              ("work-queue:mutex")()
        )
        , cond_(
            *segment_.find_or_construct<condition_t>
              ("work-queue:condition")()
        )
    {}
```

6. We need to do some minor changes to the member functions of `work_queue`, such as using `scoped_lock_t`, instead of the original unique locks:

```
boost::optional<task_type> try_pop_task() {
    boost::optional<task_type> ret;
    scoped_lock_t lock(mutex_);
    if (!tasks_.empty()) {
        ret = tasks_.front();
        tasks_.pop_front();
    }
    return ret;
}
```

7. Do not forget about the resources cleanup:

```
void cleanup() {
    segment_.destroy<condition_t>("work-queue:condition");
    segment_.destroy<mutex_t>("work-queue:mutex");
    segment_.destroy<deque_t>("work-queue:deque");

    boost::interprocess::shared_memory_object
        ::remove("work-queue");
}
```

How it works…

In this recipe, we are doing almost exactly the same things as in the *Making a work_queue class* recipe in Chapter 5, *Multithreading*, but we allocate the data in shared memory.

Take additional care when storing the shared memory objects that have pointers or references as member fields. We'll see how to cope with pointers in the next recipe.

Take a look at *step 2*. We did not use `boost::function` as a task type because it has pointers in it, so it does not work in shared memory.

Step 3 is interesting because of `allocator_t`. If memory is not allocated from the shared memory segment, it is available to other processes; that's why a specific allocator for containers is required. An `allocator_t` is a stateful allocator, which means that it is copied along with the container. Also, it cannot be default constructed.

Step 4 is pretty simple, except that we have only references to `tasks_`, `mutex_`, and `cond_`. This is done because objects themselves are constructed in the shared memory. So, `work_queue` may only store references in them.

In *step 5*, we are initializing members. This code must be familiar to you. We were doing exactly the same things in the previous recipe.

We are providing an instance of the allocator to `tasks_` while constructing it. That's because `allocator_t` cannot be constructed by the container itself. Shared memory is not destructed at the exit event of a process, so we may run the program once, post the tasks to a work queue, stop the program, start some other program, and get tasks stored by the first instance of the program. Shared memory is destroyed only at restart, or if you explicitly call `segment.deallocate("work-queue");`.

There's more…

As was already mentioned in the previous recipe, C++17 has no classes from `Boost.Interprocess`. Moreover, you must not use C++17 or C++03 containers in shared memory segments. Some of those containers may work, but that behavior is not portable.

If you look inside some of the `<boost/interprocess/containers/*.hpp>` headers, you'll find that they just use containers from the `Boost.Containers` library:

```
namespace boost { namespace interprocess {
    using boost::container::vector;
}}
```

Containers of `Boost.Interprocess` have all the benefits of the `Boost.Containers` library, including rvalue references and their emulation on older compilers.

A `Boost.Interprocess` is the fastest solution for communication of processes that are running on the same machine.

See also

- The *Using pointers in shared memory* recipe
- Read Chapter 5, *Multithreading*, for more information about synchronization primitives and multithreading
- Refer to Boost's official documentation of the `Boost.Interprocess` library for more examples and information; it is available at the following link: http://boost.org/libs/interprocess

Using pointers in a shared memory

It is hard to imagine writing some low-level C++ core classes without pointers. Pointers and references are everywhere in C++, and they do not work in shared memory! So, if we have a structure like this in shared memory and assign the address of some integer variable in shared memory to `pointer_`, the `pointer_` would be invalid in other process:

```
struct with_pointer {
    int* pointer_;
    // ...
    int value_holder_;
};
```

How can we fix that?

Getting ready

The previous recipe is required for understanding this one. The example requires linking against the runtime system library `rt` on some platforms.

How to do it…

Fixing it is very simple; we need only to replace the pointer with `offset_ptr<>`:

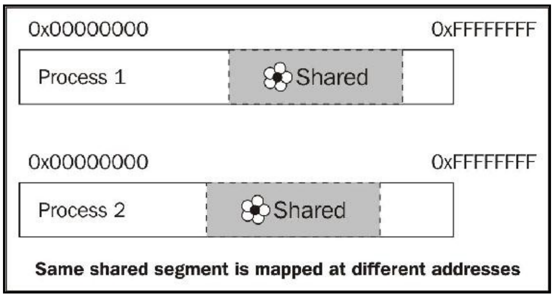
```
#include <boost/interprocess/offset_ptr.hpp>

struct correct_struct {
    boost::interprocess::offset_ptr<int> pointer_;
    // ...
    int value_holder_;
};
```

Now, we are free to use it like a usual pointer:

```
int main() {
    boost::interprocess::managed_shared_memory
        segment(boost::interprocess::open_or_create, "segment", 4096);

    correct_struct* ptr =
        segment.find<correct_struct>("structure").first;

    if (ptr) {
        std::cout << "Structure found\n";
        assert(*ptr->pointer_ == ethalon_value);
        segment.destroy<correct_struct>("structure");
    }
}
```

How it works…

We cannot use pointers in shared memory because, when a piece of shared memory is mapped into the address space of a process, its address is valid only for that process. When we are getting the address of a variable, it is just a local address for that process. Other processes will map shared memory to a different base address, and as a result, the variable address differs.



So, how can we work with an address that is always changing? There is a trick! As the pointer and structure are in the same shared memory segment, the distance between them does not change. The idea behind `boost::interprocess::offset_ptr` is to remember that distance between `offset_ptr` and the pointed value. On deference, `offset_ptr` adds the distance value to the process-dependent address of the `offset_ptr` variable.

The offset pointer imitates the behavior of pointers, so it is a drop-in replacement that can be applied quickly.

Do not place the classes that may have pointers or references into shared memory!

There’s more…

An offset pointer works slightly slower than the usual pointer because, on each dereference, it is required to compute the address. But, this difference is not usually what should bother you.

C++17 has no offset pointers.

See also

- Boost's official documentation contains many examples and more advanced `Boost.Interprocess` features; it is available at http://boost.org/libs/interprocess
- The *Fastest way to read files* recipe contains information about some nontraditional usage of the `Boost.Interprocess` library

The fastest way to read files

All around the Internet, people are asking *“What is the fastest way to read files?”*. Let’s make our task for this recipe even harder: what is the fastest and portable way to read binary files?

Getting ready

Basic knowledge of C++ and the `std::fstream` is required for this recipe.

How to do it…

The technique from this recipe is widely used by applications critical to input and output performance. It’s the fastest way to read files:

1. We need to include two headers from the `Boost.Interprocess` library:

```
#include <boost/interprocess/file_mapping.hpp>
#include <boost/interprocess/mapped_region.hpp>
```

2. Now, we need to open a file:

```
const boost::interprocess::mode_t mode = boost::interprocess::read_only;
boost::interprocess::file_mapping fm(filename, mode);
```

3. The main part of this recipe is mapping all the files to memory:

```
boost::interprocess::mapped_region region(fm, mode, 0, 0);
```

4. Getting a pointer to the data in the file:

```
const char* begin = static_cast<const char*>(
    region.get_address()
);
```

That’s it! Now, we may work with a file just like with a usual memory:

```
const char* pos = std::find(
    begin, begin + region.get_size(), '\1'
);
```

How it works…

All popular operating systems have the ability to map a file to processes' address space. After such mapping is done, the process may work with those addresses just like with a usual memory. The operating system takes care of all the file operations, such as caching and read ahead.

Why is it faster than traditional read/writes? That's because in most cases read/write is implemented as memory mapping and copying data to a user-specified buffer. So, read usually does a little bit more than memory map.

Just like in the case of standard library's `std::fstream`, we must provide an open mode when opening a file. See *step 2* where we provided the `boost::interprocess::read_only` mode.

See *step 3* where we mapped a whole file at once. This operation is actually really fast, because OS does not read data from the disk, but waits for requests to a part of the mapped region. After a part of the mapped region was requested, the OS loads that part of the file from the disk into the memory. As we may see, memory mapping operations are lazy, and the size of the mapped region does not affect performance.

However, a 32-bit OS cannot memory-map big files, so you have to map them by pieces. POSIX (Linux) operating systems require the `_FILE_OFFSET_BITS=64` macro to be defined for the whole project to work with big files on a 32-bit platform. Otherwise, the OS won't be able to map parts of the file that are beyond 4 GB.

Now, it's time to measure the performance:

```
$ TIME="%E" time ./reading_files m
mapped_region: 0:00.08


$ TIME="%E" time ./reading_files r
ifstream: 0:00.09

$ TIME="%E" time ./reading_files a
C: 0:00.09
```

Just as it was expected, memory-mapped files are slightly faster than traditional reads. We may also see that pure C methods have the same performance as the C++ `std::ifstream` class, so do not use functions related to `FILE*` in C++. They are just for C, not for C++!

For optimal performance of `std::ifstream`, do not forget to open files in binary mode and read data by blocks:

```
std::ifstream f(filename, std::ifstream::binary);
// ...
char c[kilobyte];
f.read(c, kilobyte);
```

There’s more…

Unfortunately, classes for memory mapping files are not a part of C++17 and looks like they won’t be in C++20 either.

Writing to memory-mapped regions is also a very fast operation. The OS caches the writes and does not flush modifications to the disc immediately. There is a difference between OS and the `std::ofstream` data caching. In case the `std::ofstream` data is cached by an application and if it terminates, the cached data can be lost. When data is cached by the OS, termination of the application does not lead to data loss. Power failures and OS crashes lead to data loss in both cases.

If multiple processes map a single file, and one of the processes modifies the mapped region, changes are immediately visible to other processes (even without actually writing the data to disk! Modern OS are very clever!).

See also

The `Boost.Interprocess` library contains a lot of useful features to work with the system; not all of them are covered in this book. You may read more about this great library at the official site: http://boost.org/libs/interprocess.

Coroutines - saving the state and postponing the execution

Nowadays, plenty of embedded devices still have only a single core. Developers write for those devices, trying to squeeze maximum performance out of them.

Using `Boost.Threads` or some other thread library for such devices is not effective. The OS will be forced to schedule threads for execution, manage resources, and so on, as the hardware cannot run them in parallel.

So, how can we force a program to switch to the execution of a subprogram while waiting for some resource in the main part? Moreover, how can we control the time of the subprogram’s execution?

Getting ready

Basic knowledge of C++ and templates is required for this recipe. Reading some recipes about `Boost.Function` may also help.

How to do it…

This recipe is about **coroutines** or **subroutines** that allow multiple entry points. Multiple entry points give us an ability to suspend and resume the execution of a program at certain locations, switching to/from other subprograms.

1. The `Boost.Coroutine2` library takes care of almost everything. We just need to include its header:

```
#include <boost/coroutine2/coroutine.hpp>
```

2. Make a coroutine type with the required input parameter type:

```
typedef boost::coroutines2::asymmetric_coroutine<std::size_t> corout_t;
```

3. Make a class, representing a subprogram:

```
struct coroutine_task {
    std::string& result;

    coroutine_task(std::string& r)
        : result(r)
    {}

    void operator()(corout_t::pull_type& yield);

private:
    std::size_t ticks_to_work;
    void tick(corout_t::pull_type& yield);
};
```

4. Let's create the coroutine itself:

```
int main() {
    std::string result;
    coroutine_task task(result);
    corout_t::push_type coroutine(task);
```

5. Now, we may execute the subprogram while waiting for some event in the main program:

```
    // Somewhere in main():

    while (!spinlock.try_lock()) {
        // We may do some useful work, before
        // attempting to lock a spinlock once more.
        coroutine(10); // 10 is the ticks count to run.
    }
    // Spinlock is locked.
    // ...

    while (!port.block_ready()) {
        // We may do some useful work, before
        // attempting to get block of data once more.
        coroutine(300); // 300 is the ticks count to run.

        // Do something with `result` variable.
    }
```

6. The coroutine method may look like this:

```
void coroutine_task::operator()(corout_t::pull_type& yield) {
    ticks_to_work = yield.get();

    // Prepare buffers.
    std::string buffer0;

    while (1) {
        const bool requiers_1_more_copy = copy_to_buffer(buffer0);
        tick(yield);

        if (requiers_1_more_copy) {
            std::string buffer1;
            copy_to_buffer(buffer1);
            tick(yield);

            process(buffer1);
            tick(yield);
        }

        process(buffer0);
        tick(yield);
    }
}
```

7. The `tick()` function could be implemented like this:

```
void coroutine_task::tick(corout_t::pull_type& yield) {
    if (ticks_to_work != 0) {
        --ticks_to_work;
    }

    if (ticks_to_work == 0) {
        // Switching back to main.
        yield();

        ticks_to_work = yield.get();
    }
}
```

How it works…

At *step 2*, we are describing the input parameter of our subprogram using the `std::size_t` as a template parameter.

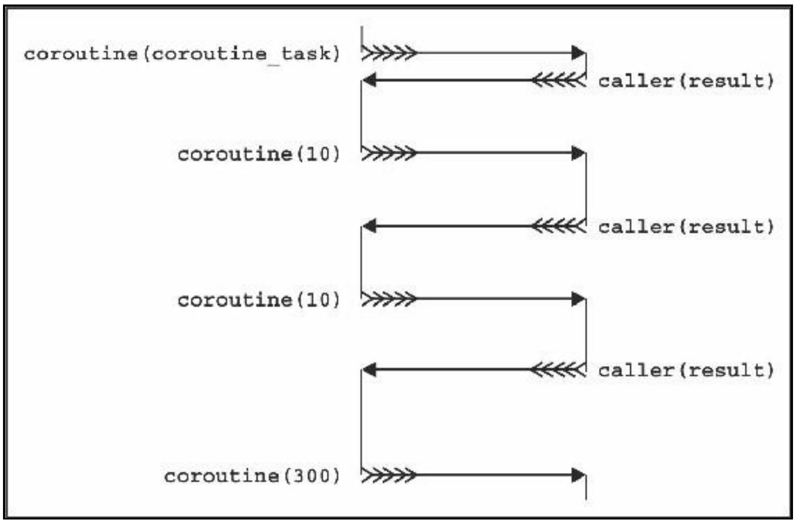
Step 3 is pretty simple, except for the `corout_t::pull_type& yield` parameters. We'll see it in action in a few seconds.

When we call `coroutine(10)` in *step 5*, we are making a coroutine program to execute. Execution jumps to `coroutine_task::operator()`, where a call to `yield.get()` returns the input parameter `10`. The execution continues and the `coroutine_task::tick` function measures elapsed ticks.

Here comes the most interesting part!

In *step 7,* if in function `coroutine_task::tick` the `ticks_to_work` variable becomes equal to `0`, the execution of the coroutine suspends at `yield()` and `main()` continues execution. On the next call to `coroutine(some_value)`, the execution of the coroutine continues from the middle of the `tick` function, right at the line next to `yield()`. In that line, `ticks_to_work = yield.get();` is executed and the `ticks_to_work` variable starts to hold a new input value `some_value`.

It means that we can suspend/continue the coroutine in multiple places of the function. All the function state and variables are restored:



Let me describe the main difference between coroutines and threads. When a coroutine is executed, the main task does nothing. When the main task is executed, the coroutine task does nothing. You have no such guarantee with threads. With coroutines, you explicitly specify when to start a subtask and when to suspend it. In a single core environment, threads may switch at any moment; you cannot control that behavior.

There’s more…

While switching threads, the OS does a lot of work, so it is not a very fast operation. However, with coroutines, you have full control over switching tasks; moreover, you do not need to do some OS-specific internal kernel work. Switching coroutines is much faster than switching threads, although, not as fast as calling `boost::function`.

The `Boost.Coroutine2` library takes care about calling a destructor for variables in a coroutine task, so there’s no need to worry about leaks.

Coroutines use the `boost::coroutines2::detail::forced_unwind` exception to free resources that are not derived from `std::exception`. You must take care to not catch that exception in coroutine tasks.

You cannot copy `Boost.Coroutine2` coroutines, but you can `std::move` them.

There is a `Boost.Coroutine` library (without `2` at the end!), that does not require a C++11 compatible compiler. But that library is deprecated and has some differences (for example it does not propagate exceptions from coroutines). Beware of the differences! `Boost.Coroutine` also changed its interface significantly in Boost 1.56.

C++17 has no coroutines. But **Coroutines TS** is almost ready, so the chances are high that next C++ standard will have them out of the box.

Coroutines TS differs from `Boost.Coroutine2`! Boost provides **stackful** coroutines, which means that you do not need to specially decorate your code with macro/keywords to use them. But it also means that Boost coroutines are harder to optimize by the compiler and that they may allocate more memory. Coroutines TS provides **stackless** coroutines, which means that compiler could precisely compute the required memory for a coroutine and even optimize out the whole coroutine. However, this approach requires code changes and may be slightly harder to adopt.

See also

- Boost's official documentation contains more examples, performance notes, restrictions, and use cases for the `Boost.Coroutines2` library; it is available at the following link http://boost.org/libs/coroutine2
- Take a look at recipes from Chapter 2, *Managing Resources*, and Chapter 5, *Multithreading*, to get the difference between the `Boost.Coroutine`, `Boost.Thread`, and `Boost.Function` libraries
- Interested in Coroutines TS? Here's an interesting talk on their implementation from the author *CppCon 2016: Gor Nishanov. C++ Coroutines: Under the covers* at https://www.youtube.com/watch?v=8C8NnE1Dg4A

Scratching the Tip of the Iceberg

In this chapter, we will cover:

- Working with graphs
- Visualizing graphs
- Using a true random number generator
- Using portable math functions
- Writing test cases
- Combining multiple test cases in one test module
- Manipulating images

Introduction

Boost is a huge collection of libraries. Some of these libraries are small and meant for everyday use, while others require a separate book to describe all their features. This chapter is devoted to some of those big libraries and provides a basic understanding of it.

The first two recipes will explain the usage of `Boost.Graph`. It is a big library with an insane number of algorithms. We'll see some basics and probably the most important part of development—visualization of graphs.

We'll also see a very useful recipe for generating true random numbers. This is a very important recipe for writing secure cryptography systems.

Some C++ standard libraries lack math functions. We'll see how that can be fixed using Boost. But, the format of this book leaves no space for describing all the functions.

Writing test cases is described in the *Writing test cases* and *Combining multiple test cases in one test module* recipes. This is important for any production-quality system.

The last recipe is about a library that helped me in a lot of my course work during my university days. Images can be created and modified using it. I personally used it to visualize different algorithms, hide data in images, sign images, and generate textures.

Unfortunately, even this chapter cannot tell you about all the Boost libraries. Maybe someday, I'll write one more book, and then, a few more.

Working with graphs

Some tasks require representing data as a graph. The `Boost.Graph` is a library that was designed to provide a flexible way of constructing and representing graphs in memory. It also contains a lot of algorithms to work with graphs, such as topological sort, breadth first search, depth first search, and Dijkstra shortest paths.

Well, let’s perform some basic tasks with `Boost.Graph`!

Getting ready

Only basic knowledge of C++ and templates are required for this recipe.

How to do it…

In this recipe, we’ll describe a graph type, create a graph of that type, add some vertexes and edges to the graph, and search for a specific vertex. That should be enough to start with `Boost.Graph`.

1. We start by describing the graph type:

```
#include <boost/graph/adjacency_list.hpp>
#include <string>

typedef std::string vertex_t;
typedef boost::adjacency_list<
    boost::vecS
    , boost::vecS
    , boost::bidirectionalS
    , vertex_t
> graph_type;
```

2. Now, we construct it:

```
int main() {
    graph_type graph;
```

3. Let’s carry out some undocumented trick that speeds up graph construction:

```
    static const std::size_t vertex_count = 5;
    graph.m_vertices.reserve(vertex_count);
```

4. Now, we are ready to add vertexes to the graph:

```
    typedef boost::graph_traits<
        graph_type
    >::vertex_descriptor descriptor_t;

    descriptor_t cpp
        = boost::add_vertex(vertex_t("C++"), graph);
    descriptor_t stl
        = boost::add_vertex(vertex_t("STL"), graph);
    descriptor_t boost
        = boost::add_vertex(vertex_t("Boost"), graph);
    descriptor_t guru
        = boost::add_vertex(vertex_t("C++ guru"), graph);
    descriptor_t ansic
        = boost::add_vertex(vertex_t("C"), graph);
```

5. It is time to connect vertexes with edges:

```
    boost::add_edge(cpp, stl, graph);
    boost::add_edge(stl, boost, graph);
    boost::add_edge(boost, guru, graph);
    boost::add_edge(ansic, guru, graph);
} // end of main()
```

6. We may make a function that searches for some vertex:

```
inline void find_and_print(
    const graph_type& graph, boost::string_ref name)
{
```

7. Next is a code that gets iterators to all vertexes:

```
        typedef typename boost::graph_traits<
            graph_type
        >::vertex_iterator vert_it_t;

        vert_it_t it, end;
        boost::tie(it, end) = boost::vertices(graph);
```

8. It’s time to run a search for the required vertex:

```
        typedef typename boost::graph_traits<
            graph_type
        >::vertex_descriptor desc_t;

        for (; it != end; ++ it) {
            const desc_t desc = *it;
            const vertex_t& vertex = boost::get(
                boost::vertex_bundle, graph
            )[desc];

            if (vertex == name.data()) {
                break;
            }
        }

        assert(it != end);
        std::cout << name << '\n';
    } /* find_and_print */
```

How it works…

At *step 1*, we are describing what our graph must look like and on what types it must be based. The `boost::adjacency_list` is a class that represents graphs as two-dimensional structures, where the first dimension contains vertexes and the second dimension contains edges for that vertex. The `boost::adjacency_list` must be the default choice for representing a graph because it suits most cases.

The first template parameter, `boost::adjacency_list`, describes the structure used to represent the edge list for each of the vertexes. The second one describes a structure to store vertexes. We may choose different standard library containers for those structures using specific selectors, as listed in the following table:

Selector	Standard library container
`boost::vecS`	`std::vector`
`boost::listS`	`std::list`
`boost::slistS`	`std::slist`
`boost::setS`	`std::set`
`boost::multisetS`	`std::multiset`
`boost::hash_setS`	`std::hash_set`

The third template parameter is used to make an indirect, directed, or bidirectional graph. Use the `boost::undirectedS`, `boost::directedS`, and `boost::bidirectionalS` selectors respectively.

The fifth template parameter describes the datatype that is used as a vertex. In our example we choose `std::string`. We may also support a datatype for edges and provide it as a template parameter.

Steps 2 and *3* are simple, but in *step 4* you may see some undocumented way to speed up graph construction. In our example, we use `std::vector` as a container for storing vertexes, so we may force it to reserve memory for the required number of vertexes. This leads to less memory allocations/deallocations and copy operations during insertion of vertexes into the graph. This step is not very portable and may break in one of the future versions of Boost, because the step is highly dependent on the current implementation of `boost::adjacency_list` and on the chosen container type for storing vertexes.

At *step 4*, we see how vertexes can be added to the graph. Note how `boost::graph_traits<graph_type>` has been used. The `boost::graph_traits` class is used to get types that are specific for a graph type. We'll see its usage and the description of some graph-specific types later in this chapter. *Step 5* shows what we need to connect vertexes with edges.

If we had provided some datatype for the edges, adding an edge would look as follows: `boost::add_edge(ansic, guru, edge_t(initialization_parameters), graph)`

In *step 6*, the graph type is a `template` parameter. This is recommended to achieve better code re-usability and make this function work with other graph types.

At *step 7*, we see how to iterate over all the vertexes of the graph. The type of vertex iterator is received from `boost::graph_traits`. The function `boost::tie` is a part of `Boost.Tuple` and is used for getting values from tuples to the variables. So, calling `boost::tie(it, end) = boost::vertices(g)` puts the `begin` iterator into the `it` variable and the `end` iterator into the `end` variable.

It may come as a surprise to you, but dereferencing a vertex iterator does not return vertex data. Instead, it returns the vertex descriptor `desc`, which can be used in `boost::get(boost::vertex_bundle, g)[desc]` to get vertex data, just as we have done in *step 8*. The vertex descriptor type is used in many of the `Boost.Graph` functions. We already saw its use in the edge construction function in *step 5*.

As already mentioned, the `Boost.Graph` *library contains implementations of many algorithms. You may find many search policies implemented, but we won't discuss them in this book. We limit this recipe just to the basics of the graph library.*

There’s more…

The `Boost.Graph` library is not a part of C++17 and it won’t be a part of the next C++ standard. The current implementation does not support C++11 features like rvalue references. If we are using vertexes that are heavy to copy, we may gain speed using the following trick:

```
vertex_descriptor desc = boost::add_vertex(graph);
boost::get(boost::vertex_bundle, g_)[desc] = std::move(vertex_data);
```

It avoids copy constructions inside `boost::add_vertex(vertex_data, graph)` and uses the default construction with move assignment instead.

The efficiency of `Boost.Graph` depends on multiple factors, such as the underlying containers types, graph representation, edge, and vertex datatypes.

See also

Reading the *Visualizing graphs* recipe can help you work easily with graphs. You may also consider reading its official documentation at the following link: http://boost.org/libs/graph

Visualizing graphs

Making programs that manipulate graphs was never easy because of issues with visualization. When we work with standard library containers such as `std::map` and `std::vector`, we can always print the container's contents and see what is going on inside. But when we work with complex graphs, it is hard to visualize the content in a clear way; textual representation is not human friendly because it typically contains too many vertexes and edges.

In this recipe, we'll take a look at the visualization of `Boost.Graph` using the **Graphviz** tool.

Getting ready

To visualize graphs, you will need a Graphviz visualization tool. Knowledge of the preceding recipe is also required.

How to do it…

Visualization is done in two phases. In the first phase, we make our program output the graph's description in a text format suitable for Graphviz. In the second phase, we import the output from the first step to the visualization tool. The numbered steps in this recipe are all about the first phase.

1. Let's write the `std::ostream` operator for `graph_type` as done in the preceding recipe:

```
#include <boost/graph/graphviz.hpp>

std::ostream& operator<<(std::ostream& out, const graph_type& g) {
    detail::vertex_writer<graph_type> vw(g);
    boost::write_graphviz(out, g, vw);

    return out;
}
```

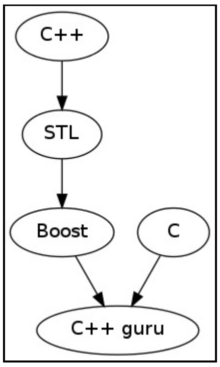
2. The `detail::vertex_writer` structure, used in the preceding step, must be defined as follows:

```
#include <iosfwd>

namespace detail {
    template <class GraphT>
    class vertex_writer {
        const GraphT& g_;

    public:
        explicit vertex_writer(const GraphT& g)
            : g_(g)
        {}

        template <class VertexDescriptorT>
        void operator()(
            std::ostream& out,
            const VertexDescriptorT& d) const
        {
            out << " [label=\""
                << boost::get(boost::vertex_bundle, g_)[d]
                << "\"]";
        }
    }; // vertex_writer
} // namespace detail
```

That's all. Now, if we visualize the graph from the previous recipe using the `std::cout << graph;` command, the output can be used to create graphical pictures using the `dot` command-line utility:

```
$ dot -Tpng -o dot.png

digraph G {
0 [label="C++"];
1 [label="STL"];
2 [label="Boost"];
3 [label="C++ guru"];
4 [label="C"];
0->1 ;
1->2 ;
2->3 ;
4->3 ;
}
```

The output of the preceding command is depicted in the following figure:



We may also use the **Gvedit** or **XDot** programs for visualization, if the command line frightens you.

How it works…

The `Boost.Graph` library contains function to output graphs in Graphviz (DOT) format. If we write `boost::write_graphviz(out, g)` with two parameters in *step 1*, the function outputs a graph picture with vertexes numbered from `0`. That's not very useful, so we provide an instance of the hand-written `vertex_writer` class that outputs vertex names.

As we can see in *step 2*, the Graphviz tool understands DOT format. If you wish to output more info for your graph, then you may need to read the Graphviz documentation for more info about the DOT format.

If you wish to add some data to the edges during visualization, we need to provide an instance of the edge visualizer as a fourth parameter to `boost::write_graphviz`.

There’s more…

C++17 does not contain `Boost.Graph` or the tools for graph visualization. But you do not need to worry, as there are a lot of other graph formats and visualization tools and `Boost.Graph` can work with plenty of them.

See also

- The *Working with graphs* recipe contains information about the construction of `Boost.Graphs`
- You will find a lot of information about the DOT format and Graphviz at http://www.graphviz.org/
- Boost's official documentation of the `Boost.Graph` library contains multiple examples and useful information, and can be found at http://boost.org/libs/graph

Using a true random number generator

I know of many examples of commercial products that use incorrect methods for getting random numbers. It's a shame that some companies still use `rand()` in cryptography and banking software.

Let's see how to get a fully random **uniform distribution** using `Boost.Random` that is suitable for banking software.

Getting started

Basic knowledge of C++ is required for this recipe. Knowledge about different types of distributions will also be helpful. The code in this recipe requires linking against the `boost_random` library.

How to do it…

To create a true random number, we need some help from the operating system or processor. This is how it can be done using Boost:

1. We need to include the following headers:

```
#include <boost/config.hpp>
#include <boost/random/random_device.hpp>
#include <boost/random/uniform_int_distribution.hpp>
```

2. Advanced random bits providers have different names under different platforms:

```
int main() {
    static const std::string provider =
#ifdef BOOST_WINDOWS
        "Microsoft Strong Cryptographic Provider"
#else
        "/dev/urandom"
#endif
    ;
```

3. Now, we are ready to initialize the generator with `Boost.Random`:

```
    boost::random_device device(provider);
```

4. Let’s get a uniform distribution that returns a value between `1000` and `65535`:

```
    boost::random::uniform_int_distribution<unsigned short> random(1000);
```

That’s it. Now, we may get true random numbers using the `random(device)` call.

How it works…

Why does the `rand()` function not suit banking? Because it generates pseudo-random numbers, which means that the hacker may predict the next generated number. This is an issue with all pseudo-random number algorithms. Some algorithms are easier to predict and some harder, but it's still possible.

That's why, we are using `boost::random_device` in this example (see *step 3*). That device gathers **entropy**—information about random events from all around the operating system to produce unpredictable uniform random bits. The examples of such events are delays between pressed keys, delays between some of the hardware interruptions, and the internal CPU's random bits generators.

Operating systems may have more than one such type of random bit generator. In our example for POSIX systems, we used `/dev/urandom` instead of the more secure `/dev/random` because the latter remains in a blocked state until enough random events have been captured by the OS. Waiting for entropy may take seconds, which is usually unsuitable for applications. Use `/dev/random` for the long-lifetime **GPG**/**SSL**/**SSH** keys.

Now that we are done with generators, it's time to move to *step 4* and talk about distribution classes. If the generator just generates uniform distributed bits, the distribution class makes a random number from those bits. In *step 4*, we made a uniform distribution that returns a random number of `unsigned short` type. The parameter `1000` means that distribution must return numbers greater or equal to `1000`. We can also provide the maximum number as a second parameter, which is by default equal to the maximum value storable in the return type.

There’s more…

The `Boost.Random` has a huge number of true/pseudo random bit generators and distributions for different needs. Avoid copying distributions and generators. This may turn out to be an expensive operation.

C++11 has support for different distribution classes and generators. You may find all the classes from this example in the `<random>` header in the `std::` namespace. The `Boost.Random` libraries do not use C++11 features, and they are not really required for that library either. Should you use Boost implementation or standard library? Boost provides better portability across systems. However, some standard libraries may have assembly-optimized implementations and may provide some useful extensions.

See also

The official documentation contains a full list of generators and distributions with descriptions. It is available at the following link: http://boost.org/libs/random.

Using portable math functions

Some projects require specific trigonometric functions, a library for numerically solving ordinary differential equations and working with distributions and constants. All those parts of `Boost.Math` will be hard to fit even in a separate book. A single recipe definitely won't be enough. So, let's focus on very basic everyday-use functions to work with float types.

We'll write a portable function that checks input value for infinity and **Not-a-Number** (**NaN**) values and changes the sign if the value is negative.

Getting ready

Basic knowledge of C++ is required for this recipe. Those who know C99 standard will find a lot common in this recipe.

How to do it…

Perform the following steps to check the input value for infinity and NaN values and change the sign if the value is negative:

1. We need the following headers:

```
#include <boost/math/special_functions.hpp>
#include <cassert>
```

2. Asserting for infinity and NaN can be done like this:

```
template <class T>
void check_float_inputs(T value) {
    assert(!boost::math::isinf(value));
    assert(!boost::math::isnan(value));
```

3. Use the following code to change the sign:

```
    if (boost::math::signbit(value)) {
        value = boost::math::changesign(value);
    }

    // ...
} // check_float_inputs
```

That’s it! Now, we may check that `check_float_inputs(std::sqrt(-1.0))` and `check_float_inputs(std::numeric_limits<double>::max() * 2.0)` will trigger asserts.

How it works…

Real types have specific values that cannot be checked using equality operators. For example, if the variable `v` contains NaN, `assert(v != v)` may or may not pass depending on the compiler.

For such cases, `Boost.Math` provides functions that may reliably check for infinity and NaN values.

Step 3 contains the `boost::math::signbit` function, which requires clarification. This function returns a signed bit, which is `1` when the number is negative and `0` when the number is positive. In other words, it returns `true` if the value is negative.

Looking at *step 3*, some readers may ask, why can't we just multiply by `-1` instead of calling `boost::math::changesign`? We can. But, multiplication may work slower than `boost::math::changesign` and is not guaranteed at work for special values. For example, if your code can work with `nan`, the code in *step 3* is able to change the sign of `-nan` and write `nan` to the variable.

The `Boost.Math` library maintainers recommend wrapping math functions from this example in round parentheses to avoid collisions with C macro. It is better to write `(boost::math::isinf)(value)` instead of `boost::math::isinf(value)`.

There’s more…

C99 contains all the functions described in this recipe. Why do we need them in Boost? Well, some compiler vendors think that programmers do not need the full support of C99, so you won’t find those functions in at least one very popular compiler. Another reason is that the `Boost.Math` functions may be used for classes that behave like numbers.

`Boost.Math` is a very fast, portable, and reliable library. **Mathematical special functions** are part of the `Boost.Math` library and some mathematical special functions were accepted into C++17. A `Boost.Math`, however, provides more of them and has highly usable recurrent versions that have better complexities and better suit some of the tasks (like numerical integrations).

See also

Boost’s official documentation contains lots of interesting examples and tutorials that will help you get used to `Boost.Math`. Browse to http://boost.org/libs/math to read about it.

Writing test cases

This recipe and the next one are devoted to auto-testing using the `Boost.Test` library, which is used by many Boost libraries. Let’s get hands-on with it and write some tests for our own class:

```
#include <stdexcept>
struct foo {
    int val_;

    operator int() const;
    bool is_not_null() const;
    void throws() const; // throws(std::logic_error)
};
```

Getting ready

Basic knowledge of C++ is required for this recipe. To compile code of this recipe, define `BOOST_TEST_DYN_LINK` macro and link against the `boost_unit_test_framework` and `boost_system` libraries.

How to do it…

To be honest, there is more than one test library in Boost. We’ll take a look at the most functional one.

1. To use it, we need to define the macro and include the following header:

```
#define BOOST_TEST_MODULE test_module_name
#include <boost/test/unit_test.hpp>
```

2. Each set of tests must be written in the test case:

```
BOOST_AUTO_TEST_CASE(test_no_1) {
```

3. Checking some function for the `true` result must be done as follows:

```
    foo f1 = {1}, f2 = {2};
    BOOST_CHECK(f1.is_not_null());
```

4. Checking for nonequality must be implemented in the following way:

```
    BOOST_CHECK_NE(f1, f2);
```

5. Checking for an exception being thrown must look like this:

```
    BOOST_CHECK_THROW(f1.throws(), std::logic_error);
} // BOOST_AUTO_TEST_CASE(test_no_1)
```

That’s it! After compilation and linking, we’ll have a binary that automatically tests `foo` and outputs test results in a human-readable format.

How it works…

Writing unit tests is easy. You know how the function works and what result it will produce in specific situations. Therefore, you just check if the expected result is the same as the function's actual output. That's what we did in *step 3*. We know that `f1.is_not_null()` returns `true` and we checked it. At *step 4*, we do know that `f1` is not equal to `f2`, so we checked it too. The call to `f1.throws()` produces the `std::logic_error` exception and we check that an exception of the expected type is thrown.

At *step 2*, we are making a test case—a set of checks to validate correct behavior of the `foo` structure. We may have multiple test cases in a single source file. For example, if we add the following code:

```
BOOST_AUTO_TEST_CASE(test_no_2) {
    foo f1 = {1}, f2 = {2};
    BOOST_REQUIRE_NE(f1, f2);
    // ...
} // BOOST_AUTO_TEST_CASE(test_no_2)
```

This code will run along with the `test_no_1` test case.

The parameter passed to the `BOOST_AUTO_TEST_CASE` macro is just a unique name of the test case that is shown in case of error.

```
Running 2 test cases...
main.cpp(15): error in "test_no_1": check f1.is_not_null() failed
main.cpp(17): error in "test_no_1": check f1 != f2 failed [0 == 0]
main.cpp(19): error in "test_no_1": exception std::logic_error is expected
main.cpp(24): fatal error in "test_no_2": critical check f1 != f2 failed [0 == 0]

*** 4 failures detected in test suite "test_module_name"
```

There is a small difference between the `BOOST_REQUIRE_*` and `BOOST_CHECK_*` macros. If the `BOOST_REQUIRE_*` macro check fails, the execution of the current test case stops and `Boost.Test` runs the next test case. However, failing `BOOST_CHECK_*` does not stop the execution of the current test case.

Step 1 requires additional care. Note the `BOOST_TEST_MODULE` macro definition. This macro must be defined before including the `Boost.Test` headers; otherwise, linking the program will fail. More information can be found in the *See also* section of this recipe.

There’s more…

Some readers may wonder, why did we write `BOOST_CHECK_NE(f1, f2)` in *step 4* instead of `BOOST_CHECK(f1 != f2)`? The answer is simple: the macro at *step 4* provides a more readable and verbose output on older versions of `Boost.Test` library.

C++17 lacks support for unit testing. However, the `Boost.Test` library can be used to test C++17 and pre-C++11 code.

Remember that the more tests you have, the more reliable code you get!

See also

- The *Combining multiple test cases in one test module* recipe contains more information about testing and the `BOOST_TEST_MODULE` macro.
- Refer to Boost's official documentation at http://boost.org/libs/test for a full list of test macros and information about advanced features of `Boost.Test`

Combining multiple test cases in one test module

Writing auto tests is good for your project. However, managing test cases is hard when the project is big and many developers work on it. In this recipe, we'll take a look at how to run individual tests and how to combine multiple test cases in a single module.

Let's pretend that two developers are testing the `foo` structure declared in the `foo.hpp` header and we wish to give them separate source files to write tests to. In that case, both developers won't bother each other and may work in parallel. However, the default test run must execute tests of both developers.

Getting ready

Basic knowledge of C++ is required for this recipe. This recipe partially reuses code from the previous recipe and it also requires the `BOOST_TEST_DYN_LINK` macro defined and linkage against the `boost_unit_test_framework` and `boost_system` libraries.

How to do it…

This recipe uses the code from the previous one. This is a very useful recipe for testing big projects. Do not underestimate it.

1. Of all the headers in `main.cpp` from the previous recipe, leave only these two lines:

```
#define BOOST_TEST_MODULE test_module_name
#include <boost/test/unit_test.hpp>
```

2. Let’s move the test cases from the previous example into two different source files:

```
// developer1.cpp
#include <boost/test/unit_test.hpp>
#include "foo.hpp"
BOOST_AUTO_TEST_CASE(test_no_1) {
    // ...
}
// developer2.cpp
#include <boost/test/unit_test.hpp>
#include "foo.hpp"
BOOST_AUTO_TEST_CASE(test_no_2) {
    // ...
}
```

That’s it! Thus compiling and linking all the sources and both test cases will work on program execution.

How it works…

All the magic is done by the `BOOST_TEST_MODULE` macro. If it is defined before `<boost/test/unit_test.hpp>`, `Boost.Test` thinks that this source file is the main one and all the helper testing infrastructure must be placed in it. Otherwise, only the test macro is be included from `<boost/test/unit_test.hpp>`.

All the `BOOST_AUTO_TEST_CASE` tests will run if you link them with the source file that contains the `BOOST_TEST_MODULE` macro. When working on a big project, each developer may enable compilation and linking only of their own sources. That gives independence from other developers and increases the speed of development - no need to compile alien sources and run alien tests while debugging.

There's more…

The Boost.Test library is good because of its ability to run tests selectively. We may choose what tests to run and pass them as command-line arguments. For example, the following command runs only the test_no_1 test case:

```
./testing_advanced -run=test_no_1
```

The following command runs two test cases:

```
./testing_advanced -run=test_no_1,test_no_2
```

Unfortunately, C++17 standard does not have built-in testing support and it looks like C++20 also won't adopt the classes and methods of Boost.Test.

See also

- The *Writing test cases* recipe contains more information about the `Boost.Test` library. Read Boost's official documentation at http://boost.org/libs/test for more information about `Boost.Test`.
- Brave ones may try to take a look at some of the test cases from the Boost library. Those test cases are allocated in the `libs` sub-folder located in the `boost` folder. For example, `Boost.LexicalCast` tests cases are allocated at `boost_1_XX_0/libs/lexical_cast/test`.

Manipulating images

I’ve left you something really tasty for dessert - Boost’s Generic Image Library or just `Boost.GIL`, which allows you to manipulate images without worrying too much about image formats.

Let’s do something simple and interesting with it. For example, let’s make a program that negates any picture.

Getting ready

This recipe requires basic knowledge of C++, templates, and `Boost.Variant`. The example requires linking against the `png` library.

How to do it…

For simplicity of the example, we'll be working only with PNG images.

1. Let's start by including the header files:

```
#include <boost/gil/gil_all.hpp>
#include <boost/gil/extension/io/png_dynamic_io.hpp>
#include <string>
```

2. Now, we need to define the image types that we wish to work with:

```
int main(nt argc, char *argv[]) {
    typedef boost::mpl::vector<
            boost::gil::gray8_image_t,
            boost::gil::gray16_image_t,
            boost::gil::rgb8_image_t
    > img_types;
```

3. Opening an existing PNG image can be implemented like this:

```
    std::string file_name(argv[1]);
    boost::gil::any_image<img_types> source;
    boost::gil::png_read_image(file_name, source);
```

4. We need to apply the operation to the picture as follows:

```
    boost::gil::apply_operation(
        view(source),
        negate()
    );
```

5. The following code line will help you to write an image:

```
    boost::gil::png_write_view("negate_" + file_name, const_view(source));
```

6. Let's take a look at the modifying operation:

```
struct negate {
    typedef void result_type; // required

    template <class View>
    void operator()(const View& source) const {
        // ...
    }
}; // negate
```

7. The body of `operator()` consists of getting a channel type:

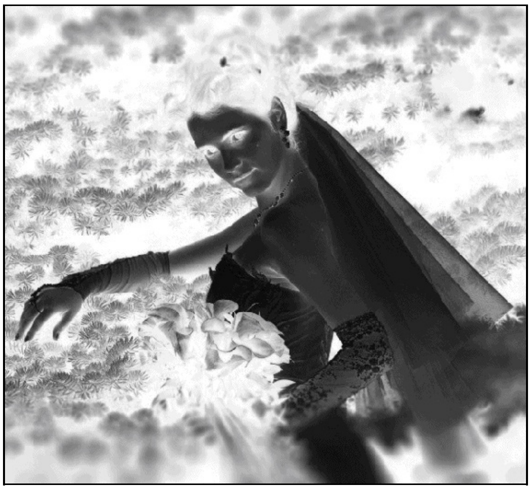
```
typedef typename View::value_type value_type;
typedef typename boost::gil::channel_type<value_type>::type channel_t;
```

8. It also iterates through pixels:

```
const std::size_t channels = boost::gil::num_channels<View>::value;
const channel_t max_val = (std::numeric_limits<channel_t>::max)();

for (unsigned int y = 0; y < source.height(); ++y) {
```

```
            for (unsigned int x = 0; x < source.width(); ++x) {
                for (unsigned int c = 0; c < channels; ++c) {
                    source(x, y)[c] = max_val - source(x, y)[c];
                }
            }
        }
```

Now let’s see the results of our program:



The previous picture is the negative of the one that follows:

How it works…

At *step 2*, we are describing the types of images we wish to work with. These images are gray images with 8 and 16 bits per pixel and RGB pictures with 8 bits per pixel.

The `boost::gil::any_image<img_types>` class is a kind of `Boost.Variant` that may hold an image of one of the `img_types` variables. As you may have already guessed, `boost::gil::png_read_image` reads images into image variables.

The `boost::gil::apply_operation` function at *step 4* is almost equal to `boost::apply_visitor` from the `Boost.Variant` library. Note the usage of `view(source)`. The `boost::gil::view` function constructs a light wrapper around the image that interprets it as a two-dimensional array of pixels.

Do you remember that for `Boost.Variant` we were deriving visitors from `boost::static_visitor`? When we are using GIL's version of variant, we need to make a `result_type` typedef inside `visitor`. You can see it in *step 6*.

A little bit of theory: images consist of points called **pixels**. One image has pixels of the same type. However, pixels of different images may differ in channels count and color bits for a single channel. A channel represents a primary color. In the case of an RGB image, we have a pixel consisting of three channels - red, green, and blue. In the case of a gray image, we have a single channel representing gray.

Back to our image. At *step 2*, we described the types of images we wish to work with. At *step 3*, one of those image types is read from file and stored in the source variable. At *step 4*, the `operator()` method of the `negate` visitor is instantiated for all image types.

At *step 7*, we can see how to get the channel type from the image view.

At *step 8*, we iterate through pixels and channels and negate them. Negation is done via `max_val - source(x, y)[c]` and the result is written back to the image view.

We write an image back at *step 5*.

There’s more…

C++17 has no built-in methods to work with images. There is ongoing work to add 2D drawing to the C++ standard library, though it’s a kind of orthogonal functionality.

The `Boost.GIL` library is fast and efficient. The compilers optimize its code well and we may even help the optimizer using some of the `Boost.GIL` methods to unroll loops. But this chapter talks about only some of the library basics, so it is time to stop.

See also

- More information about `Boost.GIL` can be found at Boost's official documentation at http://boost.org/libs/gil
- See the *Storing multiple chosen types in a variable/container* recipe in Chapter 1, *Starting to Write Your Application*, for more information about the `Boost.Variant` library
- See the https://isocpp.org/ for more news on C++
- Take a look at https://stdcpp.ru/ for discussion of C++ proposals on Russian