Preface

What you will learn

This book is an introduction to the Boost C++ Libraries. The Boost C++ Libraries complement the standard library. Because the Boost C++ Libraries are based on the standard, they are implemented using state-of-the-art C++. They are platform independent and are supported on many operating systems, including Windows and Linux, by a large developer community.

The Boost C++ Libraries enable you to boost your productivity as a C++ developer. For example, you can benefit from smart pointers that help you to write more reliable code or use one of the many libraries to develop platform-independent network applications. Since the Boost libraries partly anticipate developments in the standard, you can benefit earlier from tools without having to wait for them to become available in the standard library.

What you should know

Since the Boost libraries are based on, and extend, the standard, you should know the standard well. You should understand and be able to use containers, iterators, and algorithms, and ideally you should have heard of concepts such as RAII, function objects, and predicates. The better you know the standard, the more you will benefit from the Boost libraries.

In general, you don’t need any knowledge of template meta programming to use the libraries introduced in this book. The main focus is on libraries that can be learned quickly and easily and that can be immediately of great benefit in your work as a C++ developer.

Many examples use features that were added to the standard with C++11. For example, the keyword `auto` is used to avoid specifying types explicitly. Constructors are called through uniform initialization: variables are initialized, if possible, with a pair of curly brackets instead of parentheses. Many examples use lambda functions to make code shorter and more compact. While you can understand many examples without detailed knowledge of C++11, this book is based on the current standard.

Typographical Conventions

The following text styles are used in this book:

`Monospace font`

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

`Monospace bold font`

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

Bold

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

Italic

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

Examples

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

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

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

Introduction

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

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

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

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

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

Part I. RAII and Memory Management

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

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

Table of Contents

1. Boost.SmartPointers
2. Boost.PointerContainer
3. Boost.ScopeExit
4. Boost.Pool

Chapter I. Boost.SmartPointers

Table of Contents

Sole Ownership
Shared Ownership
Special Smart Pointers

The library Boost.SmartPointers provides various smart pointers. They help you manage dynamically allocated objects, which are anchored in smart pointers that release the dynamically allocated objects in the destructor. Because destructors are executed when the scope of smart pointers ends, releasing dynamically allocated objects is guaranteed. There can't be a memory leak if, for example, you forget to call `delete`.

The standard library has included the smart pointer `std::auto_ptr` since C++98, but since C++11, `std::auto_ptr` has been deprecated. With C++11, new and better smart pointers were introduced in the standard library. `std::shared_ptr` and `std::weak_ptr` originate from Boost.SmartPointers and are called `boost::shared_ptr` and `boost::weak_ptr` in this library. There is no counterpart to `std::unique_ptr`. However, Boost.SmartPointers provides four additional smart pointers – `boost::scoped_ptr`, `boost::scoped_array`, `boost::shared_array`, and `boost::intrusive_ptr` – which are not in the standard library.

Chapter 2. Boost.PointerContainer

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

Example 2.1. Using `boost::ptr_vector`

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

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

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

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

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

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

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

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

With `boost::ptr_set`, the order of the elements depends on the `int` values. `std::set` compares pointers of type `std::unique_ptr` and not the variables the pointers refer to. To make `std::set` sort the elements based on `int` values, the container must be told how to compare elements. In Example 2.2, `boost::indirect_fun` (provided by Boost.PointerContainer) is used. With `boost::indirect_fun`, `std::set` is told that elements

shouldn't be sorted based on pointers of type `std::unique_ptr`, but instead based on the `int` values the pointers refer to. That's why the example displays `1` twice.

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

Example 2.3. Inserters for containers from Boost.PointerContainer

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

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

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

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

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

Exercise

Create a program with multiple objects of a type `animal` with the member variables **`name`**, **`legs`** and **`has_tail`**. Store the objects in a container from Boost.PointerContainer. Sort the container in ascending order based on legs and write all elements to standard output.

Chapter 3. Boost.ScopeExit

The library Boost.ScopeExit makes it possible to use RAII without resource-specific classes.

Example 3.1. Using `BOOST_SCOPE_EXIT`

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

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

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

Boost.ScopeExit provides the macro `BOOST_SCOPE_EXIT`, which can be used to define something that looks like a local function but doesn't have a name. However, it does have a parameter list in parentheses and a block in braces.

The header file `boost/scoped_exit.hpp` must be included to use `BOOST_SCOPE_EXIT`.

The parameter list for the macro contains variables from the outer scope which should be accessible in the block. The variables are passed by copy. To pass a variable by reference, it must be prefixed with an ampersand, as in Example 3.1.

Code in the block can only access variables from the outer scope if the variables are in the parameter list.

`BOOST_SCOPE_EXIT` is used to define a block that will be executed when the scope the block is defined in ends. In Example 3.1 the block defined with `BOOST_SCOPE_EXIT` is executed just before `foo()` returns.

`BOOST_SCOPE_EXIT` can be used to benefit from RAII without having to use resource-specific classes. `foo()` uses `new` to create an `int` variable. In order to free the variable, a block that calls `delete` is defined with `BOOST_SCOPE_EXIT`. This block is guaranteed to be executed even if, for example, the function returns early because of an exception. In Example 3.1, `BOOST_SCOPE_EXIT` is as good as a smart pointer.

Please note that the variable **`i`** is set to 0 at the end of the block defined by `BOOST_SCOPE_EXIT`. **`i`** is then returned by `foo()` and written to the standard output stream in `main()`. However, the example doesn't display 0. **`j`** is set to a random value – namely the address where the `int` variable was before the memory was freed. The block behind `BOOST_SCOPE_EXIT` got a

reference to **i** and freed the memory. But since the block is executed at the end of `foo()`, the assignment of 0 to **i** is too late. The return value of `foo()` is a copy of **i** that gets created before **i** is set to 0.

You can ignore Boost.ScopeExit if you use a C++11 development environment. In that case, you can use RAII without resource-specific classes with the help of lambda functions.

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

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

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

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

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

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

Example 3.2 defines the class `scope_exit` whose constructor accepts a function. This function is called by the destructor. Furthermore, a helper function, `make_scope_exit()`, is defined that makes it possible to instantiate `scope_exit` without having to specify a template parameter.

In `foo()` a lambda function is passed to `make_scope_exit()`. The lambda function looks like the block after `BOOST_SCOPE_EXIT` in Example 3.1: The dynamically allocated `int` variable whose address is stored in **i** is freed with `delete`. Then 0 is assigned to **i**.

The example does the same thing as the previous one. Not only is the `int` variable deleted, but **j** is not set to 0 either when it is written to the standard output stream.

Example 3.3. Peculiarities of `BOOST_SCOPE_EXIT`

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

struct x
{
  int i;

  void foo()
  {
```

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

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

Example 3.3 introduces some peculiarities of `BOOST_SCOPE_EXIT`:

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

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

Exercise

Replace `std::unique_ptr` and the user-defined deleter with `BOOST_SCOPE_EXIT`:

```
#include <string>
#include <memory>
#include <cstdio>

struct CloseFile
{
    void operator()(std::FILE *file)
    {
        std::fclose(file);
    }
};

void write_to_file(const std::string &s)
{
    std::unique_ptr<std::FILE, CloseFile> file{
      std::fopen("hello-world.txt", "a") };
    std::fprintf(file.get(), s.c_str());
}

int main()
{
    write_to_file("Hello, ");
    write_to_file("world!");
}
```

Chapter 4. Boost.Pool

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

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

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

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

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

Example 4.1. Using `boost::simple_segregated_storage`

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

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

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

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

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

The header file `boost/pool/simple_segregated_storage.hpp` must be included to use the class template `boost::simple_segregated_storage`. Example 4.1 passes `std::size_t` as the template parameter. This parameter specifies which type should be used for numbers

passed to member functions of `boost::simple_segregated_storage` to refer, for example, to the size of a segment. The practical relevance of this template parameter is rather low.

More interesting are the member functions called on `boost::simple_segregated_storage`. First, `add_block()` is called to pass a memory block with 1024 bytes to **`storage`**. The memory is provided by the vector **`v`**. The third parameter passed to `add_block()` specifies that the memory block should be partitioned in segments with 256 bytes each. Because the total size of the memory block is 1024 bytes, the memory managed by **`storage`** consists of four segments.

The calls to `malloc()` and `malloc_n()` request memory from **`storage`**. While `malloc()` returns a pointer to a free segment, `malloc_n()` returns a pointer to one or more contiguous segments that provide as many bytes in one block as requested. Example 4.1 requests a block with 512 bytes with `malloc_n()`. This call consumes two segments, since each segment is 256 bytes. After the calls to `malloc()` and `malloc_n()`, **`storage`** has only one unused segment left.

At the end of the example, all segments are released with `free()` and `free_n()`. After these two calls, all segments are available and could be requested again with `malloc()` or `malloc_n()`.

You usually don't use `boost::simple_segregated_storage` directly. Boost.Pool provides other classes that allocate memory automatically without requiring you to allocate memory yourself and pass it to `boost::simple_segregated_storage`.

Example 4.2. Using `boost::object_pool`

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

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

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

  int *j = pool.construct(2);

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

Example 4.2 uses the class `boost::object_pool`, which is defined in `boost/pool/object_pool.hpp`. Unlike `boost::simple_segregated_storage`, `boost::object_pool` knows the type of the objects that will be stored in memory. **`pool`** in Example 4.2 is simple segregated storage for `int` values. The memory managed by **`pool`** consists of segments, each of which is the size of an `int` – 4 bytes for example.

Another difference is that you don't need to provide memory to `boost::object_pool`. `boost::object_pool` allocates memory automatically. In Example 4.2, the call to `malloc()` makes **`pool`** allocate a memory block with space for 32 `int` values. `malloc()` returns a pointer to the first of these 32 segments that an `int` value can fit into exactly.

Please note that `malloc()` returns a pointer of type `int*`. Unlike `boost::simple_segregated_storage` in Example 4.1, no cast operator is required.

`construct()` is similar to `malloc()` but initializes an object via a call to the constructor. In Example 4.2, **`j`** refers to an `int` object initialized with the value 2.

Please note that **`pool`** can return a free segment from the pool of 32 segments when `construct()` is called. The call to `construct()` does not make Example 4.2 request memory from the operating system.

The last member function called in Example 4.2 is `destroy()`, which releases an `int` object.

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

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

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

You can pass two parameters to the constructor of `boost::object_pool`. The first parameter sets the size of the memory block that `boost::object_pool` will allocate when the first segment is requested with a call to `malloc()` or `construct()`. The second parameter sets the maximum size of the memory block to allocate.

If `malloc()` or `construct()` are called so often that all segments in a memory block are used, the next call to one of these member functions will cause `boost::object_pool` to allocate a new memory block, which will be twice as big as the previous one. The size will double each time a new memory block is allocated by `boost::object_pool`. `boost::object_pool` can manage an arbitrary number of memory blocks, but their sizes will grow exponentially. The second constructor parameter lets you limit the growth.

The default constructor of `boost::object_pool` does the same as what the call to the constructor in Example 4.3 does. The first parameter sets the size of the memory block to 32 `int` values. The second parameter specifies that there is no maximum size. If 0 is passed, `boost::object_pool` can double the size of the memory block indefinitely.

The call to `construct()` in Example 4.3 makes **`pool`** allocate a memory block of 32 `int` values. **`pool`** can serve up to 32 calls to `malloc()` or `construct()` without requesting memory from the operating system. If more memory is required, the next memory block to allocate will have space for 64 `int` values.

`get_next_size()` returns the size of the next memory block to allocate. `set_next_size()` lets you set the size of the next memory block. In Example 4.3 `get_next_size()` returns 64. The call to `set_next_size()` changes the size of the next memory block to allocate from 64 to 8 `int`

values. With `set_next_size()` the size of the next memory block can be changed directly. If you only want to set a maximum size, pass it via the second parameter to the constructor.

With `boost::singleton_pool`, Boost.Pool provides a class between `boost::simple_segregated_storage` and `boost::object_pool` (see Example 4.4).

Example 4.4. Using `boost::singleton_pool`

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

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

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

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

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

`boost::singleton_pool` is defined in `boost/pool/singleton_pool.hpp`. This class is similar to `boost::simple_segregated_storage` since it also expects the segment size as a template parameter but not the type of the objects to store. That's why member functions such as `ordered_malloc()` and `malloc()`return a pointer of type `void*`, which must be cast explicitly.

This class is also similar to `boost::object_pool` because it allocates memory automatically. The size of the next memory block and an optional maximum size are passed as template parameters. Here `boost::singleton_pool` differs from `boost::object_pool`: you can't change the size of the next memory block in `boost::singleton_pool` at run time.

You can create multiple objects with `boost::singleton_pool` if you want to manage several memory pools. The first template parameter passed to `boost::singleton_pool` is a *tag*. The tag is an arbitrary type that serves as a name for the memory pool. Example 4.4 uses the structure `int_pool` as a tag to highlight that `singleton_int_pool` is a pool that manages `int` values. Thanks to tags, multiple singletons can manage different memory pools, even if the second template parameter for the size is the same. The tag has no purpose other than creating separate instances of `boost::singleton_pool`.

`boost::singleton_pool` provides two member functions to release memory: `release_memory()` releases all memory blocks that aren't used at the moment, and `purge_memory()` releases all memory blocks – including those currently being used. The call to `purge_memory()` resets `boost::singleton_pool`.

`release_memory()` and `purge_memory()` return memory to the operating system. To return memory to `boost::singleton_pool` instead of the operating system, call member functions such as `free()` or `ordered_free()`.

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

Example 4.5. Using `boost::pool_allocator`

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

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

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

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

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

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

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

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

Example 4.6. Using `boost::fast_pool_allocator`

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

int main()
{
  typedef boost::fast_pool_allocator<int,
    boost::default_user_allocator_new_delete,
```

```
    boost::details::pool::default_mutex,
    64, 128> allocator;

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

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

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

Example 4.6 illustrates which template parameters can be passed to `boost::fast_pool_allocator`. `boost::pool_allocator` accepts the same parameters.

`boost::default_user_allocator_new_delete` is a class that allocates memory blocks with `new` and releases them with `delete[]`. You can also use `boost::default_user_allocator_malloc_free`, which calls `malloc()` and `free()`.

`boost::details::pool::default_mutex` is a type definition that is set to `boost::mutex` or `boost::details::pool::null_mutex`. `boost::mutex` is the default type that supports multiple threads requesting memory from the allocator. If the macro `BOOST_POOL_NO_MT` is defined as in Example 4.6, multithreading support for Boost.Pool is disabled. The allocator in Example 4.6 uses a null mutex.

The last two parameters passed to `boost::fast_pool_allocator` in Example 4.6 set the size of the first memory block and the maximum size of memory blocks to allocate.

Part II. String Handling

The following libraries provide tools to simplify working with strings.

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

Table of Contents

5. Boost.StringAlgorithms
6. Boost.LexicalCast
7. Boost.Format
8. Boost.Regex
9. Boost.Xpressive
10. Boost.Tokenizer
11. Boost.Spirit

Chapter 5. Boost.StringAlgorithms

The Boost.StringAlgorithms library provides many free-standing functions for string manipulation. Strings can be of type `std::string`, `std::wstring`, or any other instance of the class template `std::basic_string`. This includes the string classes `std::u16string` and `std::u32string` introduced with C++11.

The functions are categorized within different header files. For example, functions converting from uppercase to lowercase are defined in `boost/algorithm/string/case_conv.hpp`. Because Boost.StringAlgorithms consists of more than 20 different categories and as many header files, `boost/algorithm/string.hpp` acts as the common header including all other header files for convenience.

Example 5.1. Converting strings to uppercase

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

using namespace boost::algorithm;

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

The function `boost::algorithm::to_upper_copy()` converts a string to uppercase, and `boost::algorithm::to_lower_copy()` converts a string to lowercase. Both functions return a copy of the input string, converted to the specified case. To convert the string in place, use the functions `boost::algorithm::to_upper()` or `boost::algorithm::to_lower()`.

Example 5.1 converts the string “Boost C++ Libraries” to uppercase using `boost::algorithm::to_upper_copy()`. The example writes `BOOST C++ LIBRARIES` to standard output.

Functions from Boost.StringAlgorithms consider locales. Functions like `boost::algorithm::to_upper_copy()` use the global locale if no locale is passed explicitly as a parameter.

Example 5.2. Converting a string to uppercase with a locale

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

using namespace boost::algorithm;

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

```
  std::cout << upper_case1 << '\n';
  std::cout << upper_case2 << '\n';
}
```

Example 5.2 calls `boost::algorithm::to_upper_copy()` twice to convert the Turkish string "Boost C++ kütüphaneleri" to uppercase. The first call to `boost::algorithm::to_upper_copy()` uses the global locale, which in this case is the C locale. In the C locale, there is no uppercase mapping for characters with umlauts, so the output will look like this: `BOOST C++ KüTüPHANELERI`.

The Turkish locale is passed to the second call to `boost::algorithm::to_upper_copy()`. Since this locale does have uppercase equivalents for umlauts, the entire string can be converted to uppercase. Therefore, the second call to `boost::algorithm::to_upper_copy()` correctly converts the string, which looks like this: `BOOST C++ KÜTÜPHANELERI`.

Note

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

Example 5.3. Algorithms to remove characters from a string

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

using namespace boost::algorithm;

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

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

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

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

using namespace boost::algorithm;

int main()
{
```

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

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

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

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

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

using namespace boost::algorithm;

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

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

Example 5.6. Algorithms to replace characters in a string

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

using namespace boost::algorithm;

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

Like the functions for searching strings or removing characters from strings, Boost.StringAlgorithms also provides functions for replacing substrings within a string. These include the following functions: `boost::algorithm::replace_first_copy()`,

`boost::algorithm::replace_nth_copy()`, `boost::algorithm::replace_last_copy()`, `boost::algorithm::replace_all_copy()`, `boost::algorithm::replace_head_copy()` and `boost::algorithm::replace_tail_copy()`. They can be applied in the same way as the functions for searching and removing, except they require an additional parameter – the replacement string (see Example 5.6).

Example 5.7. Algorithms to trim strings

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

using namespace boost::algorithm;

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

To remove spaces on either end of a string, use `boost::algorithm::trim_left_copy()`, `boost::algorithm::trim_right_copy()` and `boost::algorithm::trim_copy()` (see Example 5.7). The global locale determines which characters are considered to be spaces.

Boost.StringAlgorithms lets you provide a predicate as an additional parameter for different functions to determine which characters of the string the function is applied to. The versions with predicates are: `boost::algorithm::trim_right_copy_if()`, `boost::algorithm::trim_left_copy_if()`, and `boost::algorithm::trim_copy_if()`.

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

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

using namespace boost::algorithm;

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

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

Boost.StringAlgorithms provides many helper functions that return commonly used predicates.

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

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

using namespace boost::algorithm;

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

The predicate returned by `boost::algorithm::is_digit()` tests whether a character is numeric. In Example 5.9, `boost::algorithm::is_digit()` is used to remove digits from the string **s**.

Boost.StringAlgorithms also provides helper functions to check whether a character is uppercase or lowercase: `boost::algorithm::is_upper()` and `boost::algorithm::is_lower()`. All of these functions use the global locale by default, unless you pass in a different locale as a parameter.

Besides the predicates that verify individual characters of a string, Boost.StringAlgorithms also offers functions that work with strings instead (see Example 5.10).

Example 5.10. Algorithms to compare strings with others

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

using namespace boost::algorithm;

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

The `boost::algorithm::starts_with()`, `boost::algorithm::ends_with()`, `boost::algorithm::contains()`, and `boost::algorithm::lexicographical_compare()` functions compare two individual strings.

Example 5.11 introduces a function that splits a string into smaller parts.

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

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

using namespace boost::algorithm;

int main()
```

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

With `boost::algorithm::split()`, a given string can be split based on a delimiter. The substrings are stored in a container. The function requires as its third parameter a predicate that tests each character and checks whether the string should be split at the given position. Example 5.11 uses the helper function `boost::algorithm::is_space()` to create a predicate that splits the string at every space character.

Many of the functions introduced in this chapter have versions that ignore the case of the string. These versions typically have the same name, except for a leading “i”. For example, the equivalent function to `boost::algorithm::erase_all_copy()` is `boost::algorithm::ierase_all_copy()`.

Finally, many functions of Boost.StringAlgorithms also support regular expressions. Example 5.12 uses the function `boost::algorithm::find_regex()` to search for a regular expression.

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

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

using namespace boost::algorithm;

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

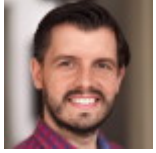
In order to use the regular expression, the program accesses a class called `boost::regex`, which is presented in Chapter 8.

Example 5.12 writes C++ to standard output.

Exercise

Create a program which asks the user to enter his full name. The program should greet the user with “Hello” followed by the user's name followed by an exclamation mark. The user's first- and lastname should start with a capital letter followed by lowercase letters. Furthermore, the user's first- und lastname should be separated with exactly one space. There should be no space before the exclamation mark.

Solutions
theboostcpplibraries.com



Solutions from the expert to all exercises in the book for $9.99

Chapter 6. Boost.LexicalCast

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

Example 6.1. Using `boost::lexical_cast`

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

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

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

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

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

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

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

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

Chapter 7. Boost.Format

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

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

Example 7.1. Formatted output with `boost::format`

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

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

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

Example 7.2. Numbered placeholders with `boost::format`

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

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

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

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

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

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

Example 7.3 uses the manipulator `std::showpos()` on the value that will be associated with "%1%". Therefore, this example will display `+1 2 +1` as output. Because the manipulator `std::showpos()` has been linked to the first data value using `boost::io::group()`, the plus

sign is automatically added whenever this value is displayed. In this case, the format placeholder “%1%” is used twice.

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

Example 7.4. Placeholders with special characters

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

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

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

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

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

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

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

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

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

Example 7.6. Placeholders without numbers

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

int main()
{
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 8. Boost.Regex

Boost.Regex allows you to use *regular expressions* in C++. As the library is part of the standard library since C++11, you don't depend on Boost.Regex if your development environment supports C++11. You can use identically named classes and functions in the namespace `std` if you include the header file `regex`.

The two most important classes in Boost.Regex are `boost::regex` and `boost::smatch`, both defined in `boost/regex.hpp`. The former defines a regular expression, and the latter saves the search results.

Boost.Regex provides three different functions to search for regular expressions.

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

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

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

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

`boost::regex_search()` searches a string for a regular expression.

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

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

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

`boost::regex_search()` expects a reference to an object of type `boost::smatch` as an additional parameter, which is used to store the results. `boost::regex_search()` only searches for groups. That's why Example 8.2 returns two strings based on the two groups found in the regular expression.

The result storage class `boost::smatch` is a container holding elements of type `boost::sub_match`, which can be accessed through an interface similar to the one of `std::vector`. For example, elements can be accessed via `operator[]`.

The class `boost::sub_match` stores iterators to the specific positions in a string corresponding to the groups of a regular expression. Because `boost::sub_match` is derived from `std::pair`, the iterators that reference a particular substring can be accessed with **`first`** and **`second`**. However, to write a substring to the standard output stream, you don't have to access these iterators (see Example 8.2). Using the overloaded operator `operator<<`, the substring can be written directly to standard output.

Please note that because iterators are used to point to matched strings, `boost::sub_match` does not copy them. This implies that results are accessible only as long as the corresponding string, which is referenced by the iterators, exists.

Furthermore, please note that the first element of the container `boost::smatch` stores iterators referencing the string that matches the entire regular expression. The first substring that matches the first group is accessible at index 1.

The third function offered by Boost.Regex is `boost::regex_replace()` (see Example 8.3).

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

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

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

In addition to the search string and the regular expression, `boost::regex_replace()` needs a format that defines how substrings that match individual groups of the regular expression should be replaced. In case the regular expression does not contain any groups, the corresponding substrings are replaced one to one using the given format. Thus, Example 8.3 will output `_Boost_Libraries_`.

`boost::regex_replace()` always searches through the entire string for the regular expression. Thus, the program actually replaces all three spaces with underscores.

Example 8.4. Format with references to groups in regular expressions

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

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

```
  std::cout << boost::regex_replace(s, expr, fmt) << '\n';
}
```

The format can access substrings returned by groups of the regular expression. Example 8.4 uses this technique to swap the first and last word, displaying `Libraries Boost` as a result.

There are different standards for regular expressions and formats. Each of the three functions takes an additional parameter that allows you to select a specific standard. You can also specify whether or not special characters should be interpreted in a specific format or whether the format should replace the complete string that matches the regular expression.

Example 8.5. Flags for formats

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

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

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

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

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

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

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

Example 8.6 displays `Boost` and `Libraries`.

Example 8.7. Accessing groups with `boost::regex_token_iterator`

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

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

You can pass a number as an additional parameter to the constructor of `boost::regex_token_iterator`. If 1 is passed, as in Example 8.7, the iterator returns the first group in the regular expression. Because the regular expression “(\w)\w+” is used, Example 8.7 writes the initials B and L to standard output.

If -1 is passed to `boost::regex_token_iterator`, the regular expression is the delimiter. An iterator initialized with -1 returns substrings that do not match the regular expression.

Example 8.8. Linking a locale to a regular expression

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

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

Example 8.8 links a locale with `imbue()` to **expr**. This is done to apply the regular expression to the string “Boost kütüphaneleri,” which is the Turkish translation of “Boost Libraries.” If umlauts should be parsed as valid letters, the locale must be set – otherwise `boost::regex_match()` returns `false`.

To use a locale of type `std::locale`, **expr** must be based on a class instantiated with the type `boost::cpp_regex_traits`. That’s why Example 8.8 doesn’t use `boost::regex` but instead uses `boost::basic_regex<char, boost::cpp_regex_traits<char>>`. With the second template parameter of `boost::basic_regex`, the parameter for `imbue()` can be defined indirectly. Only with `boost::cpp_regex_traits` can a locale of type `std::locale` be passed to `imbue()`.

Note

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

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

Chapter 9. Boost.Xpressive

Like Boost.Regex, Boost.Xpressive provides functions to search strings using *regular expressions*. However, Boost.Xpressive makes it possible to write down regular expressions as C++ code rather than strings. That makes it possible to check at compile time whether a regular expression is valid or not.

Only Boost.Regex was incorporated into C++11. The standard library doesn't provide any support for writing regular expressions as C++ code.

`boost/xpressive/xpressive.hpp` provides access to most library functions in Boost.Xpressive. For some functions, additional header files must be included. All definitions of the library can be found in the namespace `boost::xpressive`.

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

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

using namespace boost::xpressive;

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

Boost.Xpressive basically provides the same functions as Boost.Regex, except they are defined in the namespace of Boost.Xpressive. `boost::xpressive::regex_match()` compares strings, `boost::xpressive::regex_search()` searches in strings, and `boost::xpressive::regex_replace()` replaces characters in strings. You can see this in Example 9.1, which uses the function `boost::xpressive::regex_match()`, and which looks similar to Example 8.1.

However, there is a fundamental difference between Boost.Xpressive and Boost.Regex. The type of the regular expression in Boost.Xpressive depends on the type of the string being searched. Because **s** is based on `std::string` in Example 9.1, the type of the regular expression must be `boost::xpressive::sregex`. Compare this with Example 9.2, where the regular expression is applied to a string of type `const char*`.

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

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

using namespace boost::xpressive;

int main()
{
  const char *c = "Boost Libraries";
  cregex expr = cregex::compile("\\w+\\s\\w+");
```

```
  std::cout << std::boolalpha << regex_match(c, expr) << '\n';
}
```

For strings of type `const char*`, use the class `boost::xpressive::cregex`. If you use other string types, such as `std::wstring` or `const wchar_t*`, use `boost::xpressive::wsregex` or `boost::xpressive::wcregex`.

You must call the static member function `compile()` for regular expressions written as strings. The member function must be called on the type used for the regular expression.

Boost.Xpressive supports direct initialization of regular expressions that are written as C++ code. The regular expression has to be expressed in the notation supported by Boost.Xpressive (see Example 9.3).

Example 9.3. A regular expression with C++ code

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

using namespace boost::xpressive;

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

The regular expression from Example 9.2, which was written as the string “\w+\s\w+”, is now expressed in Example 9.3 as `+_w >> _s >> +_w`. It is exactly the same regular expression. Both examples search for at least one alphanumeric character followed by one space followed by at least one alphanumeric character.

Boost.Xpressive makes it possible to write regular expressions with C++ code. The library provides objects for character groups. For example, the object **_w** is similar to “\w”. **_s** has the same meaning as “\s”.

While “\w” and “\s” can be written one after another in a string, objects like **_w** and **_s** must be concatenated with an operator. Otherwise, the result wouldn’t be valid C++ code. Boost.Xpressive provides the operator `operator>>`, which is used in Example 9.3.

To express that at least one alphanumeric character should be found, **_w** is prefixed with a plus sign. While the syntax of regular expressions expects that quantifiers are put behind character groups – like with “\w+” – the plus sign must be put in front of **_w**. The plus sign is an unary operator, which in C++ must be put in front of an object.

Boost.Xpressive emulates the rules of regular expressions as much as they can be emulated in C++. However, there are limits. For example, the question mark is a meta character in regular expressions to express that a preceding item is optional. Since the question mark isn’t a valid operator in C++, Boost.Xpressive replaces it with the exclamation mark. A notation like “\w?” becomes `!_w` with Boost.Xpressive because the exclamation mark must be prefixed.

Boost.Xpressive supports actions that can be linked to expressions – something Boost.Regex doesn't support.

Example 9.4. Linking actions to expressions

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

using namespace boost::xpressive;

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

Example 9.4 returns `true` for `boost::xpressive::regex_match()` and writes `Boost` to standard output.

You can link actions to expressions. An action is executed when the respective expression is found. In Example 9.4, the expression `+_w` is linked to the action `*boost::xpressive::ref(it) = _`. The action is a lambda function. The object _ refers to characters found by the expression – in this case the first word in **`s`**. The respective characters are assigned to the iterator **`it`**. Because **`it`** is an iterator of type `std::ostream_iterator`, which has been initialized with **`std::cout`**, `Boost` is written to standard output.

Please note that you must use the function `boost::xpressive::ref()` to wrap the iterator **`it`**. Only then it is possible to assign _ to the iterator. _ is an object provided by Boost.Xpressive in the namespace `boost::xpressive`, which normally couldn't be assigned to an iterator of type `std::ostream_iterator`. Because the assignment happens only when the string "Boost" has been found with `+_w`, `boost::xpressive::ref()` turns the assignment into a *lazy* operation. Although the code in square brackets attached to `+_w` is, according to C++ rules, immediately executed, the assignment to the iterator **`it`** can only occur when the regular expression is used. Thus, `*boost::xpressive::ref(it) = _` isn't executed immediately.

Example 9.4 includes the header file `boost/xpressive/regex_actions.hpp`. This is required because actions aren't available through `boost/xpressive/xpressive.hpp`.

Like Boost.Regex, Boost.Xpressive supports iterators to split a string with regular expressions. The classes `boost::xpressive::regex_token_iterator` and `boost::xpressive::regex_iterator` do this. It is also possible to link a locale to a regular expression to use a locale other than the global one.

Chapter 10. Boost.Tokenizer

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

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

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

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

Boost.Tokenizer defines a class template called `boost::tokenizer` in `boost/tokenizer.hpp`. It expects as a template parameter a class that identifies coherent expressions. Example 10.1 uses the class `boost::char_separator`, which interprets spaces and punctuation marks as separators.

A tokenizer must be initialized with a string of type `std::string`. Using the member functions `begin()` and `end()`, the tokenizer can be accessed like a container. Partial expressions of the string used to initialize the tokenizer are available via iterators. How partial expressions are evaluated depends on the kind of class passed as the template parameter.

Because `boost::char_separator` interprets spaces and punctuation marks as separators by default, Example 10.1 displays `Boost`, `C`, `+`, `+`, and `Libraries`. `boost::char_separator` uses `std::isspace()` and `std::ispunct()` to identify separator characters. Boost.Tokenizer distinguishes between separators that should be displayed and separators that should be suppressed. By default, spaces are suppressed and punctuation marks are displayed.

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

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

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

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

The constructor of `boost::char_separator` accepts a total of three parameters, but only the first one is required. The first parameter describes the individual separators that are suppressed. Example 10.2, like Example 10.1, treats spaces as separators.

The second parameter specifies the separators that should be displayed. If this parameter is omitted, no separators are displayed, and the program will now display `Boost`, `C++` and `Libraries`.

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

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

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

If a plus sign is passed as the second parameter, Example 10.3 behaves like Example 10.1.

The third parameter determines whether or not empty partial expressions are displayed. If two separators are found back-to-back, the corresponding partial expression is empty. By default, these empty expressions are not displayed. Using the third parameter, the default behavior can be changed.

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

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

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

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

Example 10.5. Boost.Tokenizer with wide strings

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

int main()
```

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

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

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

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

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

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

`boost::escaped_list_separator` is used to read multiple values separated by commas. This format is commonly known as CSV (Comma Separated Values). `boost::escaped_list_separator` also handles double quotes and escape sequences. Therefore, the output of Example 10.6 is `Boost` and `C++ Libraries`.

The second class provided is `boost::offset_separator`, which must be instantiated. The corresponding object must be passed to the constructor of `boost::tokenizer` as a second parameter.

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

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

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

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

Chapter II. Boost.Spirit

Table of Contents

API
Parsers
Actions
Attributes
Rules
Grammar

This chapter introduces the library Boost.Spirit. Boost.Spirit is used to develop parsers for text formats. For example, you can use Boost.Spirit to develop a parser to load configuration files. Boost.Spirit can also be used for binary formats, although its usefulness in this respect is limited.

Boost.Spirit simplifies the development of parsers because formats are described with rules. Rules define what a format looks like – Boost.Spirit does the rest. You can compare Boost.Spirit to regular expressions, in the sense that it lets you handle complex processes – pattern searching in the case of regular expressions and parsing for Boost.Spirit – without having to write code to implement that process.

Boost.Spirit expects rules to be described using Parsing Expression Grammar (PEG). PEG is related to Extended Backus-Naur-Form (EBNF). Even if you are not familiar with these languages, the examples in this chapter should be sufficient to get you started.

There are two versions of Boost.Spirit. The first version is known as Spirit.Classic. This version should not be used anymore. The current version is 2.5.2. This is the version introduced in this chapter.

Since version 2.x, Boost.Spirit can be used to generate generators as well as parsers. While parsers read text formats, generators write them. The component of Boost.Spirit that is used to develop parsers is called Spirit.Qi. Spirit.Karma is the component used to develop generators. Namespaces are partitioned accordingly: classes and functions to develop parsers can be found in `boost::spirit::qi` and classes and functions to develop generators can be found in `boost::spirit::karma`.

Besides Spirit.Qi and Spirit.Karma, the library contains a component called Spirit.Lex, which can be used to develop lexers.

This chapter focuses on developing parsers. The examples mainly use classes and functions from `boost::spirit` and `boost::spirit::qi`. For these classes and functions, it is sufficient to include the header file `boost/spirit/include/qi.hpp`.

If you don't want to include a master header file like `boost/spirit/include/qi.hpp`, you can include header files from `boost/spirit/include/` individually. It is important to include header files from this directory only. `boost/spirit/include/` is the interface to the user. Header files in other directories can change in new library versions.

Part III. Containers

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

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

Table of Contents

12. Boost.MultiIndex
13. Boost.Bimap
14. Boost.Array
15. Boost.Unordered
16. Boost.CircularBuffer
17. Boost.Heap
18. Boost.Intrusive
19. Boost.MultiArray

20. Boost.Container

Chapter 12. Boost.MultiIndex

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

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

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

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

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

using namespace boost::multi_index;

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

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

int main()
{
  animal_multi animals;

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

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

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

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

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

The key advantage of containers based on Boost.MultiIndex is that you can access elements via different interfaces. When you define a new container, you can specify the number and type of interfaces. The container in Example 12.1 needs to support searching for animals by name or number of legs, so two interfaces are defined. Boost.MultiIndex calls these interfaces indexes – that's where the library's name comes from.

Interfaces are defined with the help of the class `boost::multi_index::indexed_by`. Each interface is passed as a template parameter. Two interfaces of type `boost::multi_index::hashed_non_unique`, which is defined in `boost/multi_index/hashed_index.hpp`, are used in Example 12.1. Using these interfaces makes the container behave like `std::unordered_set` and look up values using a hash value.

The class `boost::multi_index::hashed_non_unique` is a template as well and expects as its sole parameter a class that calculates hash values. Because both interfaces of the container need to look up animals, one interface calculates hash values for the name, while the other interface does so for the number of legs.

Boost.MultiIndex offers the helper class template `boost::multi_index::member`, which is defined in `boost/multi_index/member.hpp`, to access a member variable. As seen in Example 12.1, several parameters have been specified to let `boost::multi_index::member` know which member variable of `animal` should be accessed and which type the member variable has.

Even though the definition of `animal_multi` looks complicated at first, the class works like a map. The name and number of legs of an animal can be regarded as a key/value pair. The advantage of the container `animal_multi` over a map like `std::unordered_map` is that animals can be looked up by name or by number of legs. `animal_multi` supports two interfaces, one based on the name and one based on the number of legs. The interface determines which member variable is the key and which member variable is the value.

To access a MultiIndex container, you need to select an interface. If you directly access the object **`animals`** using `insert()` or `count()`, the first interface is used. In Example 12.1, this is the hash container for the member variable **`name`**. If you need a different interface, you must explicitly select it.

Interfaces are numbered consecutively, starting at index 0 for the first interface. To access the second interface – as shown in Example 12.1 – call the member function `get()` and pass in the index of the desired interface as the template parameter.

The return value of `get()` looks complicated. It accesses a class of the MultiIndex container called `nth_index` which, again, is a template. The index of the interface to be used must be specified as a template parameter. This index must be the same as the one passed to `get()`. The final step is to access the type definition named `type` of `nth_index`. The value of `type` represents the type of the corresponding interface. The following examples use the keyword `auto` to simplify the code.

Although you do not need to know the specifics of an interface, since they are automatically derived from `nth_index` and `type`, you should still understand what kind of interface is accessed. Since interfaces are numbered consecutively in the container definition, this can be answered easily, since the index is passed to both `get()` and `nth_index`. Thus, **`legs_index`** is a hash interface that looks up animals by legs.

Because data such as names and legs can be keys of the MultiIndex container, they cannot be arbitrarily changed. If the number of legs is changed after an animal has been looked up by name, an interface using legs as a key would be unaware of the change and would not know that a new hash value needs to be calculated.

Just as the keys in a container of type `std::unordered_map` cannot be modified, neither can data stored within a MultiIndex container. Strictly speaking, all data stored in a MultiIndex container is constant. This includes member variables that aren't used by any interface. Even if no interface accesses **`legs`**, **`legs`** cannot be changed.

To avoid having to remove elements from a MultiIndex container and insert new ones, Boost.MultiIndex provides member functions to change values directly. Because these member functions operate on the MultiIndex container itself, and because no element in a container is modified directly, all interfaces will be notified and can calculate new hash values.

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

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

using namespace boost::multi_index;

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

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
```

```
      >
    >,
    hashed_non_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

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

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

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

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

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

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

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

using namespace boost::multi_index;

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

typedef multi_index_container<
  animal,
  indexed_by<
    hashed_non_unique<
      member<
        animal, std::string, &animal::name
      >
```

```
    >,
    hashed_unique<
      member<
        animal, int, &animal::legs
      >
    >
  >
> animal_multi;

int main()
{
  animal_multi animals;

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

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

The container in Example 12.3 uses `boost::multi_index::hashed_unique` as the second interface. That means no two animals with the same number of legs can be stored in the container because the hash values would be the same.

The example tries to store a dog, which has the same number of legs as the already stored cat. Because this violates the requirement of having unique hash values for the second interface, the dog will not be stored in the container. Therefore, when searching for animals with four legs, the program displays `1`, because only the cat was stored and counted.

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

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

using namespace boost::multi_index;

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

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

int main()
{
  animal_multi animals;
```

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

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

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

Example 12.4 introduces the last three interfaces of Boost.MultiIndex: `boost::multi_index::sequenced`, `boost::multi_index::ordered_non_unique`, and `boost::multi_index::random_access`.

The interface `boost::multi_index::sequenced` allows you to treat a MultiIndex container like a list of type `std::list`. Elements are stored in the given order.

With the interface `boost::multi_index::ordered_non_unique`, objects are automatically sorted. This interface requires that you specify a sorting criterion when defining the container. Example 12.4 sorts objects of type `animal` by the number of legs using the helper class `boost::multi_index::member`.

`boost::multi_index::ordered_non_unique` provides special member functions to find specific ranges within the sorted values. Using `lower_bound()` and `upper_bound()`, the program searches for animals that have at least four and no more than eight legs. Because they require elements to be sorted, these member functions are not provided by other interfaces.

The final interface introduced is `boost::multi_index::random_access`, which allows you to treat the MultiIndex container like a vector of type `std::vector`. The two most prominent member functions are `operator[]` and `at()`.

`boost::multi_index::random_access` includes `boost::multi_index::sequenced`. With `boost::multi_index::random_access`, all member functions of `boost::multi_index::sequenced` are available as well.

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

Example 12.5 introduces two more key extractors.

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

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

```
#include <boost/multi_index/mem_fun.hpp>
#include <string>
#include <utility>
#include <iostream>

using namespace boost::multi_index;

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

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

int main()
{
  animal_multi animals;

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

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

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

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

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

Boost.MultiIndex offers two more key extractors: `boost::multi_index::global_fun` and `boost::multi_index::composite_key`. The former can be used for free-standing or static

member functions, and the latter allows you to design a key extractor made up of several other key extractors.

Exercise

Define the class `animals_container` with Boost.MultiIndex:

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

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

class animals_container
{
public:
    void add(animal a)
    {
        // TODO: Implement this member function.
    }

    const animal *find_by_name(const std::string &name) const
    {
        // TODO: Implement this member function.
        return nullptr;
    }

    std::vector<animal> find_by_legs(int from, int to) const
    {
        // TODO: Implement this member function.
        return {};
    }

    std::vector<animal> find_by_tail(bool has_tail) const
    {
        // TODO: Implement this member function.
        return {};
    }
};

int main()
{
    animals_container animals;
    animals.add({ "cat", 4, true });
    animals.add({ "ant", 6, false });
    animals.add({ "spider", 8, false });
    animals.add({ "shark", 0, false });

    const animal *a = animals.find_by_name("cat");
    if (a)
        std::cout << "cat has " << a->legs << " legs\n";

    auto animals_with_6_to_8_legs = animals.find_by_legs(6, 9);
    for (auto a : animals_with_6_to_8_legs)
        std::cout << a.name << " has " << a.legs << " legs\n";

    auto animals_without_tail = animals.find_by_tail(false);
    for (auto a : animals_without_tail)
        std::cout << a.name << " has no tail\n";
}
```

Chapter 13. Boost.Bimap

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

Example 13.1. Using `boost::bimap`

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

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

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

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

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

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

Example 13.2. Accessing relations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`boost::bimaps::unconstrained_set_of` can be used to disable one side of `boost::bimap`. In Example 13.5, `boost::bimap` behaves like `std::map`. You can't access **`right`** to search for animals by legs.

Example 13.5 illustrates another reason why it can make sense to prefer `boost::bimap` over `std::map`. Since Boost.Bimap is based on Boost.MultiIndex, member functions from Boost.MultiIndex are available. Example 13.5 modifies a key using `modify_key()` – something that is not possible with `std::map`.

Note how the key is modified. A new value is assigned to the current key using **`boost::bimaps::_key`**, which is a placeholder that is defined in `boost/bimap/support/lambda.hpp`.

`boost/bimap/support/lambda.hpp` also defines **`boost::bimaps::_data`**. When calling the member function `modify_data()`, **`boost::bimaps::_data`** can be used to modify a value in a container of type `boost::bimap`.

Exercise

Implement the class `animals_container` with Boost.Bimap:

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

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

    animal(std::string n, int l) : name(n), legs(l) {}
};

class animals_container
{
public:
    void add(animal a)
    {
        // TODO: Implement this member function.
    }

    boost::optional<animal> find_by_name(const std::string &name) const
    {
        // TODO: Implement this member function.
        return {};
    }

    std::vector<animal> find_by_legs(int from, int to) const
    {
        // TODO: Implement this member function.
        return {};
    }
};

int main()
{
    animals_container animals;
    animals.add({ "cat", 4 });
    animals.add({ "ant", 6 });
    animals.add({ "spider", 8 });
    animals.add({ "shark", 0 });

    auto shark = animals.find_by_name("shark");
    if (shark)
        std::cout << "shark has " << shark->legs << " legs\n";

    auto animals_with_4_to_6_legs = animals.find_by_legs(4, 7);
    for (auto animal : animals_with_4_to_6_legs)
        std::cout << animal.name << " has " << animal.legs << " legs\n";
}
```

Chapter 14. Boost.Array

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

With `boost::array`, an array can be created that exhibits the same properties as a C array. In addition, `boost::array` conforms to the requirements of C++ containers, which makes handling such an array as easy as handling any other container. In principle, one can think of `boost::array` as the container `std::vector`, except the number of elements in `boost::array` is constant.

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

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

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

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

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

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

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

As seen in Example 14.1, using `boost::array` is fairly simple and needs no additional explanation since the member functions called have the same meaning as their counterparts from `std::vector`.

Chapter 15. Boost.Unordered

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

Example 15.1. Using `boost::unordered_set`

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

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

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

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

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

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

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

Example 15.2. Using `boost::unordered_map`

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

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

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

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

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

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

Example 15.2 uses `boost::unordered_map` to store the names and the number of legs for several animals. Once again, `boost::unordered_map` could be replaced with `std::unordered_map`.

Example 15.3. User-defined type with Boost.Unordered

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

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

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

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

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

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

In Example 15.3 elements of type `animal` are stored in a container of type `boost::unordered_set`. Because the hash function of `boost::unordered_set` doesn't know the class `animal`, hash values can't be automatically calculated for elements of this type. That's why a hash function must be defined – otherwise the example can't be compiled.

The name of the hash function to define is `hash_value()`. It must expect as its sole parameter an object of the type the hash value should be calculated for. The type of the return value of `hash_value()` must be `std::size_t`.

The function `hash_value()` is automatically called when the hash value has to be calculated for an object. This function is defined for various types in the Boost libraries, including `std::string`. For user-defined types like `animal`, it must be defined by the developer.

Usually, the definition of `hash_value()` is rather simple: Hash values are created by accessing the member variables of an object one after another. This is done with the function `boost::hash_combine()`, which is provided by Boost.Hash and defined in `boost/functional/hash.hpp`. You don't have to include this header file if you use Boost.Unordered because all containers from this library access Boost.Hash to calculate hash values.

In addition to defining `hash_value()`, you need to make sure two objects can be compared using `==`. That's why the operator `operator==` is overloaded for `animal` in Example 15.3.

The hash containers from the C++11 standard library use a hash function from the header file `functional`. The hash containers from Boost.Unordered expect the hash function `hash_value()`. Whether you use Boost.Hash within `hash_value()` doesn't matter. Boost.Hash makes sense because functions like `boost::hash_combine()` make it easier to calculate hash values from multiple member variables step by step. However, this is only an implementation detail of `hash_value()`. Apart from the different hash functions used, the hash containers from Boost.Unordered and the standard library are basically equivalent.

Chapter 16. Boost.CircularBuffer

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

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

A circular buffer makes sense when the amount of available memory is limited, and you need to prevent a container from growing arbitrarily big. Another example is continuous data flow where old data becomes irrelevant as new data becomes available. Memory is automatically reused by overwriting old data.

To use the circular buffer from Boost.CircularBuffer, include the header file `boost/circular_buffer.hpp`. This header file defines the class `boost::circular_buffer`.

Example 16.1. Using `boost::circular_buffer`

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

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

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

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

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

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

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

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

`boost::circular_buffer` is a template and must be instantiated with a type. For instance, the circular buffer **`cb`** in Example 16.1 stores numbers of type `int`.

The capacity of the circular buffer is specified when instantiating the class, not through a template parameter. The default constructor of `boost::circular_buffer` creates a buffer with a capacity of zero elements. Another constructor is available to set the capacity. In Example 16.1, the buffer **`cb`** has a capacity of three elements.

The capacity of a circular buffer can be queried by calling `capacity()`. In Example 16.1, `capacity()` will return 3.

The capacity is not equivalent to the number of stored elements. While the return value of `capacity()` is constant, `size()` returns the number of elements in the buffer, which may be different. The return value of `size()` will always be between 0 and the capacity of the circular buffer.

Example 16.1 returns 0 the first time `size()` is called since the buffer does not contain any data. After calling `push_back()` three times, the buffer contains three elements, and the second call to `size()` will return 3. Calling `push_back()` again does not cause the buffer to grow. The three new numbers overwrite the previous three. Therefore, `size()` will return 3 when called for the third time.

As a verification, the stored numbers are written to standard output at the end of Example 16.1. The output contains the numbers 3, 4, and 5 since the previously stored numbers have been overwritten.

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

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

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

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

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

  circular_buffer::array_range ar1, ar2;

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

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

  cb.linearize();

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

Example 16.2 uses the member functions `is_linearized()`, `array_one()`, `array_two()` and `linearize()`, which do not exist in other containers. These member functions clarify the internals of the circular buffer.

A circular buffer is essentially comparable to `std::vector`. Because the beginning and end are well defined, a vector can be treated as a conventional C array. That is, memory is contiguous, and the first and last elements are always at the lowest and highest memory address. However, a circular buffer does not offer such a guarantee.

Even though it may sound strange to talk about the beginning and end of a circular buffer, they do exist. Elements can be accessed via iterators, and `boost::circular_buffer` provides member functions such as `begin()` and `end()`. While you don't need to be concerned about the position of the beginning and end when using iterators, the situation becomes a bit more complicated when accessing elements using regular pointers, unless you use `is_linearized()`, `array_one()`, `array_two()`, and `linearize()`.

The member function `is_linearized()` returns `true` if the beginning of the circular buffer is at the lowest memory address. In this case, all the elements in the buffer are stored consecutively from beginning to the end at increasing memory addresses, and elements can be accessed like a conventional C array.

If `is_linearized()` returns `false`, the beginning of the circular buffer is not at the lowest memory address, which is the case in Example 16.2. While the first three elements 0, 1, and 2 are stored in exactly this order, calling `push_back()` for the fourth time will overwrite the number 0 with the number 3. Because 3 is the last element added by a call to `push_back()`, it is now the new end of the circular buffer. The beginning is now the element with the number 1, which is stored at the next higher memory address. This means elements are no longer stored consecutively at increasing memory addresses.

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

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

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

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

Boost.CircularBuffer offers an additional class called `boost::circular_buffer_space_optimized`. This class is also defined in

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

Chapter 17. Boost.Heap

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

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

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

using namespace boost::heap;

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

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

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

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

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

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

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

using namespace boost::heap;

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

  binomial_heap<int> bh2;
  bh2.push(4);
  bh.merge(bh2);

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

Example 17.2 introduces the class `boost::heap::binomial_heap`. In addition to allowing you to iterate over elements in priority order, it also lets you merge priority queues. Elements from one queue can be added to another queue.

The example calls `merge()` on the queue **bh**. The queue **bh2** is passed as a parameter. The call to `merge()` moves the number 4 from **bh2** to **bh**. After the call, **bh** contains four numbers, and **bh2** is empty.

The `for` loop calls `ordered_begin()` and `ordered_end()` on **bh**. `ordered_begin()` returns an iterator that iterates from high priority elements to low priority elements. Thus, Example 17.2 writes the numbers 4, 3, 2, and 1 in order to standard output.

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

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

using namespace boost::heap;

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

  bh.update(handle, 4);

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

`boost::heap::binomial_heap` lets you change elements after they have been added to the queue. Example 17.3 saves a handle returned by `push()`, making it possible to access the number 2 stored in **bh**.

`update()` is a member function of `boost::heap::binomial_heap` that can be called to change an element. Example 17.3 calls the member function to replace 2 with 4. Afterwards, the element with the highest priority, now 4, is fetched with `top()`.

In addition to `update()`, `boost::heap::binomial_heap` provides other member functions to change elements. The member functions `increase()` or `decrease()` can be called if you know in advance whether a change will result in a higher or lower priority. In Example 17.3, the call to `update()` could be replaced with a call to `increase()` since the number is increased from 2 to 4.

Boost.Heap provides additional priority queues whose member functions mainly differ in their runtime complexity. For example, you can use the class `boost::heap::fibonacci_heap` if you want the member function `push()` to have a constant runtime complexity. The documentation on Boost.Heap provides a table with an overview of the runtime complexities of the various classes and functions.

Chapter 18. Boost.Intrusive

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

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

The advantages are paid for with more complicated code because preconditions must be met to store objects in intrusive containers. You cannot store objects of arbitrary types in intrusive containers. For example, you cannot put strings of type `std::string` in an intrusive container; instead you must use containers from the standard library.

Example 18.1 prepares a class `animal` to allow objects of this type to be stored in an intrusive list.

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

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

using namespace boost::intrusive;

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

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

  typedef list<animal> animal_list;
  animal_list animals;

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

  a1.name = "dog";

  for (const animal &a : animals)
```

```
    std::cout << a.name << '\n';
}
```

In a list, an element is always accessed from another element, usually using a pointer. If an intrusive list is to store objects of type `animal` without dynamic memory allocation, pointers must exist somewhere to concatenate elements.

To store objects of type `animal` in an intrusive list, the class must provide the variables required by the intrusive list to concatenate elements. Boost.Intrusive provides *hooks* – classes from which the required variables are inherited. To allow objects of the type `animal` to be stored in an intrusive list, `animal` must be derived from the class `boost::intrusive::list_base_hook`.

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

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

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

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

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

Example 18.2. Removing and destroying dynamically allocated objects

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

using namespace boost::intrusive;

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

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

```
  typedef list<animal> animal_list;
  animal_list animals;

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

  animals.pop_back();
  delete a3;

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

Example 18.2 creates an object of type `animal` with `new` and inserts it to the list **`animals`**. If you want to destroy the object with `delete` when you don't need it anymore, you must remove it from the list. Make sure that you remove the object from the list before you destroy it – the order is important. Otherwise, the pointers in the elements of the intrusive container might refer to a memory location that no longer contains an object of type `animal`.

Because intrusive containers neither allocate nor free memory, objects stored in an intrusive container continue to exist when the intrusive container is destroyed.

Since removing elements from intrusive containers doesn't automatically destroy them, the containers provide non-standard extensions. `pop_back_and_dispose()` is one such member function.

Example 18.3. Removing and destroying with `pop_back_and_dispose()`

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

using namespace boost::intrusive;

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

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

  typedef list<animal> animal_list;
  animal_list animals;

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

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

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

`pop_back_and_dispose()` removes an element from a list and destroys it. Because intrusive containers don't know how an element should be destroyed, you need to pass to `pop_back_and_dispose()` a function or function object that does know how to destroy the element. `pop_back_and_dispose()` will remove the object from the list, then call the function or function object and pass it a pointer to the object to be destroyed. Example 18.3 passes a lambda function that calls `delete`.

In Example 18.3, only the third element in **`animals`** can be removed with `pop_back_and_dispose()`. The other elements in the list haven't been created with `new` and, thus, must not be destroyed with `delete`.

Boost.Intrusive supports another mechanism to link removing and destroying of elements.

Example 18.4. Removing and destroying with auto unlink mode

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

using namespace boost::intrusive;

typedef link_mode<auto_unlink> mode;

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

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

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

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

  delete a3;

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

Hooks support a parameter to set a link mode. The link mode is set with the class template `boost::intrusive::link_mode`. If `boost::intrusive::auto_unlink` is passed as a template parameter, the auto unlink mode is selected.

The auto unlink mode automatically removes an element from an intrusive container when it is destroyed. Example 18.4 writes only `cat` and `shark` to standard output.

The auto unlink mode can only be used if the member function `size()`, which is provided by all intrusive containers, has no *constant complexity*. By default, it has constant complexity, which means: the time it takes for `size()` to return the number of elements doesn't depend on how many elements are stored in a container. Switching constant complexity on or off is another option to optimize performance.

To change the complexity of `size()`, use the class template `boost::intrusive::constant_time_size`, which expects either `true` or `false` as a template parameter. `boost::intrusive::constant_time_size` can be passed as a second template parameter to intrusive containers, such as `boost::intrusive::list`, to set the complexity for `size()`.

Now that we've seen that intrusive containers support link mode and that there is an option to set the complexity for `size()`, it might seem as though there is still much more to discover, but there actually isn't. There are, for example, only three link modes supported, and auto unlink mode is the only one you need to know. The default mode used if you don't pick a link mode is good enough for all other use cases.

Furthermore, there are no options for other member functions. There are no other classes, other than `boost::intrusive::constant_time_size`, that you need to learn about.

Example 18.5 introduces a hook mechanism using another intrusive container: `boost::intrusive::set`.

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

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

using namespace boost::intrusive;

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

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

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

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

  for (const animal &a : animals)
```

```
    std::cout << a.name << '\n';
}
```

There are two ways to add a hook to a class: either derive the class from a hook or define the hook as a member variable. While the previous examples derived a class from `boost::intrusive::list_base_hook`, Example 18.5 uses the class `boost::intrusive::set_member_hook` to define a member variable.

Please note that the name of the member variable doesn't matter. However, the hook class you use depends on the intrusive container. For example, to define a hook as a member variable for an intrusive list, use `boost::intrusive::list_member_hook` instead of `boost::intrusive::set_member_hook`.

Intrusive containers have different hooks because they have different requirements for elements. However, you can use different several hooks to allow objects to be stored in multiple intrusive containers. `boost::intrusive::any_base_hook` and `boost::intrusive::any_member_hook` let you store objects in any intrusive container. Thanks to these classes, you don't need to derive from multiple hooks or define multiple member variables as hooks.

Intrusive containers expect hooks to be defined in base classes by default. If a member variable is used as a hook, as in Example 18.5, the intrusive container has to be told which member variable to use. That's why both `animal` and the type `hook` are passed to `boost::intrusive::set`. `hook` is defined with `boost::intrusive::member_hook`, which is used whenever a member variable serves as a hook. `boost::intrusive::member_hook` expects the element type, the type of the hook, and a pointer to the member variable as template parameters.

Example 18.5 writes `shark`, `cat`, and `spider`, in that order, to standard output.

In addition to the classes `boost::intrusive::list` and `boost::intrusive::set` introduced in this chapter, Boost.Intrusive also provides, for example, `boost::intrusive::slist` for singly linked lists and `boost::intrusive::unordered_set` for hash containers.

Chapter 19. Boost.MultiArray

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

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

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

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

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

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

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

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

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

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

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

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

Example 19.2. Views and subarrays of a two-dimensional array

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

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

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

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

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

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

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

Example 19.2 creates a two-dimensional array. The number of elements in the first dimension is set to 2 and for the second dimension set to 6. Think of the array as a table with two rows and six columns.

The first row of the table will contain the word Boost. Since only five letters need to be stored for this word, a *view* is created which spans exactly five elements of the array.

A view, which is based on the class `boost::multi_array::array_view`, lets you access a part of an array and treat that part as though it were a separate array.

`boost::multi_array::array_view` is a template that expects the number of dimensions in the view as a template parameter. In Example 19.2 the number of dimensions for the view is 1. Because the array **a** has two dimensions, one dimension is ignored. To save the word Boost, a one-dimensional array is sufficient; more dimensions would be confusing.

As with `boost::multi_array`, the number of dimensions is passed in as a template parameter, and the size of each dimension is set at runtime. However, with `boost::multi_array::array_view` this isn't done with **`boost::extents`**. Instead it's done with **`boost::indices`**, which is another global object provided by Boost.MultiArray.

As with **`boost::extents`**, indexes must be passed to **`boost::indices`**. While only numbers may be passed to **`boost::extents`**, **`boost::indices`** accepts also ranges. These are defined using `boost::multi_array_types::index_range`.

In Example 19.2, the first parameter passed to **`boost::indices`** isn't a range, it's the number 0. When a number is passed, you cannot use `boost::multi_array_types::index_range`. In the example, the view will take the first dimension of **a** – the one with index 0.

For the second parameter, `boost::multi_array_types::index_range` is used to define a range. By passing 0 and 5 to the constructor, the first five elements of the first dimension of **a** are made available. The range starts at index 0 and ends at index 5 – excluding the element at index 5. The sixth element in the first dimension is ignored.

Thus, **view** is a one-dimensional array consisting of five elements – the first five elements in the first row of **a**. When **view** is accessed to copy a string with `std::memcpy()` and reverse the elements with `std::reverse()`, this relation doesn't matter. Once the view is created, it acts like an independent array with five elements.

When `operator[]` is called on an array of type `boost::multi_array`, the return value depends on the number of dimensions. In Example 19.1, the operator returns `char` elements because the array accessed is one dimensional.

In Example 19.2, **a** is a two-dimensional array. Thus, `operator[]` returns a subarray rather than a `char` element. Because the type of the subarray isn't public, `boost::multi_array::reference` must be used. This type isn't identical to `boost::multi_array::array_view`, even if the subarray behaves like a view. A view must be defined explicitly and can span arbitrary parts of an array, whereas a subarray is automatically returned by `operator[]` and spans all elements in every dimension.

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

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

int main()
{
  char c[12] =
  {
    't', 's', 'o', 'o', 'B', '\0',
    'C', '+', '+', '\0', '\0', '\0'
  };

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

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

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

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

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

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

Chapter 20. Boost.Container

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

Boost.Container provides additional advantages:

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

Example 20.1 illustrates incomplete types.

Note

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

Example 20.1. Recursive containers with Boost.Container

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

using namespace boost::container;

struct animal
{
  vector<animal> children;
};

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

The class `animal` has a member variable **`children`** of type `boost::container::vector<animal>`. `boost::container::vector` is defined in the header file `boost/container/vector.hpp`. Thus, the type of the member variable **`children`** is based

on the class `animal`, which defines the variable **`children`**. At this point, `animal` hasn't been defined completely. While the standard doesn't require containers from the standard library to support incomplete types, recursive containers are explicitly supported by Boost.Container. Whether containers defined by the standard library can be used recursively is implementation dependent.

Example 20.2. Using `boost::container::stable_vector`

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

using namespace boost::container;

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

Boost.Container provides containers in addition to the well-known containers from the standard library. Example 20.2 introduces the container `boost::container::stable_vector`, which behaves similarly to `std::vector`, except that if `boost::container::stable_vector` is changed, all iterators and references to existing elements remain valid. This is possible because elements aren't stored contiguously in `boost::container::stable_vector`. It is still possible to access elements with an index even though elements are not stored next to each other in memory.

Boost.Container guarantees that the reference **`i`** in Example 20.2 remains valid when the first element in the vector is erased. The example displays **`1`**.

Please note that neither `boost::container::stable_vector` nor other containers from this library support C++11 initializer lists. In Example 20.2 **`v`** is initialized with two elements both set to 1.

`boost::container::stable_vector` is defined in `boost/container/stable_vector.hpp`.

Additional containers provided by Boost.Container are `boost::container::flat_set`, `boost::container::flat_map`, `boost::container::slist`, and `boost::container::static_vector`:

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

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

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

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

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

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

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

Part IV. Data Structures

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

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

Table of Contents

21. Boost.Optional
22. Boost.Tuple
23. Boost.Any
24. Boost.Variant
25. Boost.PropertyTree
26. Boost.DynamicBitset
27. Boost.Tribool
28. Boost.CompressedPair

Chapter 21. Boost.Optional

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

Example 21.1. Special values to denote optional return values

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

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

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

Example 21.1 uses the function `get_even_random_number()`, which should return an even random number. It does this in a rather naive fashion by calling the function `std::rand()` from the standard library. If `std::rand()` generates an even random number, that number is returned by `get_even_random_number()`. If the generated random number is odd, -1 is returned.

In this example, -1 means that no even random number could be generated. Thus, `get_even_random_number()` can't guarantee that an even random number is returned. The return value is optional.

Many functions use special values like -1 to denote that no result can be returned. For example, the member function `find()` of the class `std::string` returns the special value **`std::string::npos`** if a substring can't be found. Functions whose return value is a pointer often return 0 to indicate that no result exists.

Boost.Optional provides `boost::optional`, which makes it possible to clearly mark optional return values.

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

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

using boost::optional;

optional<int> get_even_random_number()
```

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

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

In Example 21.2 the return value of `get_even_random_number()` has a new type, `boost::optional<int>`. `boost::optional` is a template that must be instantiated with the actual type of the return value. `boost/optional.hpp` must be included for `boost::optional`.

If `get_even_random_number()` generates an even random number, the value is returned directly, automatically wrapped in an object of type `boost::optional<int>`, because `boost::optional` provides a non-exclusive constructor. If `get_even_random_number()` does not generate an even random number, an empty object of type `boost::optional<int>` is returned. The return value is created with a call to the default constructor.

`main()` checks whether **`i`** is empty. If it isn't empty, the number stored in **`i`** is accessed with `operator*`. `boost::optional` appears to work like a pointer. However, you should not think of `boost::optional` as a pointer because, for example, values in `boost::optional` are copied by the copy constructor while a pointer does not copy the value it points to.

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

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

using boost::optional;

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

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

Example 21.3 introduces other useful member functions of `boost::optional`. This class provides a special constructor that takes a condition as the first parameter. If the condition is true, an object of type `boost::optional` is initialized with the second parameter. If the condition is false, an empty object of type `boost::optional` is created. Example 21.3 uses this constructor in the function `get_even_random_number()`.

With `is_initialized()` you can check whether an object of type `boost::optional` is not empty. Boost.Optional speaks about initialized and uninitialized objects – hence, the name of the member function `is_initialized()`. The member function `get()` is equivalent to `operator*`.

Example 21.4. Various helper functions of Boost.Optional

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

using namespace boost;

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

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

Boost.Optional provides free-standing helper functions such as `boost::make_optional()` and `boost::get_optional_value_or()` (see Example 21.4). `boost::make_optional()` can be called to create an object of type `boost::optional`. If you want a default value to be returned when `boost::optional` is empty, you can call `boost::get_optional_value_or()`.

The function `boost::get_optional_value_or()` is also provided as a member function of `boost::optional`. It is called `get_value_or()`.

Along with `boost/optional/optional_io.hpp`, Boost.Optional provides a header file with overloaded stream operators, which let you write objects of type `boost::optional` to, for example, standard output.

Chapter 22. Boost.Tuple

The library Boost.Tuple provides a class called `boost::tuple`, which is a generalized version of `std::pair`. While `std::pair` can only store exactly two values, `boost::tuple` lets you choose how many values to store.

The standard library has provided the class `std::tuple` since C++11. If you work with a development environment supporting C++11, you can ignore Boost.Tuple because `boost::tuple` and `std::tuple` are identical.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A tuple can also contain references, as shown in Example 22.4.

Example 22.4. Tuples with references

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

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

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

Usually, you can use `std::ref()` from the C++11 standard library instead of `boost::ref()`. However, Example 22.4 uses `boost::ref()` because only Boost.Ref provides an operator to write to standard output.

`std::pair` uses the member variables **`first`** and **`second`** to provide access. Because a tuple does not have a fixed number of elements, access must be handled differently.

Example 22.5. Reading elements of a tuple

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

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

There are two ways to access values in a tuple. You can call the member function `get()`, or you can pass the tuple to the free-standing function `boost::get()`. In both cases, the index of the corresponding element in the tuple must be provided as a template parameter. Example 22.5 accesses the first element of the tuple **a** in both cases and, thus, displays `cat` twice.

Specifying an invalid index results in a compiler error because index validity is checked at compile time.

The member function `get()` and the free-standing function `boost::get()` both return a reference that allows you to change a value inside a tuple.

Example 22.6. Writing elements of a tuple

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

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

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

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

Example 22.7. Comparing tuples

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 22.10. Unpacking return values of a function from a tuple

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

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

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

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

Chapter 23. Boost.Any

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

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

Example 23.1. Using `boost::any`

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

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

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

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

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

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

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

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

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

Example 23.3. Accessing values with `boost::any_cast`

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

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

```
  std::cout << boost::any_cast<double>(a) << '\n';
  a = true;
  std::cout << std::boolalpha << boost::any_cast<bool>(a) << '\n';
}
```

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

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

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

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

Example 23.4 throws an exception because the template parameter of type `float` does not match the type `int` stored in **a**. The program would also throw an exception if `short` or `long` were used as the template parameter.

Because `boost::bad_any_cast` is derived from `std::bad_cast`, `catch` handlers can catch exceptions of this type, too.

To check whether or not a variable of type `boost::any` contains information, use the member function `empty()`. To check the type of the stored information, use the member function `type()`.

Example 23.5. Checking type of currently stored value

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

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

Example 23.5 uses both `empty()` and `type()`. While `empty()` returns a boolean value, the return value of `type()` is of type `std::type_info`, which is defined in the header file `typeinfo`.

Example 23.6 shows how to obtain a pointer to the value stored in a `boost::any` variable using `boost::any_cast`.

Example 23.6. Accessing values through a pointer

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

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

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

Chapter 24. Boost.Variant

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

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

Example 24.1. Using `boost::variant`

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

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

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

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

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

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

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

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

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

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

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

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

For type-safe access, Boost.Variant provides a function called `boost::apply_visitor()`.

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

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

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

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

As its first parameter, `boost::apply_visitor()` expects an object of a class derived from `boost::static_visitor`. This class must overload `operator()` for every type used by the `boost::variant` variable it acts on. Consequently, the operator is overloaded three times in Example 24.4 because **v** supports the types `double`, `char`, and `std::string`.

`boost::static_visitor` is a template. The type of the return value of `operator()` must be specified as a template parameter. If the operator does not have a return value, a template parameter is not required, as seen in the example.

The second parameter passed to `boost::apply_visitor()` is a `boost::variant` variable.

`boost::apply_visitor()` automatically calls the `operator()` for the first parameter that matches the type of the value currently stored in the second parameter. This means that the sample program uses different overloaded operators every time `boost::apply_visitor()` is invoked – first the one for `double`, followed by the one for `char`, and finally the one for `std::string`.

The advantage of `boost::apply_visitor()` is not only that the correct operator is called automatically. In addition, `boost::apply_visitor()` ensures that overloaded operators have been provided for every type supported by `boost::variant` variables. If one of the three overloaded operators had not been defined, the code could not be compiled.

If overloaded operators are equivalent in functionality, the code can be simplified by using a template (see Example 24.5).

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

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

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

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

Because `boost::apply_visitor()` ensures code correctness at compile time, it should be preferred over `boost::get()`.

Chapter 25. Boost.PropertyTree

With the class `boost::property_tree::ptree`, Boost.PropertyTree provides a tree structure to store key/value pairs. Tree structure means that a trunk exists with numerous branches that have numerous twigs. A file system is a good example of a tree structure. File systems have a root directory with subdirectories that themselves can have subdirectories and so on.

To use `boost::property_tree::ptree`, include the header file `boost/property_tree/ptree.hpp`. This is a master header file, so no other header files need to be included for Boost.PropertyTree.

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

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

using boost::property_tree::ptree;

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

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

Example 25.1 uses `boost::property_tree::ptree` to store a path to a directory. This is done with a call to `put()`. This member function expects two parameters because `boost::property_tree::ptree` is a tree structure that saves key/value pairs. The tree doesn't just consist of branches and twigs, a value must be assigned to each branch and twig. In Example 25.1 the value is "20 files".

The first parameter passed to `put()` is more interesting. It is a path to a directory. However, it doesn't use the backlash, which is the common path separator on Windows. It uses the dot.

You need to use the dot because it's the separator Boost.PropertyTree expects for keys. The parameter "C:.Windows.System" tells **`pt`** to create a branch called C: with a branch called Windows that has another branch called System. The dot creates the nested structure of branches. If "C:\Windows\System" had been passed as the parameter, **`pt`** would only have one branch called C:\Windows\System.

After the call to `put()`, **`pt`** is accessed to read the stored value "20 files" and write it to standard output. This is done by jumping from branch to branch – or directory to directory.

To access a subbranch, you call `get_child()`, which returns a reference to an object of the same type `get_child()` was called on. In Example 25.1, this is a reference to `boost::property_tree::ptree`. Because every branch can have subbranches, and because there is no structural difference between higher and lower branches, the same type is used.

The third call to `get_child()` retrieves the `boost::property_tree::ptree`, which represents the directory System. `get_value()` is called to read the value that was stored at the beginning of the example with `put()`.

Please note that `get_value()` is a function template. You pass the type of the return value as a template parameter. That way `get_value()` can do an automatic type conversion.

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

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

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

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

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

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

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

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

The `for` loop in Example 25.2 reads the number of files in all subdirectories of C:\Windows to calculate a total. As a result, the example displays `70`. The example doesn't access objects of

type `ptree` directly. Instead it iterates over elements of type `std::pair<std::string, ptree>`. **`first`** contains the key of the current branch. That is System and Cursors in Example 25.2. **`second`** provides access to an object of type `ptree`, which represents the possible subdirectories. In the example, only the values assigned to System and Cursors are read. As in Example 25.1, the member function `get_value()` is called.

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

Example 25.3. Accessing data with a translator

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

struct string_to_int_translator
{
  typedef std::string internal_type;
  typedef int external_type;

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

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

  string_to_int_translator tr;
  int files =
    pt.get<int>(ptree::path_type{"c:\\windows\\system", '\\'}, tr) +
    pt.get<int>(ptree::path_type{"c:\\windows\\cursors", '\\'}, tr);
  std::cout << files << '\n';
}
```

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

Unlike Example 25.1, `get_child()` isn't called multiple times to access subbranches. Just as `put()` can be used to store a value in a subbranch directly, a value from a subbranch can be read with `get()`. The key is defined the same way – for example using `boost::property_tree::iptree::path_type`.

Like `get_value()`, `get()` is a function template. You have to pass the type of the return value as a template parameter. Boost.PropertyTree does an automatic type conversion.

To convert types, Boost.PropertyTree uses *translators*. The library provides a few translators out of the box that are based on streams and can convert types automatically.

Example 25.3 defines the translator `string_to_int_translator`, which converts a value of type `std::string` to `int`. The translator is passed as an additional parameter to `get()`. Because the translator is just used to read, it only defines one member function, `get_value()`. If you want to use the translator for writing, too, then you would need to define a member function `put_value()` and then pass the translator as an additional parameter to `put()`.

`get_value()` returns a value of the type that is used in **`pt`**. However, because a type conversion doesn't always succeed, `boost::optional` is used. If a value is stored in Example 25.3 that can't be converted to an `int` with `std::strtol()`, an empty object of type `boost::optional` will be returned.

Please note that a translator must also define the two types `internal_type` and `external_type`. If you need to convert types when storing data, define `put_value()` similar to `get_value()`.

If you modify Example 25.3 to store the value "20" instead of value "20 files", `get_value()` can be called without passing a translator. The translators provided by Boost.PropertyTree can convert from `std::string` to `int`. However, the type conversion only succeeds when the entire string can be converted. The string must not contain any letters. Because `std::strtol()` can do a type conversion as long as the string starts with digits, the more liberal translator `string_to_int_translator` is used in Example 25.3.

Example 25.4. Various member functions of `boost::property_tree::ptree`

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

using boost::property_tree::ptree;

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

  boost::optional<std::string> c = pt.get_optional<std::string>("C:");
  std::cout << std::boolalpha << c.is_initialized() << '\n';

  pt.put_child("D:.Program Files", ptree{"50 files"});
  pt.add_child("D:.Program Files", ptree{"60 files"});

  ptree d = pt.get_child("D:");
  for (const std::pair<std::string, ptree> &p : d)
    std::cout << p.second.get_value<std::string>() << '\n';

  boost::optional<ptree&> e = pt.get_child_optional("E:");
  std::cout << e.is_initialized() << '\n';
}
```

You can call the member function `get_optional()` if you want to read the value of a key, but you aren't sure if the key exists. `get_optional()` returns the value in an object of type `boost::optional`. The object is empty if the key wasn't found. Otherwise, `get_optional()` works the same as `get()`.

It might seem like `put_child()` and `add_child()` are the same as `put()`. The difference is that `put()` creates only a key/value pair while `put_child()` and `add_child()` insert an entire subtree. Note that an object of type `boost::property_tree::ptree` is passed as the second parameter to `put_child()` and `add_child()`.

The difference between `put_child()` and `add_child()` is that `put_child()` accesses a key if that key already exists, while `add_child()` always inserts a new key into the tree. That's why the tree in Example 25.4 has two keys called "D:.Program Files". Depending on the use case, this can be confusing. If a tree represents a file system, there shouldn't be two identical paths. You have to avoid inserting identical keys if you don't want duplicates in a tree.

Example 25.4 displays the value of the keys below "D:" in the `for` loop. The example writes `50 files` and `60 files` to standard output, which proves there are two identical keys called "D:.Program Files".

The last member function introduced in Example 25.4 is `get_child_optional()`. This function is used like `get_child()`. `get_child_optional()` returns an object of type `boost::optional`. You call `boost::optional` if you aren't sure whether a key exists.

Example 25.5. Serializing a `boost::property_tree::ptree` in the JSON format

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

using namespace boost::property_tree;

int main()
{
  ptree pt;
  pt.put("C:.Windows.System", "20 files");
  pt.put("C:.Windows.Cursors", "50 files");

  json_parser::write_json("file.json", pt);

  ptree pt2;
  json_parser::read_json("file.json", pt2);

  std::cout << std::boolalpha << (pt == pt2) << '\n';
}
```

Boost.PropertyTree does more than just provide structures to manage data in memory. As can be seen in Example 25.5, the library also provides functions to save a `boost::property_tree::ptree` in a file and load it from a file.

The header file `boost/property_tree/json_parser.hpp` provides access to the functions `boost::property_tree::json_parser::write_json()` and `boost::property_tree::json_parser::read_json()`. These functions make it possible to

save and load a `boost::property_tree::ptree` serialized in the JSON format. That way you can support configuration files in the JSON format.

If you want to call functions that store a `boost::property_tree::ptree` in a file or load it from a file, you must include header files such as `boost/property_tree/json_parser.hpp`. It isn't sufficient to only include `boost/property_tree/ptree.hpp`.

In addition to the functions `boost::property_tree::json_parser::write_json()` and `boost::property_tree::json_parser::read_json()`, Boost.PropertyTree provides functions for additional data formats. You use `boost::property_tree::ini_parser::write_ini()` and `boost::property_tree::ini_parser::read_ini()` from `boost/property_tree/ini_parser.hpp` to support INI-files. With `boost::property_tree::xml_parser::write_xml()` and `boost::property_tree::xml_parser::read_xml()` from `boost/property_tree/xml_parser.hpp`, data can be loaded and stored in XML format. With `boost::property_tree::info_parser::write_info()` and `boost::property_tree::info_parser::read_info()` from `boost/property_tree/info_parser.hpp`, you can access another format that was developed and optimized to serialize trees from Boost.PropertyTree.

None of the supported formats guarantees that a `boost::property_tree::ptree` will look the same after it has been saved and reloaded. For example, the JSON format can lose type information because `boost::property_tree::ptree` can't distinguish between `true` and "true". The type is always the same. Even if the various functions make it easy to save and load a `boost::property_tree::ptree`, don't forget that Boost.PropertyTree doesn't support the formats completely. The main focus of the library is on the structure `boost::property_tree::ptree` and not on supporting various data formats.

Exercise

Create a program that loads this JSON-file and writes the names of all animals to standard output. If "all" is set to `true` the program should not only write the names but all properties of all animals to standard output:

```
{
  "animals": [
    {
      "name": "cat",
      "legs": 4,
      "has_tail": true
    },
    {
      "name": "spider",
      "legs": 8,
      "has_tail": false
    }
  ],
  "log": {
    "all": true
  }
}
```

Chapter 26. Boost.DynamicBitset

The library Boost.DynamicBitset provides the class `boost::dynamic_bitset`, which is used like `std::bitset`. The difference is that the number of bits for `std::bitset` must be specified at compile time, whereas the number of bits for `boost::dynamic_bitset` is specified at run time.

To use `boost::dynamic_bitset`, include the header file `boost/dynamic_bitset.hpp`.

Example 26.1. Using `boost::dynamic_bitset`

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

int main()
{
  boost::dynamic_bitset<> db{3, 4};

  db.push_back(true);

  std::cout.setf(std::ios::boolalpha);
  std::cout << db.size() << '\n';
  std::cout << db.count() << '\n';
  std::cout << db.any() << '\n';
  std::cout << db.none() << '\n';

  std::cout << db[0].flip() << '\n';
  std::cout << ~db[3] << '\n';
  std::cout << db << '\n';
}
```

`boost::dynamic_bitset` is a template that requires no template parameters when instantiated; default types are used in that case. More important are the parameters passed to the constructor. In Example 26.1, the constructor creates **db** with 3 bits. The second parameter initializes the bits; in this case, the number 4 initializes the most significant bit – the bit on the very left.

The number of bits inside an object of type `boost::dynamic_bitset` can be changed at any time. The member function `push_back()` adds another bit, which will become the most significant bit. Calling `push_back()` in Example 26.1 causes **db** to contain 4 bits, of which the two most significant bits are set. Therefore, **db** stores the number 12.

You can decrease the number of bits by calling the member function `resize()`. Depending on the parameter passed to `resize()`, bits will either be added or removed.

`boost::dynamic_bitset` provides member functions to query data and access individual bits. The member functions `size()` and `count()` return the number of bits and the number of bits currently set, respectively. `any()` returns `true` if at least one bit is set, and `none()` returns `true` if no bit is set.

To access individual bits, use array syntax. A reference to an internal class is returned that represents the corresponding bit and provides member functions to manipulate it. For example, the member function `flip()` toggles the bit. Bitwise operators such as `operator~` are available

as well. Overall, the class `boost::dynamic_bitset` offers the same bit manipulation functionality as `std::bitset`.

Like `std::bitset`, `boost::dynamic_bitset` does not support iterators.

Chapter 27. Boost.Tribool

The library Boost.Tribool provides the class `boost::logic::tribool`, which is similar to `bool`. However, while `bool` can distinguish two states, `boost::logic::tribool` handles three.

To use `boost::logic::tribool`, include the header file `boost/logic/tribool.hpp`.

Example 27.1. Three states of `boost::logic::tribool`

```
#include <boost/logic/tribool.hpp>
#include <iostream>

using namespace boost::logic;

int main()
{
  tribool b;
  std::cout << std::boolalpha << b << '\n';

  b = true;
  b = false;
  b = indeterminate;
  if (b)
    ;
  else if (!b)
    ;
  else
    std::cout << "indeterminate\n";
}
```

A variable of type `boost::logic::tribool` can be set to `true`, `false`, or `indeterminate`. The default constructor initializes the variable to `false`. That's why Example 27.1 writes `false` first.

The `if` statement in Example 27.1 illustrates how to evaluate **b** correctly. You have to check for `true` and `false` explicitly. If the variable is set to `indeterminate`, as in the example, the `else` block will be executed.

Boost.Tribool also provides the function `boost::logic::indeterminate()`. If you pass a variable of type `boost::logic::tribool` that is set to `indeterminate`, this function will return `true`. If the variable is set to `true` or `false`, it will return `false`.

Example 27.2. Logical operators with `boost::logic::tribool`

```
#include <boost/logic/tribool.hpp>
#include <boost/logic/tribool_io.hpp>
#include <iostream>

using namespace boost::logic;

int main()
{
  std::cout.setf(std::ios::boolalpha);

  tribool b1 = true;
  std::cout << (b1 || indeterminate) << '\n';
  std::cout << (b1 && indeterminate) << '\n';

  tribool b2 = false;
  std::cout << (b2 || indeterminate) << '\n';
```

```
  std::cout << (b2 && indeterminate) << '\n';

  tribool b3 = indeterminate;
  std::cout << (b3 || b3) << '\n';
  std::cout << (b3 && b3) << '\n';
}
```

You can use logical operators with variables of type `boost::logic::tribool`, just as you can with variables of type `bool`. In fact, this is the only way to process variables of type `boost::logic::tribool` because the class doesn't provide any member functions.

Example 27.2 returns `true` for `b1 || indeterminate`, `false` for `b2 && indeterminate`, and `indeterminate` in all other cases. If you look at the operations and their results, you will notice that `boost::logic::tribool` behaves as one would expect intuitively. The documentation on Boost.Tribool also contains tables that show which operations lead to which results.

Example 27.2 also illustrates how the values `true`, `false`, and `indeterminate` are written to standard output with variables of type `boost::logic::tribool`. The header file `boost/logic/tribool_io.hpp` must be included and the flag **`std::ios::boolalpha`** must be set for standard output.

Boost.Tribool also provides the macro `BOOST_TRIBOOL_THIRD_STATE`, which lets you substitute another value for `indeterminate`. For example, you could use `dontknow` instead of `indeterminate`.

Chapter 28. Boost.CompressedPair

Boost.CompressedPair provides `boost::compressed_pair`, a class that behaves like `std::pair`. However, if one or both template parameters are empty classes, `boost::compressed_pair` consumes less memory. `boost::compressed_pair` uses a technique known as empty base class optimization.

To use `boost::compressed_pair`, include the header file `boost/compressed_pair.hpp`.

Example 28.1. Reduced memory requirements with `boost::compressed_pair`

```
#include <boost/compressed_pair.hpp>
#include <utility>
#include <iostream>

struct empty {};

int main()
{
  std::pair<int, empty> p;
  std::cout << sizeof(p) << '\n';

  boost::compressed_pair<int, empty> cp;
  std::cout << sizeof(cp) << '\n';
}
```

Example 28.1 illustrates this by using `boost::compressed_pair` for **cp** and `std::pair` for **p**. When compiled using Visual C++ 2013 and run on a 64-bit Windows 7 system, the example returns 4 for `sizeof(cp)` and 8 for `sizeof(p)`.

Please note that there is another difference between `boost::compressed_pair` and `std::pair`: the values stored in `boost::compressed_pair` are accessed through the member functions `first()` and `second()`. `std::pair` uses two identically named member variables instead.

Part V. Algorithms

The following libraries provide algorithms that complement the algorithms from the standard library.

- Boost.Algorithm collects and provides useful algorithms.
- Boost.Range also provides algorithms, but more important, it defines a new concept called *range*, which should make using algorithms easier.
- Boost.Graph is specialized for graphs and provides algorithms such as finding the shortest path between two points.

A few libraries that contain algorithms are introduced in other parts of the book. For example, you will find algorithms for strings in the library Boost.StringAlgorithms, which is introduced in Part II

Table of Contents

29. Boost.Algorithm
30. Boost.Range
31. Boost.Graph

Chapter 29. Boost.Algorithm

Boost.Algorithm provides algorithms that complement the algorithms from the standard library. Unlike Boost.Range, Boost.Algorithm doesn't introduce new concepts. The algorithms defined by Boost.Algorithm resemble the algorithms from the standard library.

Please note that there are numerous algorithms provided by other Boost libraries. For example, you will find algorithms to process strings in Boost.StringAlgorithms. The algorithms provided by Boost.Algorithm are not bound to particular classes, such as `std::string`. Like the algorithms from the standard library, they can be used with any container.

Example 29.1. Testing for exactly one value with `boost::algorithm::one_of_equal()`

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

using namespace boost::algorithm;

int main()
{
  std::array<int, 6> a{{0, 5, 2, 1, 4, 3}};
  auto predicate = [](int i){ return i == 4; };
  std::cout.setf(std::ios::boolalpha);
  std::cout << one_of(a.begin(), a.end(), predicate) << '\n';
  std::cout << one_of_equal(a.begin(), a.end(), 4) << '\n';
}
```

`boost::algorithm::one_of()` tests whether a condition is met exactly once. The condition to test is passed as a predicate. In Example 29.1 the call to `boost::algorithm::one_of()` returns `true` since the number 4 is stored exactly once in **a**.

To test elements in a container for equality, call `boost::algorithm::one_of_equal()`. You don't pass a predicate. Instead, you pass a value to compare to `boost::algorithm::one_of_equal()`. In Example 29.1 the call to `boost::algorithm::one_of_equal()` also returns `true`.

`boost::algorithm::one_of()` complements the algorithms `std::all_of()`, `std::any_of()`, and `std::none_of()`, which were added to the standard library with C++11. However, Boost.Algorithm provides the functions `boost::algorithm::all_of()`, `boost::algorithm::any_of()`, and `boost::algorithm::none_of()` for developers whose development environment doesn't support C++11. You will find these algorithms in the header files `boost/algorithm/cxx11/all_of.hpp`, `boost/algorithm/cxx11/any_of.hpp`, and `boost/algorithm/cxx11/none_of.hpp`.

Boost.Algorithm also defines the following functions: `boost::algorithm::all_of_equal()`, `boost::algorithm::any_of_equal()`, and `boost::algorithm::none_of_equal()`.

Boost.Algorithm provides more algorithms from the C++11 standard library. For example, you have access to `boost::algorithm::is_partitioned()`, `boost::algorithm::is_permutation()`, `boost::algorithm::copy_n()`,

`boost::algorithm::find_if_not()` and `boost::algorithm::iota()`. These functions work like the identically named functions from the C++11 standard library and are provided for developers who don't use C++11. However, Boost.Algorithm provides a few function variants that could be useful for C++11 developers, too.

Example 29.2. More variants of C++11 algorithms

```
#include <boost/algorithm/cxx11/iota.hpp>
#include <boost/algorithm/cxx11/is_sorted.hpp>
#include <boost/algorithm/cxx11/copy_if.hpp>
#include <vector>
#include <iterator>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v;
  iota_n(std::back_inserter(v), 10, 5);
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_increasing(v) << '\n';
  std::ostream_iterator<int> out{std::cout, ","};
  copy_until(v, out, [](int i){ return i > 12; });
}
```

Boost.Algorithm provides the C++11 algorithm `boost::algorithm::iota()` in the header file `boost/algorithm/cxx11/iota.hpp`. This function generates sequentially increasing numbers. It expects two iterators for the beginning and end of a container. The elements in the container are then overwritten with sequentially increasing numbers.

Instead of `boost::algorithm::iota()`, Example 29.2 uses `boost::algorithm::iota_n()`. This function expects one iterator to write the numbers to. The number of numbers to generate is passed as a third parameter to `boost::algorithm::iota_n()`.

`boost::algorithm::is_increasing()` and `boost::algorithm::is_sorted()` are defined in the header file `boost/algorithm/cxx11/is_sorted.hpp`. `boost::algorithm::is_increasing()` has the same function as `boost::algorithm::is_sorted()`, but the function name expresses more clearly that the function checks that values are in increasing order. The header file also defines the related function `boost::algorithm::is_decreasing()`.

In Example 29.2, **v** is passed directly to `boost::algorithm::is_increasing()`. All functions provided by Boost.Algorithm have a variant that operates based on ranges. Containers can be passed directly to these functions.

`boost::algorithm::copy_until()` is defined in `boost/algorithm/cxx11/copy_if.hpp`. This is another variant of `std::copy()`. Boost.Algorithm also provides `boost::algorithm::copy_while()`.

Example 29.2 displays `true` as a result from `boost::algorithm::is_increasing()`, and `boost::algorithm::copy_until()` writes the numbers `10`, `11`, and `12` to standard output.

Example 29.3. C++14 algorithms from Boost.Algorithm

```
#include <boost/algorithm/cxx14/equal.hpp>
#include <boost/algorithm/cxx14/mismatch.hpp>
#include <vector>
#include <iostream>

using namespace boost::algorithm;

int main()
{
  std::vector<int> v{1, 2};
  std::vector<int> w{1, 2, 3};
  std::cout.setf(std::ios::boolalpha);
  std::cout << equal(v.begin(), v.end(), w.begin(), w.end()) << '\n';
  auto pair = mismatch(v.begin(), v.end(), w.begin(), w.end());
  if (pair.first != v.end())
    std::cout << *pair.first << '\n';
  if (pair.second != w.end())
    std::cout << *pair.second << '\n';
}
```

Besides the algorithms from the C++11 standard library, Boost.Algorithm also defines algorithms that will very likely be added to the standard library with C++14. Example 29.3 uses new variants of two of these functions, `boost::algorithm::equal()` and `boost::algorithm::mismatch()`. In contrast to the identically named functions that have been part of the standard library since C++98, four iterators, rather than three, are passed to these new functions. The algorithms in Example 29.3 don't expect the second sequence to contain as many elements as the first sequence.

While `boost::algorithm::equal()` returns a `bool`, `boost::algorithm::mismatch()` returns two iterators in a `std::pair`. **`first`** and **`second`** refer to the elements in the first and second sequence that are the first ones mismatching. These iterators may also refer to the end of a sequence.

Example 29.3 writes `false` and `3` to standard output. `false` is the return value of `boost::algorithm::equal()`, `3` the third element in **`w`**. Because the first two elements in **`v`** and **`w`** are equal, `boost::algorithm::mismatch()` returns, in **`first`**, an iterator to the end of **`v`** and, in **`second`**, an iterator to the third element of **`w`**. Because **`first`** refers to the end of **`v`**, the iterator isn't de-referenced, and there is no output.

Example 29.4. Using `boost::algorithm::hex()` and `boost::algorithm::unhex()`

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

using namespace boost::algorithm;

int main()
{
  std::vector<char> v{'C', '+', '+'};
  hex(v, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string s = "C++";
  std::cout << hex(s) << '\n';
```

```
  std::vector<char> w{'4', '3', '2', 'b', '2', 'b'};
  unhex(w, std::ostream_iterator<char>{std::cout, ""});
  std::cout << '\n';

  std::string t = "432b2b";
  std::cout << unhex(t) << '\n';
}
```

Example 29.4 uses the two functions `boost::algorithm::hex()` and `boost::algorithm::unhex()`. These functions are designed after the identically named functions from the database system MySQL. They convert characters to hexadecimal values or hexadecimal values to characters.

Example 29.4 passes the vector **v** with the characters “C”, “+”, and “+” to `boost::algorithm::hex()`. This function expects an iterator as the second parameter to write the hexadecimal values to. The example writes `43` for “C” and `2B` (twice) for the two instances of “+” to standard output. The second call to `boost::algorithm::hex()` does the same thing except that “C++” is passed as a string and “432B2B” is returned as a string.

`boost::algorithm::unhex()` is the opposite of `boost::algorithm::hex()`. If the array **w** from Example 29.4 is passed with six hexadecimal values, each of the three pairs of values is interpreted as ASCII-Code. The same happens with the second call to `boost::algorithm::unhex()` when six hexadecimal values are passed as a string. In both cases `C++` is written to standard output.

Boost.Algorithm provides even more algorithms. For example, there are several string matching algorithms that search text efficiently. The documentation contains an overview of all available algorithms.

Exercise

Use a function from Boost.Algorithm to assign the numbers 51 to 56 in ascending order to an array with six elements. Interpret the numbers in the array as hexadecimal values, convert them to characters and write the result to standard output.

Chapter 30. Boost.Range

Table of Contents

Algorithms
Adaptors
Helper Classes and Functions

Boost.Range is a library that, on the first sight, provides algorithms similar to those provided by the standard library. For example, you will find the function `boost::copy()`, which does the same thing as `std::copy()`. However, `std::copy()` expects two parameters while `boost::copy()` expects a range.

Chapter 31. Boost.Graph

Table of Contents

Vertices and Edges
Algorithms
Containers

Boost.Graph provides tools to work with graphs. Graphs are two-dimensional point clouds with any number of lines between points. A subway map is a good example of a graph. Subway stations are points, which are connected by subway lines.

The graph theory is the field of mathematics that researches graphs. Graph theory tries to answer questions such as how to determine the shortest path between two points. Auto navigation systems have to solve that problem to guide drivers to their desired location using the shortest path. Graphs are very important in practice because many problems can be modelled with them.

Boost.Graph provides containers to define graphs. However, even more important are the algorithms Boost.Graph offers to operate on graphs, for example, to find the shortest path. This chapter introduces you to the containers and algorithms in Boost.Graph.

Part VI. Communication

The following libraries facilitate communication with other programs.

- Boot.Asio is for communicating over networks. Boost.Asio does not just support network operations. Asio stands for asynchronous input/output. You can use this library to process data asynchronously, for example, when your program communicates with devices, such as network cards, that can handle tasks concurrently with code executed in your program.
- Boost.Interprocess is for communicating through shared memory.

Table of Contents

32. Boost.Asio
33. Boost.Interprocess

Chapter 32. Boost.Asio

Table of Contents

I/O Services and I/O Objects
Scalability and Multithreading
Network programming
Coroutines
Platform-specific I/O Objects

This chapter introduces the library Boost.Asio. Asio stands for asynchronous input/output. This library makes it possible to process data asynchronously. Asynchronous means that when operations are initiated, the initiating program does not need to wait for the operation to end. Instead, Boost.Asio notifies a program when an operation has ended. The advantage is that other operations can be executed concurrently.

Boost.Thread is another library that makes it possible to execute operations concurrently. The difference between Boost.Thread and Boost.Asio is that with Boost.Thread, you access resources inside of a program, and with Boost.Asio, you access resources outside of a program. For example, if you develop a function which needs to run a time-consuming calculation, you can call this function in a thread and make it execute on another CPU core. Threads allow you to access and use CPU cores. From the point of view of your program, CPU cores are an internal resource. If you want to access external resources, you use Boost.Asio.

Network connections are an example of external resources. If data has to be sent or received, a network card is told to execute the operation. For a send operation, the network card gets a pointer to a buffer with the data to send. For a receive operation the network card gets a pointer to a buffer it should fill with the data being received. Since the network card is an external resource for your program, it can execute the operations independently. It only needs time – time you could use in your program to execute other operations. Boost.Asio allows you to use the available devices more efficiently by benefiting from their ability to execute operations concurrently.

Sending and receiving data over a network is implemented as an asynchronous operation in Boost.Asio. Think of an asynchronous operation as a function that immediately returns, but without any result. The result is handed over later.

In the first step, an asynchronous operation is started. In the second step, a program is notified when the asynchronous operation has ended. This separation between starting and ending makes it possible to access external resources without having to call blocking functions.

Chapter 33. Boost.Interprocess

Table of Contents

Shared Memory
Managed Shared Memory
Synchronization

Interprocess communication describes mechanisms to exchange data between programs running on the same computer. It does not include network communication. To exchange data between programs running on different computers connected through a network, see Chapter 32, which covers Boost.Asio.

This chapter presents the library Boost.Interprocess, which contains numerous classes that abstract operating system specific interfaces for interprocess communication. Even though the concepts of interprocess communication are similar between different operating systems, the interfaces can vary greatly. Boost.Interprocess provides platform-independent access.

While Boost.Asio can be used to exchange data between processes running on the same computer, Boost.Interprocess usually provides better performance. Boost.Interprocess calls operating system functions optimized for data exchange between processes running on the same computer and thus should be the first choice to exchange data without a network.

Part VII. Streams and Files

The following libraries facilitate working with streams and files.

- Boost.IOStreams provides streams that go far beyond what the standard library offers. Boost.IOStreams gives you access to more streams to read and write data from and to many different sources and sinks. In addition, filters enable a variety of operations such as compressing/uncompressing data while reading or writing.
- Boost.Filesystem provides access to the filesystem. With Boost.Filesystem you can, for example, copy files or iterate over files in a directory.

Table of Contents

34. Boost.IOStreams
35. Boost.Filesystem

Chapter 34. Boost.IOStreams

Table of Contents

Devices
Filters

This chapter introduces the library Boost.IOStreams. Boost.IOStreams breaks up the well-known streams from the standard library into smaller components. The library defines two concepts: *device*, which describes data sources and sinks, and *stream*, which describes an interface for formatted input/output based on the interface from the standard library. A stream defined by Boost.IOStreams isn't automatically connected to a data source or sink.

Boost.IOStreams provides numerous implementations of the two concepts. For example, there is the device `boost::iostreams::mapped_file`, which loads a file partially or completely into memory. The stream `boost::iostreams::stream` can be connected to a device like `boost::iostreams::mapped_file` to use the familiar stream operators `operator<<` and `operator>>` to read and write data.

In addition to `boost::iostreams::stream`, Boost.IOStreams provides the stream `boost::iostreams::filtering_stream`, which lets you add data filters. For example, you can use `boost::iostreams::gzip_compressor` to write data compressed in the GZIP format.

Boost.IOStreams can also be used to connect to platform-specific objects. The library provides devices to connect to a Windows handle or a file descriptor. That way objects from low-level APIs can be made available in platform-independent C++ code.

The classes and functions provided by Boost.IOStreams are defined in the namespace `boost::iostreams`. There is no master header file. Because Boost.IOStreams contains more than header files, it must be prebuilt. This can be important because, depending on how Boost.IOStreams has been prebuilt, support for some features could be missing.

Chapter 35. Boost.Filesystem

Table of Contents

Paths
Files and Directories
Directory Iterators
File Streams

The library Boost.Filesystem makes it easy to work with files and directories. It provides a class called `boost::filesystem::path` that processes paths. In addition, many free-standing functions are available to handle tasks like creating directories or checking whether a file exists.

Boost.Filesystem has been revised several times. This chapter introduces Boost.Filesystem 3, the current version. This version has been the default since the Boost C++ Libraries 1.46.0. Boost.Filesystem 2 was last shipped with version 1.49.0.

Part VIII. Time

The following libraries process time values.

- Boost.DateTime defines classes for time points and periods – for both time of day and calendar dates – and functions to process them. For example, it is possible to iterate over dates.
- Boost.Chrono and Boost.Timer provide clocks to measure time. The clocks provided by Boost.Timer are specialized for measuring code execution time and are only used when optimizing code.

Table of Contents

36. Boost.DateTime
37. Boost.Chrono
38. Boost.Timer

Chapter 36. Boost.DateTime

Table of Contents

Calendar Dates
Location-independent Times
Location-dependent Times
Formatted Input and Output

The library Boost.DateTime can be used to process time data such as calendar dates and times. In addition, Boost.DateTime provides extensions to account for time zones and supports formatted input and output of calendar dates and times. If you are looking for functions to get the current time or measure time, see Boost.Chrono in Chapter 37.

Chapter 37. Boost.Chrono

The library Boost.Chrono provides a variety of clocks. For example, you can get the current time or you can measure the time passed in a process.

Parts of Boost.Chrono were added to C++11. If your development environment supports C++11, you have access to several clocks defined in the header file `chrono`. However, C++11 doesn't support some features, for example clocks to measure CPU time. Furthermore, only Boost.Chrono supports user-defined output formats for time.

You have access to all Boost.Chrono clocks through the header file `boost/chrono.hpp`. The only extension is user-defined formatting, which requires the header file `boost/chrono_io.hpp`.

Example 37.1. All clocks from Boost.Chrono

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

using namespace boost::chrono;

int main()
{
  std::cout << system_clock::now() << '\n';
#ifdef BOOST_CHRONO_HAS_CLOCK_STEADY
  std::cout << steady_clock::now() << '\n';
#endif
  std::cout << high_resolution_clock::now() << '\n';

#ifdef BOOST_CHRONO_HAS_PROCESS_CLOCKS
  std::cout << process_real_cpu_clock::now() << '\n';
  std::cout << process_user_cpu_clock::now() << '\n';
  std::cout << process_system_cpu_clock::now() << '\n';
  std::cout << process_cpu_clock::now() << '\n';
#endif

#ifdef BOOST_CHRONO_HAS_THREAD_CLOCK
  std::cout << thread_clock::now() << '\n';
#endif
}
```

Example 37.1 introduces all of the clocks provided by Boost.Chrono. All clocks have in common the member function `now()`, which returns a timepoint. All timepoints are relative to a universally valid timepoint. This reference timepoint is called *epoch*. An often used epoch is 1 January 1970. Example 37.1 writes the epoch for every timepoint displayed.

Boost.Chrono includes the following clocks:

- `boost::chrono::system_clock` returns the system time. This is the time usually displayed on the desktop of your computer. If you change the time on your computer, `boost::chrono::system_clock` returns the new time. Example 37.1 writes a string to standard output that looks like the following: `13919594042183544 [1/10000000]seconds since Jan 1, 1970`.

  The epoch isn't standardized for `boost::chrono::system_clock`. The epoch 1 January 1970, which is used in these examples, is implementation dependent. However, if you

specifically want to get the time since 1 January 1970, call `to_time_t()`. `to_time_t()` is a static member function that returns the current system time as the number of seconds since 1 January 1970 as a `std::time_t`.

- `boost::chrono::steady_clock` is a clock that will always return a later time when it is accessed later. Even if the time is set back on a computer, `boost::chrono::steady_clock` will return a later time. This time is known as *monotonic time*.

  Example 37.1 displays the number of nanoseconds since the system was booted. The message looks like the following: `10594369282958 nanoseconds since boot`. `boost::chrono::steady_clock` measures the time elapsed since the last boot. However, starting the measurement since the last boot is an implementation detail. The reference point could change with a different implementation.

  `boost::chrono::steady_clock` isn't supported on all platforms. The clock is only available if the macro `BOOST_CHRONO_HAS_CLOCK_STEADY` is defined.

- `boost::chrono::high_resolution_clock` is a type definition for `boost::chrono::system_clock` or `boost::chrono::steady_clock`, depending on which clock measures time more precisely. Thus, the output is identical to the output of the clock `boost::chrono::high_resolution_clock` is based on.

- `boost::chrono::process_real_cpu_clock` returns the CPU time a process has been running. The clock measures the time since program start. Example 37.1 writes a string to standard output that looks like the following: `1000000 nanoseconds since process start-up`.

  You could also get this time using `std::clock()` from `ctime`. In fact, the current implementation of `boost::chrono::process_real_cpu_clock` is based on `std::clock()`.

  The `boost::chrono::process_real_cpu_clock` clock and other clocks measuring CPU time can only be used if the macro `BOOST_CHRONO_HAS_PROCESS_CLOCKS` is defined.

- `boost::chrono::process_user_cpu_clock` returns the CPU time a process spent in *user space*. User space refers to code that runs separately from operating system functions. The time it takes to execute code in operating system functions called by a program is not counted as user space time.

  `boost::chrono::process_user_cpu_clock` returns only the time spent running in user space. If a program is halted for a while, for example through the Windows `Sleep()` function, the time spent in `Sleep()` isn't measured by `boost::chrono::process_user_cpu_clock`.

  Example 37.1 writes a string to standard output that looks like the following: `15600100 nanoseconds since process start-up`.

- `boost::chrono::process_system_cpu_clock` is similar to `boost::chrono::process_user_cpu_clock`. However, this clock measures the time spent in *kernel space*. `boost::chrono::process_system_cpu_clock` returns the CPU time a process spends executing operating system functions.

  Example 37.1 writes a string to the standard output that looks like the following: `0 nanoseconds since process start-up`. Because this example doesn't call operating system functions directly and because Boost.Chrono uses only a few operating system functions, `boost::chrono::process_system_cpu_clock` may return 0.

- `boost::chrono::process_cpu_clock` returns a tuple with the CPU times which are returned by `boost::chrono::process_real_cpu_clock`, `boost::chrono::process_user_cpu_clock` and `boost::chrono::process_system_cpu_clock`. Example 37.1 writes a string to standard output that looks like the following: `{1000000;15600100;0} nanoseconds since process start-up`.

- `boost::chrono::thread_clock` returns the time used by a thread. The time measured by `boost::chrono::thread_clock` is comparable to CPU time, except it is per thread, rather than per process. `boost::chrono::thread_clock` returns the CPU time the thread has been running. It does not distinguish between time spent in user and kernel space.

  `boost::chrono::thread_clock` isn't supported on all platforms. You can only use `boost::chrono::thread_clock` if the macro `BOOST_CHRONO_HAS_THREAD_CLOCK` is defined.

  Boost.Chrono provides the macro, `BOOST_CHRONO_THREAD_CLOCK_IS_STEADY`, to detect whether `boost::chrono::thread_clock` measures monotonic time like `boost::chrono::steady_clock`.

  Example 37.1 writes a string to standard output that looks like the following: `15600100 nanoseconds since thread start-up`.

All of the clocks in Boost.Chrono depend on operating system functions; thus, the operating system determines how precise and reliable the returned times are.

Example 37.2. Adding and subtracting durations using Boost.Chrono

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

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << p - nanoseconds{1} << '\n';
  std::cout << p + milliseconds{1} << '\n';
  std::cout << p + seconds{1} << '\n';
  std::cout << p + minutes{1} << '\n';
  std::cout << p + hours{1} << '\n';
}
```

`now()` returns an object of type `boost::chrono::time_point` for all clocks. This type is tightly coupled with a clock because the timepoint is measured relative to a reference timepoint that is defined by a clock. `boost::chrono::time_point` is a template that expects the type of a clock as a parameter. Each clock type provides a type definition for its specialized `boost::chrono::time_point`. For example, the type definition for `process_real_cpu_clock` is `process_real_cpu_clock::time_point`.

Boost.Chrono also provides the class `boost::chrono::duration`, which describes durations. Because `boost::chrono::duration` is also a template, Boost.Chrono provides the six classes `boost::chrono::nanoseconds`, `boost::chrono::milliseconds`, `boost::chrono::microseconds`, `boost::chrono::seconds`, `boost::chrono::minutes`, and `boost::chrono::hours`, which are easier to use.

Boost.Chrono overloads several operators to process timepoints and durations. Example 37.2 subtracts durations from or adds durations to **p** to get new timepoints, which are written to standard output.

Example 37.2 displays all timepoints in nanoseconds. Boost.Chrono automatically uses the smallest unit when timepoints and durations are processed to make sure that results are as precise as possible. If you want to use a timepoint with another unit, you have to cast it.

Example 37.3. Casting timepoints with `boost::chrono::time_point_cast()`

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

using namespace boost::chrono;

int main()
{
  process_real_cpu_clock::time_point p = process_real_cpu_clock::now();
  std::cout << p << '\n';
  std::cout << time_point_cast<minutes>(p) << '\n';
}
```

The `boost::chrono::time_point_cast()` function is used like a cast operator. Example 37.3 uses `boost::chrono::time_point_cast()` to convert a timepoint based on nanoseconds to a timepoint in minutes. You must use `boost::chrono::time_point_cast()` in this case because the timepoint cannot be expressed in a less precise unit (minutes) without potentially losing precision. You don't require `boost::chrono::time_point_cast()` to convert from less precise to more precise units.

Boost.Chrono also provides cast operators for durations.

Example 37.4. Casting durations with `boost::chrono::duration_cast()`

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

using namespace boost::chrono;

int main()
{
```

```
  minutes m{1};
  seconds s{35};

  std::cout << m + s << '\n';
  std::cout << duration_cast<minutes>(m + s) << '\n';
}
```

Example 37.4 uses the function `boost::chrono::duration_cast()` to cast a duration from seconds to minutes. This example writes `1 minute` to standard output.

Example 37.5. Rounding durations

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

using namespace boost::chrono;

int main()
{
  std::cout << floor<minutes>(minutes{1} + seconds{45}) << '\n';
  std::cout << round<minutes>(minutes{1} + seconds{15}) << '\n';
  std::cout << ceil<minutes>(minutes{1} + seconds{15}) << '\n';
}
```

Boost.Chrono also provides functions to round durations when casting. `boost::chrono::round()` rounds up or down, `boost::chrono::floor()` rounds down, and `boost::chrono::ceil()` rounds up. `boost::chrono::floor()` uses `boost::chrono::duration_cast()` – there is no difference between these two functions.

Example 37.5 writes `1 minute`, `1 minute`, and `2 minutes` to standard output.

Example 37.6. Stream manipulators for user-defined output

```
#define BOOST_CHRONO_VERSION 2
#include <boost/chrono.hpp>
#include <boost/chrono/chrono_io.hpp>
#include <iostream>

using namespace boost::chrono;

int main()
{
  std::cout << symbol_format << minutes{10} << '\n';

  std::cout << time_fmt(boost::chrono::timezone::local, "%H:%M:%S") <<
    system_clock::now() << '\n';
}
```

Boost.Chrono provides various stream manipulators to format the output of timepoints and durations. For example, with the manipulator `boost::chrono::symbol_format()`, the time unit is written as a symbol instead of a name. Thus, Example 37.6 displays `10 min`.

The manipulator `boost::chrono::time_fmt()` can be used to set a timezone and a format string. The timezone must be set to `boost::chrono::timezone::local` or `boost::chrono::timezone::utc`. The format string can use flags to refer to various components of a timepoint. For example, Example 37.6 writes a string to the standard output that looks like the following: `15:46:44`.

Beside stream manipulators, Boost.Chrono provides facets for many different customizations. For example, there is a facet that makes it possible to output timepoints in another language.

Note

There are two versions of the input/output functions since Boost 1.52.0. Since Boost 1.55.0, the newer version is used by default. If you use a version older than 1.55.0, you must define the macro `BOOST_CHRONO_VERSION` and set it to 2 for Example 37.6 to work.

Chapter 38. Boost.Timer

Boost.Timer provides clocks to measure code performance. At first, it may seem like this library competes with Boost.Chrono. However, while Boost.Chrono provides clocks to measure arbitrary periods, Boost.Timer measures the time it takes to execute code. Although Boost.Timer uses Boost.Chrono, when you want to measure code performance, you should use Boost.Timer rather than Boost.Chrono.

Since version 1.48.0 of the Boost libraries, there have been two versions of Boost.Timer. The second version of Boost.Timer has only one header file: `boost/timer/timer.hpp`. The library also ships a header file `boost/timer.hpp`. Do not use this header file. It belongs to the first version of Boost.Timer, which shouldn't be used anymore.

The clocks provided by Boost.Timer are implemented in the classes `boost::timer::cpu_timer` and `boost::timer::auto_cpu_timer`. `boost::timer::auto_cpu_timer` is derived from `boost::timer::cpu_timer` and automatically stops the time in the destructor. It then writes the time to an output stream.

Example 38.1 starts by introducing the class `boost::timer::cpu_timer`. This example and the following examples do some calculations to make sure enough time elapses to be measurable. Otherwise the timers would always measure 0, and it would be difficult to introduce the clocks from this library.

Example 38.1. Measuring time with `boost::timer::cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

Measurement starts when `boost::timer::cpu_timer` is instantiated. You can call the member function `format()` at any point to get the elapsed time. Example 38.1 displays output in the following format: `0.099170s wall, 0.093601s user + 0.000000s system = 0.093601s CPU (94.4%)`.

Boost.Timer measures wall and CPU time. The wall time is the time which passes according to a wall clock. You could measure this time yourself with a stop watch. The CPU time says how much time the program spent executing code. On today's multitasking systems a processor isn't available for a program all the time. A program may also need to halt and wait for user input. In these cases the wall time moves on but not the CPU time.

CPU time is divided between time spent in *user space* and time spent in *kernel space*. Kernel space refers to code that is part of the operating system. User space is code that doesn't belong to the operating system. User space includes your program code and code from third-party libraries. For example, the Boost libraries are included in user space. The amount of time spent in kernel space depends on the operating system functions called and how much time those functions need.

Example 38.2. Stopping and resuming timers

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.stop();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';

  timer.resume();

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
  std::cout << timer.format() << '\n';
}
```

`boost::timer::cpu_timer` provides the member functions `stop()` and `resume()`, which stop and resume timers. In Example 38.2, the timer is stopped before the second `for` loop runs and resumed afterwards. Thus, the second `for` loop isn't measured. This is similar to a stop watch that is stopped and then resumed after a while. The time returned by the second call to `format()` in Example 38.2 is the same as if the second `for` loop didn't exist.

`boost::timer::cpu_timer` also provides a member function `start()`. If you call `start()`, instead of `resume()`, the timer restarts from zero. The constructor of `boost::timer::cpu_timer` calls `start()`, which is why the timer starts immediately when `boost::timer::cpu_timer` is instantiated.

Example 38.3. Getting wall and CPU time as a tuple

```
#include <boost/timer/timer.hpp>
#include <iostream>
#include <cmath>

using namespace boost::timer;

int main()
{
  cpu_timer timer;
```

```
  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);

  cpu_times times = timer.elapsed();
  std::cout << times.wall << '\n';
  std::cout << times.user << '\n';
  std::cout << times.system << '\n';
}
```

While `format()` returns the measured wall and CPU time as a string, it is also possible to receive the times in a tuple (see Example 38.3). `boost::timer::cpu_timer` provides the member function `elapsed()` for that. `elapsed()` returns a tuple of type `boost::timer::times`. This tuple has three member variables: **`wall`**, **`user`**, and **`system`**. These member variables contain the wall and CPU times in nanoseconds. Their type is `boost::int_least64_t`.

`boost::timer::times` provides the member function `clear()` to set **`wall`**, **`user`**, and **`system`** to 0.

Example 38.4. Measuring times automatically with `boost::timer::auto_cpu_timer`

```
#include <boost/timer/timer.hpp>
#include <cmath>

using namespace boost::timer;

int main()
{
  auto_cpu_timer timer;

  for (int i = 0; i < 1000000; ++i)
    std::pow(1.234, i);
}
```

You can measure the wall and CPU time of a code block with `boost::timer::auto_cpu_timer`. Because the destructor of this class stops measuring time and writes the time to the standard output stream, Example 38.4 does the same thing as Example 38.1.

`boost::timer::auto_cpu_timer` provides several constructors. For example, you can pass an output stream that will be used to display the time. By default, the output stream is **`std::cout`**.

You can specify the format of reported times for `boost::timer::auto_cpu_timer` and `boost::timer::cpu_timer`. Boost.Timer provides format flags similar to the format flags supported by Boost.Format or `std::printf()`. The documentation contains an overview of the format flags.

Part IX. Functional Programming

In the functional programming model, functions are objects that, like other objects, can be passed as parameters to functions or stored in containers. There are numerous Boost libraries that support the functional programming model.

- Boost.Phoenix is the most extensive and, as of today, most important of these libraries. It replaces the library Boost.Lambda, which is introduced briefly, but only for completeness.
- Boost.Function provides a class that makes it easy to define a function pointer without using the syntax that originated with the C programming language.
- Boost.Bind is an adapter that lets you pass functions as parameters to other functions even if the actual signature is different from the expected signature.
- Boost.Ref can be used to pass a reference to an object, even if a function passes the parameter by copy.
- Boost.Lambda could be called a predecessor of Boost.Phoenix. It is a rather old library and allowed using lambda functions many years before they were added with C++11 to the programming language.

Table of Contents

39. Boost.Phoenix
40. Boost.Function
41. Boost.Bind
42. Boost.Ref
43. Boost.Lambda

Chapter 39. Boost.Phoenix

Boost.Phoenix is the most important Boost library for functional programming. While libraries like Boost.Bind or Boost.Lambda provide some support for functional programming, Boost.Phoenix includes the features of these libraries and goes beyond them.

In functional programming, functions are objects and can be processed like objects. With Boost.Phoenix, it is possible for a function to return another function as a result. It is also possible to pass a function as a parameter to another function. Because functions are objects, it's possible to distinguish between instantiation and execution. Accessing a function isn't equal to executing it.

Boost.Phoenix supports functional programming with function objects: Functions are objects based on classes which overload the operator `operator()`. That way function objects behave like other objects in C++. For example, they can be copied and stored in a container. However, they also behave like functions because they can be called.

Functional programming isn't new in C++. You can pass a function as a parameter to another function without using Boost.Phoenix.

Example 39.1. Predicates as global function, lambda function, and Phoenix function

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

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  std::cout << std::count_if(v.begin(), v.end(), is_odd) << '\n';

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.1 uses the algorithm `std::count_if()` to count odd numbers in vector **v**. `std::count_if()` is called three times, once with a predicate as a free-standing function, once with a lambda function, and once with a Phoenix function.

The Phoenix function differs from free-standing and lambda functions because it has no frame. While the other two functions have a function header with a signature, the Phoenix function seems to consist of a function body only.

The crucial component of the Phoenix function is **`boost::phoenix::placeholders::arg1`**. **`arg1`** is a global instance of a function object. You can use it like **`std::cout`**: These objects exist

once the respective header file is included.

arg1 is used to define an unary function. The expression `arg1 % 2 == 1` creates a new function that expects one parameter. The function isn't executed immediately but stored in **phoenix**. **phoenix** is passed to `std::count_if()` which calls the predicate for every number in **v**.

arg1 is a placeholder for the value passed when the Phoenix function is called. Since only **arg1** is used here, a unary function is created. Boost.Phoenix provides additional placeholders such as **boost::phoenix::placeholders::arg2** and **boost::phoenix::placeholders::arg3**. A Phoenix function always expects as many parameters as the placeholder with the greatest number.

Example 39.1 writes `3` three times to standard output.

Example 39.2. Phoenix function versus lambda function

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  auto lambda = [](int i){ return i % 2 == 1; };
  std::cout << std::count_if(v.begin(), v.end(), lambda) << '\n';

  std::vector<long> v2;
  v2.insert(v2.begin(), v.begin(), v.end());

  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
  std::cout << std::count_if(v2.begin(), v2.end(), phoenix) << '\n';
}
```

Example 39.2 highlights a crucial difference between Phoenix and lambda functions. In addition to requiring no function header with a parameter list, Phoenix function parameters have no types. The lambda function **lambda** expects a parameter of type `int`. The Phoenix function **phoenix** will accept any type that the modulo operator can handle.

Think of Phoenix functions as function templates. Like function templates, Phoenix functions can accept any type. This makes it possible in Example 39.2 to use **phoenix** as a predicate for the containers **v** and **v2** even though they store numbers of different types. If you try to use the predicate **lambda** with **v2**, you get a compiler error.

Example 39.3. Phoenix functions as deferred C++ code

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};
```

```
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > 2 && arg1 % 2 == 1;
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.3 uses a Phoenix function as a predicate with `std::count_if()` to count odd numbers greater than 2. The Phoenix function accesses **`arg1`** twice: Once to test if the placeholder is greater than 2 and once to test whether it's an odd number. The conditions are linked with `&&`.

You can think of Phoenix functions as C++ code that isn't executed immediately. The Phoenix function in Example 39.3 looks like a condition that uses multiple logical and arithmetic operators. However, the condition isn't executed immediately. It is only executed when it is accessed from within `std::count_if()`. The access in `std::count_if()` is a normal function call.

Example 39.3 writes `2` to standard output.

Example 39.4. Explicit Phoenix types

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  auto phoenix = arg1 > val(2) && arg1 % val(2) == val(1);
  std::cout << std::count_if(v.begin(), v.end(), phoenix) << '\n';
}
```

Example 39.4 uses explicit types for all operands in the Phoenix function. Strictly speaking, you don't see types, just the helper function `boost::phoenix::val()`. This function returns a function object initialized with the values passed to `boost::phoenix::val()`. The actual type of the function object doesn't matter. What is important is that Boost.Phoenix overloads operators like `>`, `&&`, `%` and `==` for different types. Thus, conditions aren't checked immediately. Instead, function objects are combined to create more powerful function objects. Depending on the operands they may be automatically used as function objects. Otherwise you can call helper functions like `val()`.

Example 39.5. **`boost::phoenix::placeholders::arg1`** and `boost::phoenix::val()`

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

int main()
{
  using namespace boost::phoenix::placeholders;
  std::cout << arg1(1, 2, 3, 4, 5) << '\n';

  auto v = boost::phoenix::val(2);
  std::cout << v() << '\n';
}
```

Example 39.5 illustrates how **arg1** and `val()` work. **arg1** is an instance of a function object. It can be used directly and called like a function. You can pass as many parameters as you like – **arg1** returns the first one.

`val()` is a function to create an instance of a function object. The function object is initialized with the value passed as a parameter. If the instance is accessed like a function, the value is returned.

Example 39.5 writes 1 and 2 to standard output.

Example 39.6. Creating your own Phoenix functions

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

struct is_odd_impl
{
    typedef bool result_type;

    template <typename T>
    bool operator()(T t) const { return t % 2 == 1; }
};

boost::phoenix::function<is_odd_impl> is_odd;

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

Example 39.6 explains how you can create your own Phoenix function. You pass a function object to the template `boost::phoenix::function`. The example passes the class `is_odd_impl`. This class overloads the operator `operator()`: when an odd number is passed in, the operator returns `true`. Otherwise, the operator returns `false`.

Please note that you must define the type `result_type`. Boost.Phoenix uses it to detect the type of the return value of the operator `operator()`.

`is_odd()` is a function you can use like `val()`. Both functions return a function object. When called, parameters are forwarded to the operator `operator()`. For Example 39.6, this means that `std::count_if()` still counts odd numbers.

Example 39.7. Transforming free-standing functions into Phoenix functions

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

bool is_odd_function(int i) { return i % 2 == 1; }

BOOST_PHOENIX_ADAPT_FUNCTION(bool, is_odd, is_odd_function, 1)
```

```
int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), is_odd(arg1)) << '\n';
}
```

If you want to transform a free-standing function into a Phoenix function, you can proceed as in Example 39.7. You don't necessarily have to define a function object as in the previous example.

You use the macro `BOOST_PHOENIX_ADAPT_FUNCTION` to turn a free-standing function into a Phoenix function. Pass the type of the return value, the name of the Phoenix function to define, the name of the free-standing function, and the number of parameters to the macro.

Example 39.8. Phoenix functions with `boost::phoenix::bind()`

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

bool is_odd(int i) { return i % 2 == 1; }

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  std::cout << std::count_if(v.begin(), v.end(), bind(is_odd, arg1)) << '\n';
}
```

To use a free-standing function as a Phoenix function, you can also use `boost::phoenix::bind()` as in Example 39.8. `boost::phoenix::bind()` works like `std::bind()`. The name of the free-standing function is passed as the first parameter. All further parameters are forwarded to the free-standing function.

Tip

Avoid `boost::phoenix::bind()`. Create your own Phoenix functions. This leads to more readable code. Especially with complex expressions it's not helpful having to deal with the additional details of `boost::phoenix::bind()`.

Example 39.9. Arbitrarily complex Phoenix functions

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

int main()
{
  std::vector<int> v{1, 2, 3, 4, 5};

  using namespace boost::phoenix;
  using namespace boost::phoenix::placeholders;
  int count = 0;
```

```
  std::for_each(v.begin(), v.end(), if_(arg1 > 2 && arg1 % 2 == 1)
    [
      ++ref(count)
    ]);
  std::cout << count << '\n';
}
```

Boost.Phoenix provides some function objects that simulate C++ keywords. For example, you can use the function `boost::phoenix::if_()` (see Example 39.9) to create a function object that acts like `if` and tests a condition. If the condition is true, the code passed to the function object with `operator[]` will be executed. Of course, that code also has to be based on function objects. That way, you can create complex Phoenix functions.

Example 39.9 increments **`count`** for every odd number greater than 2. To use the increment operator on **`count`**, **`count`** is wrapped in a function object using `boost::phoenix::ref()`. In contrast to `boost::phoenix::val()`, no value is copied into the function object. The function object returned by `boost::phoenix::ref()` stores a reference – here a reference to **`count`**.

Tip

Don’t use Boost.Phoenix to create complex functions. It is better to use lambda functions from C++11. While Boost.Phoenix comes close to C++ syntax, using keywords like `if_` or code blocks between square brackets doesn’t necessarily improve readability.

Chapter 40. Boost.Function

Boost.Function provides a class called `boost::function` to encapsulate function pointers. It is defined in `boost/function.hpp`.

If you work in a development environment supporting C++11, you have access to the class `std::function` from the header file `functional`. In this case you can ignore Boost.Function because `boost::function` and `std::function` are equivalent.

Example 40.1. Using `boost::function`

```
#include <boost/function.hpp>
#include <iostream>
#include <cstdlib>
#include <cstring>

int main()
{
  boost::function<int(const char*)> f = std::atoi;
  std::cout << f("42") << '\n';
  f = std::strlen;
  std::cout << f("42") << '\n';
}
```

`boost::function` makes it possible to define a pointer to a function with a specific signature. Example 40.1 defines a pointer **f** that can point to functions that expect a parameter of type `const char*` and return a value of type `int`. Once defined, functions with matching signatures can be assigned to the pointer. Example 40.1 first assigns the function `std::atoi()` to **f** before `std::strlen()` is assigned to **f**.

Please note that types do not need to match exactly. Even though `std::strlen()` uses `std::size_t` as its return type, it can still be assigned to **f**.

Because **f** is a function pointer, the assigned function can be called using `operator()`. Depending on what function is currently assigned, either `std::atoi()` or `std::strlen()` is called.

If **f** is called without having a function assigned, an exception of type `boost::bad_function_call` is thrown (see Example 40.2).

Example 40.2. `boost::bad_function_call` thrown if `boost::function` is empty

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

int main()
{
  try
  {
    boost::function<int(const char*)> f;
    f("");
  }
  catch (boost::bad_function_call &ex)
  {
    std::cerr << ex.what() << '\n';
```

```
  }
}
```

Note that assigning `nullptr` to a function pointer of type `boost::function` releases any currently assigned function. Calling it after it has been released will result in a `boost::bad_function_call` exception being thrown. To check whether or not a function pointer is currently assigned to a function, you can use the member functions `empty()` or `operator bool`.

It is also possible to assign class member functions to objects of type `boost::function` (see Example 40.3).

Example 40.3. Binding a class member function to `boost::function`

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

struct world
{
  void hello(std::ostream &os)
  {
    os << "Hello, world!\n";
  }
};

int main()
{
  boost::function<void(world*, std::ostream&)> f = &world::hello;
  world w;
  f(&w, std::ref(std::cout));
}
```

When calling such a function, the first parameter passed indicates the particular object for which the function is called. Therefore, the first parameter after the open parenthesis inside the template definition must be a pointer to that particular class. The remaining parameters denote the signature of the corresponding member function.

Chapter 41. Boost.Bind

Boost.Bind is a library that simplifies and generalizes capabilities that originally required `std::bind1st()` and `std::bind2nd()`. These two functions were added to the standard library with C++98 and made it possible to connect functions even if their signatures aren't compatible.

Boost.Bind was added to the standard library with C++11. If your development environment supports C++11, you will find the function `std::bind()` in the header file `functional`. Depending on the use case, it may be better to use lambda functions or Boost.Phoenix than `std::bind()` or Boost.Bind.

Example 41.1. `std::for_each()` with a compatible function

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

void print(int i)
{
  std::cout << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), print);
}
```

The third parameter of `std::for_each()` is a function or function object that expects a sole parameter. In Example 41.1, `std::for_each()` passes the numbers in the container **v** as sole parameters, one after another, to `print()`.

If you need to pass in a function whose signature doesn't meet the requirements of an algorithm, it gets more difficult. For example, if you want `print()` to accept an output stream as an additional parameter, you can no longer use it as is with `std::for_each()`.

Example 41.2. `std::for_each()` with `std::bind1st()`

```
#include <vector>
#include <algorithm>
#include <functional>
#include <iostream>

class print : public std::binary_function<std::ostream*, int, void>
{
public:
  void operator()(std::ostream *os, int i) const
  {
    *os << i << '\n';
  }
};

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), std::bind1st(print{}, &std::cout));
}
```

Like Example 41.1, Example 41.2 writes all numbers in **v** to standard output. However, this time, the output stream is passed to `print()` as a parameter. To do this, the function `print()` is defined as a function object derived from `std::binary_function`.

With Boost.Bind, you don't need to transform `print()` from a function to a function object. Instead, you use the function template `boost::bind()`, which is defined in `boost/bind.hpp`.

Example 41.3. `std::for_each()` with `boost::bind()`

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

void print(std::ostream *os, int i)
{
  *os << i << '\n';
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(), boost::bind(print, &std::cout, _1));
}
```

Example 41.3 uses `print()` as a function, not as a function object. Because `print()` expects two parameters, the function can't be passed directly to `std::for_each()`. Instead, `boost::bind()` is passed to `std::for_each()` and `print()` is passed as the first parameter to `boost::bind()`.

Since `print()` expects two parameters, those two parameters must also be passed to `boost::bind()`. They are a pointer to **`std::cout`** and **`_1`**.

`_1` is a placeholder. Boost.Bind defines placeholders from **`_1`** to **`_9`**. These placeholders tell `boost::bind()` to return a function object that expects as many parameters as the placeholder with the greatest number. If, as in Example 41.3, only the placeholder **`_1`** is used, `boost::bind()` returns an unary function object – a function object that expects a sole parameter. This is required in this case since `std::for_each()` passes only one parameter.

`std::for_each()` calls a unary function object. The value passed to the function object – a number from the container **v** – takes the position of the placeholder **`_1`**. `boost::bind()` takes the number and the pointer to **`std::cout`** and forwards them to `print()`.

Please note that `boost::bind()`, like `std::bind1st()` and `std::bind2nd()`, takes parameters by value. To prevent the calling program from trying to copy **`std::cout`**, `print()` expects a pointer to a stream. Boost.Ref provides a function which allows you to pass a parameter by reference.

Example 41.4 illustrates how to define a binary function object with `boost::bind()`. It uses the algorithm `std::sort()`, which expects a binary function as its third parameter.

Example 41.4. `std::sort()` with `boost::bind()`

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

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _1, _2));
  for (int i : v)
    std::cout << i << '\n';
}
```

In Example 41.4, a binary function object is created because the placeholder **_2** is used. The algorithm `std::sort()` calls this binary function object with two values from the container **v** and evaluates the return value to sort the container. The function `compare()` is defined to sort **v** in descending order.

Since `compare()` is a binary function, it can be passed to `std::sort()` directly. However, it can still make sense to use `boost::bind()` because it lets you change the order of the parameters. For example, you can use `boost::bind()` if you want to sort the container in ascending order but don't want to change `compare()`.

Example 41.5. `std::sort()` with `boost::bind()` and changed order of placeholders

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

bool compare(int i, int j)
{
  return i > j;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::sort(v.begin(), v.end(), boost::bind(compare, _2, _1));
  for (int i : v)
    std::cout << i << '\n';
}
```

Example 41.5 sorts **v** in ascending order simply by swapping the placeholders: **_2** is passed first and **_1** second.

Chapter 42. Boost.Ref

The library Boost.Ref provides two functions, `boost::ref()` and `boost::cref()`, in the header file `boost/ref.hpp`. They are useful if you use, for example, `std::bind()` for a function which expects parameters by reference. Because `std::bind()` takes parameters by value, you have to deal with references explicitly.

Boost.Ref was added to the standard library in C++11, where you will find the functions `std::ref()` and `std::cref()` in the header file `functional`.

Example 42.1. Using `boost::ref()`

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

void print(std::ostream &os, int i)
{
  os << i << std::endl;
}

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::bind(print, boost::ref(std::cout), std::placeholders::_1));
}
```

In Example 42.1, the function `print()` is passed to `std::for_each()` to write the numbers in **v** to an output stream. Because `print()` expects two parameters – an output stream and the number to be written – `std::bind()` is used. The first parameter passed to `print()` through `std::bind()` is **`std::cout`**. However, `print()` expects a reference to an output stream, while `std::bind()` passes parameters by value. Therefore, `boost::ref()` is used to wrap **`std::cout`**. `boost::ref()` returns a proxy object that contains a reference to the object passed to it. This makes it possible to pass a reference to **`std::cout`** even though `std::bind()` takes all parameters by value.

The function template `boost::cref()` lets you pass a `const` reference.

Chapter 43. Boost.Lambda

Before C++11, you needed to use a library like Boost.Lambda to take advantage of lambda functions. Since C++11, this library can be regarded as deprecated because lambda functions are now part of the programming language. If you work in a development environment that doesn't support C++11, you should consider Boost.Phoenix before you turn to Boost.Lambda. Boost.Phoenix is a newer library and probably the better choice if you need to use lambda functions without C++11.

The purpose of lambda functions is to make code more compact and easier to understand (see Example 43.1).

Example 43.1. `std::for_each()` with a lambda function

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

int main()
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    std::cout << boost::lambda::_1 << "\n");
}
```

Boost.Lambda provides several helpers to create nameless functions. Code is written where it should be executed, without needing to be wrapped in a function and without having to call a function explicitly. In Example 43.1, `std::cout << boost::lambda::_1 << "\n"` is a lambda function that expects one parameter, which it writes, followed by a new line, to standard output.

`boost::lambda::_1` is a placeholder that creates a lambda function that expects one parameter. The number in the placeholder determines the number of expected parameters, so **`boost::lambda::_2`** expects two parameters and **`boost::lambda::_3`** expects three parameters. Boost.Lambda only provides these three placeholders. The lambda function in Example 43.1 uses **`boost::lambda::_1`** because `std::for_each()` expects a unary function.

Include `boost/lambda/lambda.hpp` to use placeholders.

Please note that `\n`, instead of `std::endl`, is used in Example 43.1 to output a new line. If you use `std::endl`, the example won't compile because the type of the lambda function `std::cout << boost::lambda::_1` differs from what the unary function template `std::endl()` expects. Thus, you can't use `std::endl`.

Example 43.2. A lambda function with `boost::lambda::if_then()`

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

int main()
```

```
{
  std::vector<int> v{1, 3, 2};
  std::for_each(v.begin(), v.end(),
    boost::lambda::if_then(boost::lambda::_1 > 1,
    std::cout << boost::lambda::_1 << "\n"));
}
```

The header file `boost/lambda/if.hpp` defines constructs you can use to create `if` control structures in a lambda function. The simplest construct is the function template `boost::lambda::if_then()`, which expects two parameters: the first parameter is a condition. If the condition is true, the second parameter is executed. Both parameters can be lambda functions, as in Example 43.2.

In addition to `boost::lambda::if_then()`, Boost.Lambda provides the function templates `boost::lambda::if_then_else()` and `boost::lambda::if_then_else_return()`, both of which expect three parameters. Function templates are also provided `for` loops and cast operators and to throw exceptions in lambda functions. The many function templates defined by Boost.Lambda make it possible to define lambda functions that are in no way inferior to normal C++ functions.

Part X. Parallel Programming

The following libraries support the parallel programming model.

- Boost.Thread lets you create and manage your own threads.
- Boost.Atomic lets you access variables of integral types with atomic operations from multiple threads.
- Boost.Lockfree provides thread-safe containers.
- Boost.MPI originates from the supercomputer domain. With Boost.MPI your program is started multiple times and executed in multiple processes. You concentrate on programming the actual tasks that should be executed concurrently, and Boost.MPI coordinates the processes. With Boost.MPI you don’t need to take care of details like synchronizing access on shared data. However, Boost.MPI does require an appropriate runtime environment.

Table of Contents

44. Boost.Thread
45. Boost.Atomic
46. Boost.Lockfree
47. Boost.MPI

Chapter 44. Boost.Thread

Table of Contents

Creating and Managing Threads
Synchronizing Threads
Thread Local Storage
Futures and Promises

Boost.Thread is the library that allows you to use threads. Furthermore, it provides classes to synchronize access on data which is shared by multiple threads.

Threads have been supported by the standard library since C++11. You will also find classes in the standard library that threads can be created and synchronized with. While Boost.Thread resembles the standard library in many regards, it offers extensions. For example, you can interrupt threads created with Boost.Thread. You will also find special locks in Boost.Thread that will probably be added to the standard library with C++14. Thus, it can make sense to use Boost.Thread even if you work in a C++11 development environment.

Chapter 45. Boost.Atomic

Boost.Atomic provides the class `boost::atomic`, which can be used to create atomic variables. They are called atomic variables because all access is atomic. Boost.Atomic is used in multithreaded programs when access to a variable in one thread shouldn't be interrupted by another thread accessing the same variable. Without `boost::atomic`, attempts to access shared variables from multiple threads would need to be synchronized with locks.

`boost::atomic` depends on the target platform supporting atomic variable access. Otherwise, `boost::atomic` uses locks. The library allows you to detect whether a target platform supports atomic variable access.

If your development environment supports C++11, you don't need Boost.Atomic. The C++11 standard library provides a header file `atomic` that defines the same functionality as Boost.Atomic. For example, you will find a class named `std::atomic`.

Boost.Atomic supports more or less the same functionality as the standard library. While a few functions are overloaded in Boost.Atomic, they may have different names in the standard library. The standard library also provides some functions, such as `std::atomic_init()` and `std::kill_dependency()`, which are missing in Boost.Atomic.

Example 45.1. Using `boost::atomic`

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  ++a;
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.1 uses two threads to increment the `int` variable **a**. Instead of a lock, the example uses `boost::atomic` for atomic access to **a**. The example writes `2` to standard output.

`boost::atomic` works because some processors support atomic access on variables. If incrementing an `int` variable is an atomic operation, a lock isn't required. If this example is run on a platform that cannot increment a variable as an atomic operation, `boost::atomic` uses a lock.

Example 45.2. `boost::atomic` with or without lock

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

int main()
{
  std::cout.setf(std::ios::boolalpha);

  boost::atomic<short> s;
  std::cout << s.is_lock_free() << '\n';

  boost::atomic<int> i;
  std::cout << i.is_lock_free() << '\n';

  boost::atomic<long> l;
  std::cout << l.is_lock_free() << '\n';
}
```

You can call `is_lock_free()` on an atomic variable to check whether accessing the variable is done without a lock. If you run the example on an Intel x86 processor, it displays `true` three times. If you run it on a processor without lock-free access on `short`, `int` and `long` variables, `false` is displayed.

Boost.Atomic provides the `BOOST_ATOMIC_INT_LOCK_FREE` and `BOOST_ATOMIC_LONG_LOCK_FREE` macros to check, at compile time, which data types support lock-free access.

Example 45.2 uses integral data types only. You should not use `boost::atomic` with classes like `std::string` or `std::vector`. Boost.Atomic supports integers, pointers, booleans (`bool`), and trivial classes. Examples of integral types include `short`, `int` and `long`. Trivial classes define objects that can be copied with `std::memcpy()`.

Example 45.3. `boost::atomic` with **`boost::memory_order_seq_cst`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_seq_cst);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.3 increments **a** twice – this time not with `operator++` but with a call to `fetch_add()`. The member function `fetch_add()` can take two parameters: the number by which **a** should be incremented and the *memory order*.

The memory order specifies the order in which access operations on memory must occur. By default, this order is undetermined and does not depend on the order of the lines of code. Compilers and processors are allowed to change the order as long as a program behaves as if memory access operations were executed in source code order. This rule only applies to a thread. If more than one thread is used, variations in the order of memory accesses can lead to a program acting erroneously. Boost.Atomic supports specifying a memory order when accessing variables to make sure memory accesses occur in the desired order in a multithreaded program.

Note

Specifying memory order optimizes performance, but it increases complexity and makes it more difficult to write correct code. Therefore, in practice, you should have a really good reason for using memory orders.

Example 45.3 uses the memory order **`boost::memory_order_seq_cst`** to increment **a** by 1. The memory order stands for *sequential consistency*. This is the most restrictive memory order. All memory accesses that appear before the `fetch_add()` call must occur before this member function is executed. All memory accesses that appear after the `fetch_add()` call must occur after this member function is executed. Compilers and processors may reorder memory accesses before and after the call to `fetch_add()`, but they must not move a memory access from before to after the call to `fetch_add()` or vice versa. **`boost::memory_order_seq_cst`** is a strict boundary for memory accesses in both directions.

`boost::memory_order_seq_cst` is the most restrictive memory order. It is used by default when memory order is not set. Therefore, in Example 45.1, when **a** is incremented with `operator++`, **`boost::memory_order_seq_cst`** will be used.

`boost::memory_order_seq_cst` isn't always necessary. For example, in Example 45.3 there is no need to synchronize memory accesses for other variables because **a** is the only variable the threads use. **a** is written to standard output in `main()`, but only after both threads have terminated. The call to `join()` guarantees that **a** is only read after both threads have finished.

Example 45.4. `boost::atomic` with **`boost::memory_order_relaxed`**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};

void thread()
{
  a.fetch_add(1, boost::memory_order_relaxed);
}

int main()
{
  std::thread t1{thread};
  std::thread t2{thread};
  t1.join();
  t2.join();
  std::cout << a << '\n';
}
```

Example 45.4 sets the memory order to **boost::memory_order_relaxed**. This is the least restrictive memory order: it allows arbitrary reordering of memory accesses. This example works with this memory order because the threads access no variables except **a**. Therefore, no specific order is required.

Example 45.5. boost::atomic with **memory_order_release** and **memory_order_acquire**

```
#include <boost/atomic.hpp>
#include <thread>
#include <iostream>

boost::atomic<int> a{0};
int b = 0;

void thread1()
{
  b = 1;
  a.store(1, boost::memory_order_release);
}

void thread2()
{
  while (a.load(boost::memory_order_acquire) != 1)
    ;
  std::cout << b << '\n';
}

int main()
{
  std::thread t1{thread1};
  std::thread t2{thread2};
  t1.join();
  t2.join();
}
```

There are choices between the most restrictive memory order, **boost::memory_order_seq_cst**, and the least restrictive one, **boost::memory_order_relaxed**. Example 45.5 introduces the memory orders **boost::memory_order_release** and **boost::memory_order_acquire**.

Memory accesses that appear in the code before the **boost::memory_order_release** statement are executed before the **boost::memory_order_release** statement is executed. Compilers and processors must not move memory accesses from before to after **boost::memory_order_release**. However, they may move memory accesses from after to before **boost::memory_order_release**.

boost::memory_order_acquire works like **boost::memory_order_release**, but refers to memory accesses after **boost::memory_order_acquire**. Compilers and processors must not move memory accesses from after the **boost::memory_order_acquire** statement to before it. However, they may move memory accesses from before to after **boost::memory_order_acquire**.

Example 45.5 uses **boost::memory_order_release** in the first thread to make sure that **b** is set to 1 before **a** is set to 1. **boost::memory_order_release** guarantees that the memory access on **b** occurs before the memory access on **a**.

To specify a memory order when accessing **a**, `store()` is called. This member function corresponds to an assignment with `operator=`.

The second thread reads **a** in a loop. This is done with the member function `load()`. Again, no assignment operator is used.

In the second thread, **`boost::memory_order_acquire`** makes sure that the memory access on **b** doesn't occur before the memory access on **a**. The second thread waits in the loop for **a** to be set to 1 by the first thread. Once this happens, **b** is read.

The example writes `1` to standard output. The memory orders ensure that all memory accesses occur in the right order. The first thread always writes 1 to **b** first before the second thread accesses and reads **b**.

Tip

For more details and examples on how memory orders work, you can find an explanation in the GCC Wiki article on memory model synchronization modes.

Chapter 46. Boost.Lockfree

Boost.Lockfree provides thread-safe and lock-free containers. Containers from this library can be accessed from multiple threads without having to synchronize access.

In version 1.56.0, Boost.Lockfree provides only two containers: a queue of type `boost::lockfree::queue` and a stack of type `boost::lockfree::stack`. For the queue, a second implementation is available: `boost::lockfree::spsc_queue`. This class is optimized for use cases where exactly one thread writes to the queue and exactly one thread reads from the queue. The abbreviation spsc in the class name stands for single producer/single consumer.

Example 46.1. Using `boost::lockfree::spsc_queue`

```
#include <boost/lockfree/spsc_queue.hpp>
#include <thread>
#include <iostream>

boost::lockfree::spsc_queue<int> q{100};
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  consume();
  std::cout << sum << '\n';
}
```

Example 46.1 uses the container `boost::lockfree::spsc_queue`. The first thread, which executes the function `produce()`, adds the numbers 1 to 100 to the container. The second thread, which executes `consume()`, reads the numbers from the container and adds them up in **`sum`**. Because the container `boost::lockfree::spsc_queue` explicitly supports concurrent access from two threads, it isn't necessary to synchronize the threads.

Please note that the function `consume()` is called a second time after the threads terminate. This is required to calculate the total of all 100 numbers, which is 5050. Because `consume()` accesses the queue in a loop, it is possible that it will read the numbers faster than they are inserted by `produce()`. If the queue is empty, `pop()` returns `false`. Thus, the thread executing `consume()` could terminate because `produce()` in the other thread couldn't fill the queue fast enough. If the thread executing `produce()` is terminated, then it's clear that all of the numbers

were added to the queue. The second call to `consume()` makes sure that numbers that may not have been read yet are added to **`sum`**.

The size of the queue is passed to the constructor. Because `boost::lockfree::spsc_queue` is implemented with a circular buffer, the queue in Example 46.1 has a capacity of 100 elements. If a value can't be added because the queue is full, `push()` returns `false`. The example doesn't check the return value of `push()` because exactly 100 numbers are added to the queue. Thus, 100 elements is sufficient.

Example 46.2. `boost::lockfree::spsc_queue` with `boost::lockfree::capacity`

```
#include <boost/lockfree/spsc_queue.hpp>
#include <boost/lockfree/policies.hpp>
#include <thread>
#include <iostream>

using namespace boost::lockfree;

spsc_queue<int, capacity<100>> q;
int sum = 0;

void produce()
{
  for (int i = 1; i <= 100; ++i)
    q.push(i);
}

void consume()
{
  while (q.consume_one([](int i){ sum += i; }))
    ;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  t1.join();
  t2.join();
  q.consume_all([](int i){ sum += i; });
  std::cout << sum << '\n';
}
```

Example 46.2 works like the previous example, but this time the size of the circular buffer is set at compile time. This is done with the template `boost::lockfree::capacity`, which expects the capacity as a template parameter. **`q`** is instantiated with the default constructor – the capacity cannot be set at run time.

The function `consume()` has been changed to use `consume_one()`, rather than `pop()`, to read the number. A lambda function is passed as a parameter to `consume_one()`. `consume_one()` reads a number just like `pop()`, but the number isn't returned through a reference to the caller. It is passed as the sole parameter to the lambda function.

When the threads terminate, `main()` calls the member function `consume_all()`, instead of `consume()`. `consume_all()` works like `consume_one()` but makes sure that the queue is empty after the call. `consume_all()` calls the lambda function as long as there are elements in the queue.

Once again, Example 46.2 writes `5050` to standard output.

Example 46.3. `boost::lockfree::queue` with variable container size

```
#include <boost/lockfree/queue.hpp>
#include <thread>
#include <atomic>
#include <iostream>

boost::lockfree::queue<int> q{100};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

Example 46.3 executes `consume()` in two threads. Because more than one thread reads from the queue, the class `boost::lockfree::spsc_queue` must not be used. This example uses `boost::lockfree::queue` instead.

Thanks to `std::atomic`, access to the variable **`sum`** is also now thread safe.

The size of the queue is set to 100 – this is the parameter passed to the constructor. However, this is only the initial size. By default, `boost::lockfree::queue` is not implemented with a circular buffer. If more items are added to the queue than the capacity is set to, it is automatically increased. `boost::lockfree::queue` dynamically allocates additional memory if the initial size isn't sufficient.

That means that `boost::lockfree::queue` isn't necessarily lock free. The allocator used by `boost::lockfree::queue` by default is `boost::lockfree::allocator`, which is based on `std::allocator`. Thus, this allocator determines whether `boost::lockfree::queue` is lock free without constraints.

Example 46.4. `boost::lockfree::queue` with constant container size

```
#include <boost/lockfree/queue.hpp>
#include <thread>
#include <atomic>
#include <iostream>
```

```
using namespace boost::lockfree;

queue<int, fixed_sized<true>> q{10000};
std::atomic<int> sum{0};

void produce()
{
  for (int i = 1; i <= 10000; ++i)
    q.push(i);
}

void consume()
{
  int i;
  while (q.pop(i))
    sum += i;
}

int main()
{
  std::thread t1{produce};
  std::thread t2{consume};
  std::thread t3{consume};
  t1.join();
  t2.join();
  t3.join();
  consume();
  std::cout << sum << '\n';
}
```

Example 46.4 uses a constant size of 10,000 elements. In this example, the queue doesn't allocate additional memory when it is full. 10,000 is a fixed upper limit.

The queue's capacity is constant because `boost::lockfree::fixed_sized` is passed as a template parameter. The capacity is passed as a parameter to the constructor and can be updated at any time using `reserve()`. If the capacity must be set at compile time, `boost::lockfree::capacity` can be passed as a template parameter to `boost::lockfree::queue`. `boost::lockfree::capacity` includes `boost::lockfree::fixed_sized`.

In Example 46.4, the queue has a capacity of 10,000 elements. Because `consume()` inserts 10,000 numbers into the queue, the upper limit isn't exceeded. If it were exceeded, `push()` would return `false`.

`boost::lockfree::queue` is similar to `boost::lockfree::spsc_queue` and also provides member functions like `consume_one()` and `consume_all()`.

The third class, `boost::lockfree::stack`, is similar to the other ones. As with `boost::lockfree::queue`, `boost::lockfree::fixed_size` and `boost::lockfree::capacity` can be passed as template parameters. The member functions are similar, too.

Exercise

Remove the class `boost::lockfree::spsc_queue` from Example 46.1 and implement the program with `std::queue`.

Chapter 47. Boost.MPI

Table of Contents

Development and Runtime Environment
Simple Data Exchange
Asynchronous data exchange
Collective Data Exchange
Communicators

Boost.MPI provides an interface to the MPI standard (Message Passing Interface). This standard simplifies the development of programs that execute tasks concurrently. You could develop such programs using threads or by making multiple processes communicate with each other through shared memory or network connections. The advantage of MPI is that you don’t need to take care of such details. You can fully concentrate on parallelizing your program.

A disadvantage is that you need an MPI runtime environment. MPI is only an option if you control the runtime environment. For example, if you want to distribute a program that can be started with a double click, you won’t be able to use MPI. While operating systems support threads, shared memory, and networks out of the box, they usually don’t provide an MPI runtime environment. Users need to perform additional steps to start an MPI program.

Part XI. Generic Programming

The following libraries support generic programming. The libraries can be used without detailed knowledge of template meta programming.

- Boost.TypeTraits provides functions to check properties of types.
- Boost.EnableIf can be used together with Boost.TypeTraits to, for example, overload functions based on their return types.
- Boost.Fusion makes it possible to create heterogeneous containers – containers whose elements can have different types.

Table of Contents

48. Boost.TypeTraits
49. Boost.EnableIf
50. Boost.Fusion

Chapter 48. Boost.TypeTraits

Types have different properties that generic programming takes advantage of. The Boost.TypeTraits library provides the tools needed to determine a type's properties and change them.

Since C++11, some functions provided by Boost.TypeTraits can be found in the standard library. You can access those functions through the header file `type_traits`. However, Boost.TypeTraits provides additional functions.

Example 48.1. Determining type categories

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

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_integral<int>::value << '\n';
  std::cout << is_floating_point<int>::value << '\n';
  std::cout << is_arithmetic<int>::value << '\n';
  std::cout << is_reference<int>::value << '\n';
}
```

Example 48.1 calls several functions to determine type categories. `boost::is_integral` checks whether a type is integral – whether it can store integers. `boost::is_floating_point` checks whether a type stores floating point numbers. `boost::is_arithmetic` checks whether a type supports arithmetic operators. And `boost::is_reference` can be used to determine whether a type is a reference.

`boost::is_integral` and `boost::is_floating_point` are mutually exclusive. A type either stores an integer or a floating point number. However, `boost::is_arithmetic` and `boost::is_reference` can apply to multiple categories. For example, both integer and floating point types support arithmetic operations.

All functions from Boost.TypeTraits provide a result in **`value`** that is either `true` or `false`. Example 48.1 outputs `true` for `is_integral<int>` and `is_arithmetic<int>` and outputs `false` for `is_floating_point<int>` and `is_reference<int>`. Because all of these functions are templates, nothing is processed at run time. The example behaves at run time as though the values `true` and `false` were directly used in the code.

In Example 48.1, the result of the various functions is a value of type `bool`, which can be written directly to standard output. If the result is to be processed by a function template, it should be forwarded as a type, not as a `bool` value.

Example 48.2. `boost::true_type` and `boost::false_type`

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

```
using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_same<is_integral<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_floating_point<int>::type, false_type>::value <<
    '\n';
  std::cout << is_same<is_arithmetic<int>::type, true_type>::value << '\n';
  std::cout << is_same<is_reference<int>::type, false_type>::value << '\n';
}
```

Besides **value**, functions from Boost.TypeTraits also provide the result in `type`. While **value** is a `bool` value, `type` is a type. Just like **value**, which can only be set to `true` or `false`, `type` can only be set to one of two types: `boost::true_type` or `boost::false_type`. `type` lets you pass the result of a function as a type to another function.

Example 48.2 uses another function from Boost.TypeTraits called `boost::is_same`. This function expects two types as parameters and checks whether they are the same. To pass the results of `boost::is_integral`, `boost::is_floating_point`, `boost::is_arithmetic`, and `boost::is_reference` to `boost::is_same`, `type` must be accessed. `type` is then compared with `boost::true_type` or `boost::false_type`. The results from `boost::is_same` are then read through **value** again. Because this is a `bool` value, it can be written to standard output.

Example 48.3. Checking type properties with Boost.TypeTraits

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

using namespace boost;

int main()
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << has_plus<int>::value << '\n';
  std::cout << has_pre_increment<int>::value << '\n';
  std::cout << has_trivial_copy<int>::value << '\n';
  std::cout << has_virtual_destructor<int>::value << '\n';
}
```

Example 48.3 introduces functions that check properties of types. `boost::has_plus` checks whether a type supports the operator `operator+` and whether two objects of the same type can be concatenated. `boost::has_pre_increment` checks whether a type supports the pre-increment operator `operator++`. `boost::has_trivial_copy` checks whether a type has a trivial copy constructor. And `boost::has_virtual_destructor` checks whether a type has a virtual destructor.

Example 48.3 displays `true` three times and `false` once.

Example 48.4. Changing type properties with Boost.TypeTraits

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

using namespace boost;

int main()
```

```
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << is_const<add_const<int>::type>::value << '\n';
  std::cout << is_same<remove_pointer<int*>::type, int>::value << '\n';
  std::cout << is_same<make_unsigned<int>::type, unsigned int>::value <<
    '\n';
  std::cout << is_same<add_rvalue_reference<int>::type, int&&>::value <<
    '\n';
}
```

Example 48.4 illustrates how type properties can be changed. `boost::add_const` adds `const` to a type. If the type is already constant, nothing changes. The code compiles without problems, and the type remains constant.

`boost::remove_pointer` removes the asterisk from a pointer type and returns the type the pointer refers to. `boost::make_unsigned` turns a type with a sign into a type without a sign. And `boost::add_rvalue_reference` transforms a type into a rvalue reference.

Example 48.4 writes `true` four times to standard output.

Chapter 49. Boost.EnableIf

Boost.EnableIf makes it possible to disable overloaded function templates or specialized class templates. Disabling means that the compiler ignores the respective templates. This helps to prevent ambiguous scenarios in which the compiler doesn't know which overloaded function template to use. It also makes it easier to define templates that can be used not just for a certain type but for a group of types.

Since C++11, Boost.EnableIf has been part of the standard library. You can call the functions introduced in this chapter without using a Boost library; just include the header file `type_traits`.

Example 49.1. Overloading functions with `boost::enable_if` on their return value

```
#include <boost/utility/enable_if.hpp>
#include <type_traits>
#include <string>
#include <iostream>

template <typename T>
typename boost::enable_if<std::is_same<T, int>, T>::type create()
{
  return 1;
}

template <typename T>
typename boost::enable_if<std::is_same<T, std::string>, T>::type create()
{
  return "Boost";
}

int main()
{
  std::cout << create<std::string>() << '\n';
}
```

Example 49.1 defines the function template `create()`, which returns an object of the type passed as a template parameter. The object is initialized in `create()`, which accepts no parameters. The signatures of the two `create()` functions don't differ. In that respect `create()` isn't an overloaded function. The compiler would report an error if Boost.EnableIf didn't enable one function and disable the other.

Boost.EnableIf provides the class `boost::enable_if`, which is a template that expects two parameters. The first parameter is a condition. The second parameter is the type of the `boost::enable_if` expression if the condition is true. The trick is that this type doesn't exist if the condition is false, in which case the `boost::enable_if` expression is invalid C++ code. However, when it comes to templates, the compiler doesn't complain about invalid code. Instead it ignores the template and searches for another one that might fit. This concept is known as SFINAE which stands for "Substitution Failure Is Not An Error."

In Example 49.1 both conditions in the `boost::enable_if` expressions use the class `std::is_same`. This class is defined in the C++11 standard library and allows you to compare two types. Because such a comparison is either true or false, it's sufficient to use `std::is_same` to define a condition.

If a condition is true, the respective `create()` function should return an object of the type that was passed to `create()` as a template parameter. That's why `T` is passed as a second parameter to `boost::enable_if`. The entire `boost::enable_if` expression is replaced by `T` if the condition is true. In Example 49.1 the compiler sees either a function that returns an `int` or a function that returns a `std::string`. If `create()` is called with any other type than `int` or `std::string`, the compiler will report an error.

Example 49.1 displays `Boost`.

Example 49.2. Specializing functions for a group of types with `boost::enable_if`

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

template <typename T>
void print(typename boost::enable_if<std::is_integral<T>, T>::type i)
{
  std::cout << "Integral: " << i << '\n';
}

template <typename T>
void print(typename boost::enable_if<std::is_floating_point<T>, T>::type f)
{
  std::cout << "Floating point: " << f << '\n';
}

int main()
{
  print<short>(1);
  print<long>(2);
  print<double>(3.14);
}
```

Example 49.2 uses `boost::enable_if` to specialize a function for a group of types. The function is called `print()` and expects one parameter. It can be overloaded, although overloading requires you to use a concrete type. To do the same for a group of types like `short`, `int` or `long`, you can define an appropriate condition using `boost::enable_if`. Example 49.2 uses `std::is_integral` to do so. The second `print()` function is overloaded with `std::is_floating_point` for all floating point numbers.

Exercise

Make `print_has_post_increment()` write to standard output whether a type supports the post-increment operator. For example, for `int` the program should output "int has a post increment operator":

```
#include <string>

template <class T>
void print_has_post_increment()
{
    // TODO: Implement this function.
}

int main()
{
```

```
    print_has_post_increment<int>();
    print_has_post_increment<long>();
    print_has_post_increment<std::string>();
}
```

Chapter 50. Boost.Fusion

The standard library provides numerous containers that have one thing in common: They are homogeneous. That is, containers from the standard library can only store elements of one type. A vector of the type `std::vector<int>` can only store `int` values, and a vector of type `std::vector<std::string>` can only store strings.

Boost.Fusion makes it possible to create heterogeneous containers. For example, you can create a vector whose first element is an `int` and whose second element is a string. In addition, Boost.Fusion provides algorithms to process heterogeneous containers. You can think of Boost.Fusion as the standard library for heterogeneous containers.

Strictly speaking, since C++11, the standard library has provided a heterogeneous container, `std::tuple`. You can use different types for the values stored in a tuple. `boost:fusion::tuple` in Boost.Fusion is a similar type. While the standard library doesn't have much more to offer, tuples are just the starting place for Boost.Fusion.

Example 50.1. Processing Fusion tuples

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

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  std::cout << get<0>(t) << '\n';
  std::cout << get<1>(t) << '\n';
  std::cout << std::boolalpha << get<2>(t) << '\n';
  std::cout << get<3>(t) << '\n';
}
```

Example 50.1 defines a tuple consisting of an `int`, a `std::string`, a `bool`, and a `double`. The tuple is based on `boost:fusion::tuple`. In Example 50.1, the tuple is then instantiated, initialized, and the various elements are retrieved with `boost::fusion::get()` and written to standard output. The function `boost::fusion::get()` is similar to `std::get()`, which accesses elements in `std::tuple`.

Fusion tuples don't differ from tuples from the standard library. Thus it's no surprise that Boost.Fusion provides a function `boost::fusion::make_tuple()`, which works like `std::make_tuple()`. However, Boost.Fusion provides additional functions that go beyond what is offered in the standard library.

Example 50.2. Iterating over a tuple with `boost::fusion::for_each()`

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

using namespace boost::fusion;
```

```
struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  for_each(t, print{});
}
```

Example 50.2 introduces the algorithm `boost::fusion::for_each()`, which iterates over a Fusion container. The function is used here to write the values in the tuple `t` to standard output.

`boost::fusion::for_each()` is designed to work like `std::for_each()`. While `std::for_each()` only iterates over homogeneous containers, `boost::fusion::for_each()` works with heterogeneous containers. You pass a container, not an iterator, to `boost::fusion::for_each()`. If you don't want to iterate over all elements in a container, you can use a *view*.

Example 50.3. Filtering a Fusion container with `boost::fusion::filter_view`

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/view.hpp>
#include <boost/fusion/algorithm.hpp>
#include <boost/type_traits.hpp>
#include <boost/mpl/arg.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

struct print
{
  template <typename T>
  void operator()(const T &t) const
  {
    std::cout << std::boolalpha << t << '\n';
  }
};

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  filter_view<tuple_type, boost::is_integral<boost::mpl::arg<1>>> v{t};
  for_each(v, print{});
}
```

Boost.Fusion provides views, which act like containers but don't store data. With views, data in a container can be accessed differently. Views are similar to adaptors from Boost.Range. However, while adaptors from Boost.Range can be applied to only one container, views from Boost.Fusion can span data from multiple containers.

Example 50.3 uses the class `boost::fusion::filter_view` to filter the tuple **`t`**. The filter directs `boost::fusion::for_each()` to only write elements based on an integral type.

`boost::fusion::filter_view` expects as a first template parameter the type of the container to filter. The second template parameter must be a predicate to filter elements. The predicate must filter the elements based on their type.

The library is called Boost.Fusion because it combines two worlds: C++ programs process values at run time and types at compile time. For developers, values at run time are usually more important. Most tools from the standard library process values at run time. To process types at compile time, template meta programming is used. While values are processed depending on other values at run time, types are processed depending on other types at compile time. Boost.Fusion lets you process values depending on types.

The second template parameter passed to `boost::fusion::filter_view` is a predicate, which will be applied to every type in the tuple. The predicate expects a type as a parameter and returns `true` if the type should be part of the view. If `false` is returned, the type is filtered out.

Example 50.3 uses the class `boost::is_integral` from Boost.TypeTraits. `boost::is_integral` is a template that checks whether a type is integral. Because the template parameter has to be passed to `boost::fusion::filter_view`, a placeholder from Boost.MPL, `boost::mpl::arg<1>`, is used to create a lambda function. `boost::mpl::arg<1>` is similar to **`boost::phoenix::place_holders::arg1`** from Boost.Phoenix. In Example 50.3, the view **`v`** will contain only the `int` and `bool` elements from the tuple, and therefore, the example will write `10` and `true` to standard output.

Example 50.4. Accessing elements in Fusion containers with iterators

```
#include <boost/fusion/tuple.hpp>
#include <boost/fusion/iterator.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef tuple<int, std::string, bool, double> tuple_type;
  tuple_type t{10, "Boost", true, 3.14};
  auto it = begin(t);
  std::cout << *it << '\n';
  auto it2 = advance<boost::mpl::int_<2>>(it);
  std::cout << std::boolalpha << *it2 << '\n';
}
```

After seeing `boost::fusion::tuple` and `boost::fusion::for_each()`, it shouldn't come as a surprise to find iterators in Example 50.4. Boost.Fusion provides several free-standing functions, such as `boost::fusion::begin()` and `boost::fusion::advance()`, that work like the identically named functions from the standard library.

The number of steps the iterator is to be incremented is passed to `boost::fusion::advance()` as a template parameter. The example again uses `boost::mpl::int_` from Boost.MPL.

`boost::fusion::advance()` returns an iterator of a different type from the one that was passed to the function. That's why Example 50.4 uses a second iterator **`it2`**. You can't assign the return value from `boost::fusion::advance()` to the first iterator **`it`**. Example 50.4 writes `10` and `true` to standard output.

In addition to the functions introduced in the example, Boost.Fusion provides other functions that work with iterators. These include the following: `boost::fusion::end()`, `boost::fusion::distance()`, `boost::fusion::next()` and `boost::fusion::prior()`.

Example 50.5. A heterogeneous vector with `boost::fusion::vector`

```
#include <boost/fusion/container.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <string>
#include <iostream>

using namespace boost::fusion;

int main()
{
  typedef vector<int, std::string, bool, double> vector_type;
  vector_type v{10, "Boost", true, 3.14};
  std::cout << at<boost::mpl::int_<0>>(v) << '\n';

  auto v2 = push_back(v, 'X');
  std::cout << size(v) << '\n';
  std::cout << size(v2) << '\n';
  std::cout << back(v2) << '\n';
}
```

So far we've only seen one heterogeneous container, `boost::fusion::tuple`. Example 50.5 introduces another container, `boost::fusion::vector`.

`boost::fusion::vector` is a vector: elements are accessed with indexes. Access is not implemented using the operator `operator[]`. Instead, it's implemented using `boost::fusion::at()`, a free-standing function. The index is passed as a template parameter wrapped with `boost::mpl::int_`.

This example adds a new element of type `char` to the vector. This is done with the free-standing function `boost::fusion::push_back()`. Two parameters are passed to `boost::fusion::push_back()`: the vector to add the element to and the value to add.

`boost::fusion::push_back()` returns a new vector. The vector **`v`** isn't changed. The new vector is a copy of the original vector with the added element.

This example gets the number of elements in the vectors **`v`** and **`v2`** with `boost::fusion::size()` and writes both values to standard output. The program displays `4` and `5`. It then calls `boost::fusion::back()` to get and write the last element in **`v2`** to standard output, in this case the value is `X`.

If you look more closely at Example 50.5, you will notice that there is no difference between `boost::fusion::tuple` and `boost::fusion::vector`; they are the same. Thus, Example 50.5 will also work with `boost::fusion::tuple`.

Boost.Fusion provides additional heterogeneous containers, including: `boost::fusion::deque`, `boost::fusion::list` and `boost::fusion::set`. Example 50.6 introduces `boost::fusion::map`, a container for key/value pairs.

Example 50.6. A heterogeneous map with `boost::fusion::map`

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

using namespace boost::fusion;

int main()
{
  auto m = make_map<int, std::string, bool, double>("Boost", 10, 3.14, true);
  if (has_key<std::string>(m))
    std::cout << at_key<std::string>(m) << '\n';
  auto m2 = erase_key<std::string>(m);
  auto m3 = push_back(m2, make_pair<float>('X'));
  std::cout << std::boolalpha << has_key<std::string>(m3) << '\n';
}
```

Example 50.6 creates a heterogeneous map with `boost::fusion::map()`. The map's type is `boost::fusion::map`, which isn't written out in the example thanks to the keyword `auto`.

A map of type `boost::fusion::map` stores key/value pairs like `std::map` does. However, the keys in the Fusion map are types. A key/value pair consists of a type and a value mapped to that type. The value may be a different type than the key. In Example 50.6, the string "Boost" is mapped to the key `int`.

After the map has been created, `boost::fusion::has_key()` is called to check whether a key `std::string` exists. Then, `boost::fusion::at_key()` is called to get the value mapped to that key. Because the number 10 is mapped to `std::string`, it is written to standard output.

The key/value pair is then erased with `boost::fusion::erase_key()`. This doesn't change the map `m`. `boost::fusion::erase_key()` returns a new map which is missing the erased key/value pair.

The call to `boost::fusion::push_back()` adds a new key/value pair to the map. The key is `float` and the value is "X". `boost::fusion::make_pair()` is called to create the new key/value pair. This function is similar to `std::make_pair()`.

Finally, `boost::fusion::has_key()` is called again to check whether the map has a key `std::string`. Because it was erased, `false` is returned.

Please note that you don't need to call `boost::fusion::has_key()` to check whether a key exists before you call `boost::fusion::at_key()`. If a key is passed to `boost::fusion::at_key()` that doesn't exist in the map, you get a compiler error.

Example 50.7. Fusion adaptors for structures

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
```

```
#include <boost/mpl/int.hpp>
#include <iostream>

struct strct
{
  int i;
  double d;
};

BOOST_FUSION_ADAPT_STRUCT(strct,
  (int, i)
  (double, d)
)

using namespace boost::fusion;

int main()
{
  strct s = {10, 3.14};
  std::cout << at<boost::mpl::int_<0>>(s) << '\n';
  std::cout << back(s) << '\n';
}
```

Boost.Fusion provides several macros that let you use structures as Fusion containers. This is possible because structures can act as heterogeneous containers. Example 50.7 defines a structure which can be used as a Fusion container thanks to the macro `BOOST_FUSION_ADAPT_STRUCT`. This makes it possible to use the structure with functions like `boost::fusion::at()` or `boost::fusion::back()`.

Example 50.8. Fusion support for `std::pair`

```
#include <boost/fusion/adapted.hpp>
#include <boost/fusion/sequence.hpp>
#include <boost/mpl/int.hpp>
#include <utility>
#include <iostream>

using namespace boost::fusion;

int main()
{
  auto p = std::make_pair(10, 3.14);
  std::cout << at<boost::mpl::int_<0>>(p) << '\n';
  std::cout << back(p) << '\n';
}
```

Boost.Fusion supports structures such as `std::pair` and `boost::tuple` without having to use macros. You just need to include the header file `boost/fusion/adapted.hpp` (see Example 50.8).

Exercise

Make `debug()` write the member variables of the structures used in the program to standard output:

```
#include <boost/math/constants/constants.hpp>
#include <iostream>

struct animal
{
```

```
    std::string name;
    int legs;
    bool has_tail;
};

struct important_numbers
{
    const float pi = boost::math::constants::pi<float>();
    const double e = boost::math::constants::e<double>();
};

template <class T>
void debug(const T &t)
{
    // TODO: Write member variables of t to standard output.
}

int main()
{
    animal a{ "cat", 4, true };
    debug(a);

    important_numbers in;
    debug(in);
}
```

Part XII. Language Extensions

The following libraries extend the programming language C++.

- Boost.Coroutine makes it possible to use coroutines in C++ – something other programming languages usually support with the keyword `yield`.
- Boost.Foreach provides a range-based `for` loop, which was added to the language with C++11.
- Boost.Parameter lets you pass parameters as name/value pairs and in any order – as is allowed in Python, for example.
- Boost.Conversion provides two cast operators that replace `dynamic_cast` and allow you to differentiate between a downcast and a cross cast.

Table of Contents

51. Boost.Coroutine
52. Boost.Foreach
53. Boost.Parameter
54. Boost.Conversion

Chapter 51. Boost.Coroutine

With Boost.Coroutine it is possible to use *coroutines* in C++. Coroutines are a feature of other programming languages, which often use the keyword `yield` for coroutines. In these programming languages, `yield` can be used like `return`. However, when `yield` is used, the function remembers the location, and if the function is called again, execution continues from that location.

C++ doesn't define a keyword `yield`. However, with Boost.Coroutine it is possible to return from functions and continue later from the same location. The Boost.Asio library also uses Boost.Coroutine and benefits from coroutines.

There are two versions of Boost.Coroutine. This chapter introduces the second version, which is the current version. This version has been available since Boost 1.55.0 and replaces the first one.

Example 51.1. Using coroutines

```
#include <boost/coroutine/all.hpp>
#include <iostream>

using namespace boost::coroutines;

void cooperative(coroutine<void>::push_type &sink)
{
  std::cout << "Hello";
  sink();
  std::cout << "world";
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  std::cout << ", ";
  source();
  std::cout << "!\n";
}
```

Example 51.1 defines a function, `cooperative()`, which is called from `main()` as a coroutine. `cooperative()` returns to `main()` early and is called a second time. On the second call, it continues from where it left off.

To use `cooperative()` as a coroutine, the types `pull_type` and `push_type` are used. These types are provided by `boost::coroutines::coroutine`, which is a template that is instantiated with `void` in Example 51.1.

To use coroutines, you need `pull_type` and `push_type`. One of these types will be used to create an object that will be initialized with the function you want to use as a coroutine. The other type will be the first parameter of the coroutine function.

Example 51.1 creates an object named **`source`** of type `pull_type` in `main()`. `cooperative()` is passed to the constructor. `push_type` is used as the sole parameter in the signature of `cooperative()`.

When **source** is created, the function `cooperative()`, which is passed to the constructor, is immediately called as a coroutine. This happens because **source** is based on `pull_type`. If **source** was based on `push_type`, the constructor wouldn't call `cooperative()` as a coroutine.

`cooperative()` writes `Hello` to standard output. Afterwards, the function accesses **sink** as if it were a function. This is possible because `push_type` overloads `operator()`. While **source** in `main()` represents the coroutine `cooperative()`, **sink** in `cooperative()` represents the function `main()`. Calling **sink** makes `cooperative()` return, and `main()` continues from where `cooperative()` was called and writes a comma to standard output.

Then, `main()` calls **source** as if it were a function. Again, this is possible because of the overloaded `operator()`. This time, `cooperative()` continues from the point where it left off and writes `world` to standard output. Because there is no other code in `cooperative()`, the coroutine ends. It returns to `main()`, which writes an exclamation mark to standard output.

The result is that Example 51.1 displays `Hello, world!`

You can think of coroutines as cooperative threads. To a certain extent, the functions `main()` and `cooperative()` run concurrently. Code is executed in turns in `main()` and `cooperative()`. Instructions inside each function are executed sequentially. Thanks to coroutines, a function doesn't need to return before another function can be executed.

Example 51.2. Returning a value from a coroutine

```
#include <boost/coroutine/all.hpp>
#include <functional>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<int>::push_type &sink, int i)
{
  int j = i;
  sink(++j);
  sink(++j);
  std::cout << "end\n";
}

int main()
{
  using std::placeholders::_1;
  coroutine<int>::pull_type source{std::bind(cooperative, _1, 0)};
  std::cout << source.get() << '\n';
  source();
  std::cout << source.get() << '\n';
  source();
}
```

Example 51.2 is similar to the previous example. This time the template `boost::coroutines::coroutine` is instantiated with `int`. This makes it possible to return an `int` from the coroutine to the caller.

The direction the `int` value is passed depends on where `pull_type` and `push_type` are used. The example uses `pull_type` to instantiate an object in `main()`. `cooperative()` has access to

an object of type `push_type`. `push_type` sends a value, and `pull_type` receives a value; thus, the direction of the data transfer is set.

`cooperative()` calls **`sink`**, with a parameter of type `int`. This parameter is required because the coroutine was instantiated with the data type `int`. The value passed to **`sink`** is received from **`source`** in `main()` by using the member function `get()`, which is provided by `pull_type`.

Example 51.2 also illustrates how a function with multiple parameters can be used as a coroutine. `cooperative()` has an additional parameter of type `int`, which can't be passed directly to the constructor of `pull_type`. The example uses `std::bind()` to link the function with `pull_type`.

The example writes `1` and `2` followed by `end` to standard output.

Example 51.3. Passing two values to a coroutine

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

using boost::coroutines::coroutine;

void cooperative(coroutine<std::tuple<int, std::string>>::pull_type &source)
{
  auto args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
  source();
  args = source.get();
  std::cout << std::get<0>(args) << " " << std::get<1>(args) << '\n';
}

int main()
{
  coroutine<std::tuple<int, std::string>>::push_type sink{cooperative};
  sink(std::make_tuple(0, "aaa"));
  sink(std::make_tuple(1, "bbb"));
  std::cout << "end\n";
}
```

Example 51.3 uses `push_type` in `main()` and `pull_type` in `cooperative()`, which means data is transferred from the caller to the coroutine.

This example illustrates how multiple values can be passed. Boost.Coroutine doesn't support passing multiple values, so a tuple must be used. You need to pack multiple values into a tuple or another structure.

Example 51.3 displays `0 aaa`, `1 bbb`, and `end`.

Example 51.4. Coroutines and exceptions

```
#include <boost/coroutine/all.hpp>
#include <stdexcept>
#include <iostream>

using boost::coroutines::coroutine;

void cooperative(coroutine<void>::push_type &sink)
{
```

```
  sink();
  throw std::runtime_error("error");
}

int main()
{
  coroutine<void>::pull_type source{cooperative};
  try
  {
    source();
  }
  catch (const std::runtime_error &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

A coroutine returns immediately when an exception is thrown. The exception is transported to the caller of the coroutine where it can be caught. Thus, exceptions are no different than with regular function calls.

Example 51.4 shows how this works. This example will write the string `error` to standard output.

Chapter 52. Boost.Foreach

Boost.Foreach provides a macro that simulates the range-based `for` loop from C++11. You can use the macro `BOOST_FOREACH`, defined in `boost/foreach.hpp`, to iterate over a sequence without using iterators. If your development environment supports C++11, you can ignore Boost.Foreach.

Example 52.1. Using `BOOST_FOREACH` and `BOOST_REVERSE_FOREACH`

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

int main()
{
  std::array<int, 4> a{{0, 1, 2, 3}};

  BOOST_FOREACH(int &i, a)
    i *= i;

  BOOST_REVERSE_FOREACH(int i, a)
  {
    std::cout << i << '\n';
  }
}
```

`BOOST_FOREACH` expects two parameters. The first parameter is a variable or reference, and the second is a sequence. The type of the first parameter needs to match the type of the elements in the sequence.

Anything offering iterators, such as containers from the standard library, classifies as a sequence. Boost.Foreach uses Boost.Range instead of directly accessing the member functions `begin()` and `end()`. However, because Boost.Range is based on iterators, anything providing iterators is compatible with `BOOST_FOREACH`.

Example 52.1 iterates over an array of type `std::array` with `BOOST_FOREACH`. The first parameter passed is a reference so that you can both read and modify the elements in the array. In Example 52.1, the first loop multiplies each number by itself.

The second loop uses the macro `BOOST_REVERSE_FOREACH`, which works the same as `BOOST_FOREACH`, but iterates backwards over a sequence. The loop writes the numbers 9, 4, 1, and 0 in that order to the standard output stream.

As usual, curly brackets can be omitted if the block only consists of one statement.

Please note that you should not use operations that invalidate the iterator inside the loop. For example, elements should not be added or removed while iterating over a vector. `BOOST_FOREACH` and `BOOST_REVERSE_FOREACH` require iterators to be valid throughout the whole iteration.

Chapter 53. Boost.Parameter

Boost.Parameter makes it possible to pass parameters as key/value pairs. In addition to supporting function parameters, the library also supports template parameters. Boost.Parameter is especially useful if you are using long parameter lists, and the order and meaning of parameters is difficult to remember. Key/value pairs make it possible to pass parameters in any order. Because every value is passed with a key, the meaning of the various values is also clearer.

Example 53.1. Function parameters as key/value pairs

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char))
    (c, (double))
    (d, (std::string))
    (e, *)
  )
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_c = 3.14, _a = 1, _d = "Boost", _b = 'B', _e = true);
}
```

Example 53.1 defines a function `complicated()`, which expects five parameters. The parameters may be passed in any order. Boost.Parameter provides the macro `BOOST_PARAMETER_FUNCTION` to define such a function.

Before `BOOST_PARAMETER_FUNCTION` can be used, the parameters for the key/value pairs must be defined. This is done with the macro `BOOST_PARAMETER_NAME`, which is just passed a parameter name. The example uses `BOOST_PARAMETER_NAME` five times to define the parameter names **a**, **b**, **c**, **d**, and **e**.

Please note that the parameter names are automatically defined in the namespace `tag`. This should avoid clashes with identically named definitions in a program.

After the parameter names have been defined, `BOOST_PARAMETER_FUNCTION` is used to define the function `complicated()`. The first parameter passed to `BOOST_PARAMETER_FUNCTION` is the type of the return value. This is `void` in the example. Please note that the type must be wrapped in parentheses – the first parameter is `(void)`.

The second parameter is the name of the function being defined. The third parameter is the namespace containing the parameter names. In the fourth parameter, the parameter names are accessed to further specify them.

In Example 53.1 the fourth parameter starts with `required`, which is a keyword that makes the parameters that follow mandatory. `required` is followed by one or more pairs consisting of a parameter name and a type. It is important to wrap the type in parentheses.

Various types are used for the parameters **a**, **b**, **c**, and **d**. For example, **a** can be used to pass an `int` value to `complicated()`. No type is given for **e**. Instead, an asterisk is used, which means that the value passed may have any type. **e** is a template parameter.

After the various parameters have been passed to `BOOST_PARAMETER_FUNCTION`, the function body is defined. This is done, as usual, between a pair of curly brackets. Parameters can be accessed in the function body. They can be used like variables, with the types assigned within `BOOST_PARAMETER_FUNCTION`. Example 53.1 writes the parameters to standard output.

`complicated()` is called from `main()`. The parameters are passed to `complicated()` in an arbitrary order. Parameter names start with an underscore. Boost.Parameter uses the underscore to avoid name clashes with other variables.

Note

To pass function parameters as key/value pairs in C++, you can also use the named parameter idiom, which doesn't require a library like Boost.Parameter.

Example 53.2. Optional function parameters

```
#include <boost/parameter.hpp>
#include <string>
#include <iostream>
#include <ios>

BOOST_PARAMETER_NAME(a)
BOOST_PARAMETER_NAME(b)
BOOST_PARAMETER_NAME(c)
BOOST_PARAMETER_NAME(d)
BOOST_PARAMETER_NAME(e)

BOOST_PARAMETER_FUNCTION(
  (void),
  complicated,
  tag,
  (required
    (a, (int))
    (b, (char)))
  (optional
```

```
    (c, (double), 3.14)
    (d, (std::string), "Boost")
    (e, *, true))
)
{
  std::cout.setf(std::ios::boolalpha);
  std::cout << a << '\n';
  std::cout << b << '\n';
  std::cout << c << '\n';
  std::cout << d << '\n';
  std::cout << e << '\n';
}

int main()
{
  complicated(_b = 'B', _a = 1);
}
```

`BOOST_PARAMETER_FUNCTION` also supports defining optional parameters.

In Example 53.2 the parameters **c**, **d**, and **e** are optional. These parameters are defined in `BOOST_PARAMETER_FUNCTION` using the `optional` keyword.

Optional parameters are defined like required parameters: a parameter name is given followed by a type. As usual, the type is wrapped in parentheses. However, optional parameters need to have a default value.

With the call to `complicated()`, only the parameters **a** and **b** are passed. These are the only required parameters. As the parameters **c**, **d**, and **e** aren't used, they are set to default values.

Boost.Parameter provides macros in addition to `BOOST_PARAMETER_FUNCTION`. For example, you can use `BOOST_PARAMETER_MEMBER_FUNCTION` to define a member function, and `BOOST_PARAMETER_CONST_MEMBER_FUNCTION` to define a constant member function.

You can define functions with Boost.Parameter that try to assign values to parameters automatically. In that case, you don't need to pass key/value pairs – it is sufficient to pass values only. If the types of all values are different, Boost.Parameter can detect which value belongs to which parameter. This might require you to have a deeper knowledge of template meta programming.

Example 53.3. Template parameters as key/value pairs

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  required<tag::floating_point_type, std::is_floating_point<_>>,
  required<tag::any_type, std::is_object<_>>
```

```
> complicated_signature;

template <class A, class B, class C>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<int>,
    any_type<bool>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Example 53.3 uses Boost.Parameter to pass template parameters as key/value pairs. As with functions, it is possible to pass the template parameters in any order.

The example defines a class `complicated`, which expects three template parameters. Because the order of the parameters doesn't matter, they are called `A`, `B`, and `C`. `A`, `B`, and `C` aren't the names of the parameters that will be used when the class template is accessed. As with functions, the parameter names are defined using a macro. For template parameters, `BOOST_PARAMETER_TEMPLATE_KEYWORD` is used. Example 53.3 defines three parameter names `integral_type`, `floating_point_type`, and `any_type`.

After the parameter names have been defined, you must specify the types that may be passed. For example, the parameter `integral_type` can be used to pass types such as `int` or `long`, but not a type like `std::string`. `boost::parameter::parameters` is used to create a signature that refers to the parameter names and defines which types may be passed with each of them.

`boost::parameter::parameters` is a tuple that describes parameters. Required parameters are marked with `boost::parameter::required`.

`boost::parameter::required` requires two parameters. The first is the name of the parameter defined with `BOOST_PARAMETER_TEMPLATE_KEYWORD`. The second identifies the type the parameter may be set to. For example, `integral_type` may be set to an integral type. This requirement is expressed with `std::is_integral<_>`. `std::is_integral<_>` is a lambda function based on Boost.MPL. `boost::mpl::placeholders::_` is a placeholder provided by this library. If the type to which `integral_type` is set is passed to `std::is_integral` instead of `boost::mpl::placeholders::_`, and the result is true, a valid type is used. The requirements for the other parameters `floating_point_type` and `any_type` are defined similarly.

After the signature has been created and defined as `complicated_signature`, it is used by the class `complicated`. First, the signature is bound with `complicated_signature::bind` to the template parameters `A`, `B`, and `C`. The new type, `args`, represents the connection between the template parameters passed and the requirements that must be met by the template

parameters. Next, `args` is accessed to get the parameter values. This is done with `boost::parameter::value_type`. `boost::parameter::value_type` expects `args` and a parameter to be passed. The parameter determines the type created. In Example 53.3, the type definition `integral_type` in the class `complicated` is used to get the type that was passed with the parameter `integral_type` to `complicated`.

`main()` accesses `complicated` to instantiate the class. The parameter `integral_type` is set to `int`, `floating_point_type` to `double`, and `any_type` to `bool`. The order of the parameters passed doesn't matter. The type definitions `integral_type`, `floating_point_type`, and `any_type` are then accessed by `typeid` to get their underlying types. Compiled with Visual C++ 2013, the example writes `int`, `double` and `bool` to standard output.

Example 53.4. Optional template parameters

```
#include <boost/parameter.hpp>
#include <boost/mpl/placeholders.hpp>
#include <type_traits>
#include <typeinfo>
#include <iostream>

BOOST_PARAMETER_TEMPLATE_KEYWORD(integral_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(floating_point_type)
BOOST_PARAMETER_TEMPLATE_KEYWORD(any_type)

using namespace boost::parameter;
using boost::mpl::placeholders::_;

typedef parameters<
  required<tag::integral_type, std::is_integral<_>>,
  optional<tag::floating_point_type, std::is_floating_point<_>>,
  optional<tag::any_type, std::is_object<_>>
> complicated_signature;

template <class A, class B = void_, class C = void_>
class complicated
{
public:
  typedef typename complicated_signature::bind<A, B, C>::type args;
  typedef typename value_type<args, tag::integral_type>::type integral_type;
  typedef typename value_type<args, tag::floating_point_type, float>::type
    floating_point_type;
  typedef typename value_type<args, tag::any_type, bool>::type any_type;
};

int main()
{
  typedef complicated<floating_point_type<double>, integral_type<short>> c;
  std::cout << typeid(c::integral_type).name() << '\n';
  std::cout << typeid(c::floating_point_type).name() << '\n';
  std::cout << typeid(c::any_type).name() << '\n';
}
```

Example 53.4 introduces optional template parameters. The signature uses `boost::parameter::optional` for the optional template parameters. The optional template parameters from `complicated` are set to `boost::parameter::void_`, and `boost::parameter::value_type` is given a default value. This default value is the type an optional parameter will be set to if the type isn't otherwise set.

`complicated` is instantiated in `main()`. This time only the parameters `integral_type` and `floating_point_type` are used. `any_type` is not used. Compiled with Visual C++ 2013, the example writes `short` for `integral_type`, `double` for `floating_point_type`, and `bool` for `any_type` to standard output.

Boost.Parameter can automatically detect template parameters. You can create signatures that allow types to be automatically assigned to parameters. As with function parameters, deeper knowledge in template meta programming is required to do this.

Chapter 54. Boost.Conversion

Boost.Conversion defines the cast operators `boost::polymorphic_cast` and `boost::polymorphic_downcast` in the header file `boost/cast.hpp`. They are designed to handle type casts – usually done with `dynamic_cast` – more precisely.

Example 54.1. Down and cross casts with `dynamic_cast`

```
struct base1 { virtual ~base1() = default; };
struct base2 { virtual ~base2() = default; };
struct derived : public base1, public base2 {};

void downcast(base1 *b1)
{
  derived *d = dynamic_cast<derived*>(b1);
}

void crosscast(base1 *b1)
{
  base2 *b2 = dynamic_cast<base2*>(b1);
}

int main()
{
  derived *d = new derived;
  downcast(d);

  base1 *b1 = new derived;
  crosscast(b1);
}
```

Example 54.1 uses the cast operator `dynamic_cast` twice: In `downcast()`, it transforms a pointer pointing to a base class to one pointing to a derived class. In `crosscast()`, it transforms a pointer pointing to a base class to one pointing to a different base class. The first transformation is a *downcast*, and the second is a *cross cast*. The cast operators from Boost.Conversion let you distinguish a downcast from a cross cast.

Example 54.2. Down and cross casts with `polymorphic_downcast` and `polymorphic_cast`

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

struct base1 { virtual ~base1() = default; };
struct base2 { virtual ~base2() = default; };
struct derived : public base1, public base2 {};

void downcast(base1 *b1)
{
  derived *d = boost::polymorphic_downcast<derived*>(b1);
}

void crosscast(base1 *b1)
{
  base2 *b2 = boost::polymorphic_cast<base2*>(b1);
}

int main()
{
  derived *d = new derived;
  downcast(d);

  base1 *b1 = new derived;
```

```
  crosscast(b1);
}
```

`boost::polymorphic_downcast` (see Example 54.2) can only be used for downcasts because it uses `static_cast` to perform the cast. Because `static_cast` does not dynamically check the cast for validity, `boost::polymorphic_downcast` must only be used if the cast is safe. In debug builds, `boost::polymorphic_downcast` uses `dynamic_cast` and `assert()` to make sure the type cast is valid. This test is only performed if the macro `NDEBUG` is not defined, which is usually the case for debug builds.

`boost::polymorphic_cast` is required for cross casts. `boost::polymorphic_cast` uses `dynamic_cast`, which is the only cast operator that can perform a cross cast. It is better to use `boost::polymorphic_cast` instead of `dynamic_cast` because the former throws an exception of type `std::bad_cast` in case of an error, while `dynamic_cast` returns a null pointer if the type cast fails.

Use `boost::polymorphic_downcast` and `boost::polymorphic_cast` only to convert pointers; otherwise, use `dynamic_cast`. Because `boost::polymorphic_downcast` is based on `static_cast`, it cannot convert objects of a base class to objects of a derived class. Also, it does not make sense to use `boost::polymorphic_cast` to convert types other than pointers because `dynamic_cast` will throw an exception of type `std::bad_cast` if a cast fails.

Part XIII. Error Handling

The following libraries support error handling.

- Boost.System provides classes to describe and identify errors. Since C++11, these classes have been part of the standard library.
- Boost.Exception makes it possible to attach data to exceptions after they have been thrown.

Table of Contents

55. Boost.System
56. Boost.Exception

Chapter 55. Boost.System

Boost.System is a library that, in essence, defines four classes to identify errors. All four classes were added to the standard library with C++11. If your development environment supports C++11, you don't need to use Boost.System. However, since many Boost libraries use Boost.System, you might encounter Boost.System through those other libraries.

`boost::system::error_code` is the most basic class in Boost.System; it represents operating system-specific errors. Because operating systems typically enumerate errors, `boost::system::error_code` saves an error code in a variable of type `int`. Example 55.1 illustrates how to use this class.

Example 55.1. Using `boost::system::error_code`

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  std::cout << ec.value() << '\n';
}
```

Example 55.1 defines the function `fail()`, which is used to return an error. In order for the caller to detect whether `fail()` failed, an object of type `boost::system::error_code` is passed by reference. Many functions that are provided by Boost libraries use `boost::system::error_code` like this. For example, Boost.Asio provides the function `boost::asio::ip::host_name()`, to which you can pass an object of type `boost::system::error_code`.

Boost.System defines numerous error codes in the namespace `boost::system::errc`. Example 55.1 assigns the error code `boost::system::errc::not_supported` to **`ec`**. Because **`boost::system::errc::not_supported`** is a number and **`ec`** is an object of type `boost::system::error_code`, the function `boost::system::errc::make_error_code()` is called. This function creates an object of type `boost::system::error_code` with the respective error code.

In `main()`, `value()` is called on **`ec`**. This member function returns the error code stored in the object.

By default, 0 means no error. Every other number refers to an error. Error code values are operating system dependent. Refer to the documentation for your operating system for a description of error codes.

In addition to `value()`, `boost::system::error_code` provides the member function `category()`, which returns an object of type `boost::system::error_category`.

Error codes are simply numeric values. While operating system manufacturers such as Microsoft are able to guarantee the uniqueness of system error codes, keeping error codes unique across all existing applications is virtually impossible for application developers. It would require a central database filled with error codes from all software developers around the world to avoid reusing the same codes for different errors. Because this is impractical, error categories exist.

Error codes of type `boost::system::error_code` belong to a category that can be retrieved with the member function `category()`. Errors created with `boost::system::errc::make_error_code()` automatically belong to the generic category. This is the category errors belong to if they aren't assigned to another category explicitly.

Example 55.2. Using `boost::system::error_category`

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}

int main()
{
  error_code ec;
  fail(ec);
  std::cout << ec.value() << '\n';
  std::cout << ec.category().name() << '\n';
}
```

As shown in Example 55.2, `category()` returns an error's category. This is an object of type `boost::system::error_category`. There are only a few member functions. For example, `name()` retrieves the name of the category. Example 55.2 writes `generic` to standard output.

You can also use the free-standing function `boost::system::generic_category()` to access the generic category.

Boost.System provides a second category. If you call the free-standing function `boost::system::system_category()`, you get a reference to the system category. If you write the category's name to standard output, `system` is displayed.

Errors are uniquely identified by the error code and the error category. Because error codes are only required to be unique within a category, you should create a new category whenever you want to define error codes specific to your program. This makes it possible to use error codes that do not interfere with error codes from other developers.

Example 55.3. Creating error categories

```
#include <boost/system/error_code.hpp>
#include <string>
#include <iostream>
```

```
class application_category :
  public boost::system::error_category
{
public:
  const char *name() const noexcept { return "my app"; }
  std::string message(int ev) const { return "error message"; }
};

application_category cat;

int main()
{
  boost::system::error_code ec{129, cat};
  std::cout << ec.value() << '\n';
  std::cout << ec.category().name() << '\n';
}
```

A new error category is defined by creating a class derived from `boost::system::error_category`. This requires you to define various member functions. At a minimum, the member functions `name()` and `message()` must be supplied because they are defined as pure virtual member functions in `boost::system::error_category`. For additional member functions, the default behavior can be overridden if required.

While `name()` returns the name of the error category, `message()` is used to retrieve the error description for a particular error code. Unlike Example 55.3, the parameter **ev** is usually evaluated to return a description based on the error code.

An object of the type of the newly created error category can be used to initialize an error code. Example 55.3 defines the error code **ec** using the new category `application_category`. Therefore, error code 129 is no longer a generic error; instead, its meaning is defined by the developer of the new error category.

Note

To compile Example 55.3 with Visual C++ 2013, remove the keyword `noexcept`. This version of the Microsoft compiler doesn't support `noexcept`.

`boost::system::error_code` provides a member function called `default_error_condition()`, that returns an object of type `boost::system::error_condition`. The interface of `boost::system::error_condition` is almost identical to the interface of `boost::system::error_code`. The only difference is the member function `default_error_condition()`, which is only provided by `boost::system::error_code`.

Example 55.4. Using `boost::system::error_condition`

```
#include <boost/system/error_code.hpp>
#include <iostream>

using namespace boost::system;

void fail(error_code &ec)
{
  ec = errc::make_error_code(errc::not_supported);
}
```

```
int main()
{
  error_code ec;
  fail(ec);
  boost::system::error_condition ecnd = ec.default_error_condition();
  std::cout << ecnd.value() << '\n';
  std::cout << ecnd.category().name() << '\n';
}
```

`boost::system::error_condition` is used just like `boost::system::error_code`. That's why it's possible, as shown in Example 55.4, to call the member functions `value()` and `category()` for an object of type `boost::system::error_condition`.

While the class `boost::system::error_code` is used for platform-dependent error codes, `boost::system::error_condition` is used to access platform-independent error codes. The member function `default_error_condition()` translates a platform-dependent error code into a platform-independent error code of type `boost::system::error_condition`.

You can use `boost::system::error_condition` to identify errors that are platform independent. Such an error could be, for example, a failed access to a non-existing file. While operating systems may provide different interfaces to access files and may return different error codes, trying to access a non-existing file is an error on all operating systems. The error code returned from operating system specific interfaces is stored in `boost::system::error_code`. The error code that describes the failed access to a non-existing file is stored in `boost::system::error_condition`.

The last class provided by Boost.System is `boost::system::system_error`, which is derived from `std::runtime_error`. It can be used to transport an error code of type `boost::system::error_code` in an exception.

Example 55.5. Using `boost::system::system_error`

```
#include <boost/system/error_code.hpp>
#include <boost/system/system_error.hpp>
#include <iostream>

using namespace boost::system;

void fail()
{
  throw system_error{errc::make_error_code(errc::not_supported)};
}

int main()
{
  try
  {
    fail();
  }
  catch (system_error &e)
  {
    error_code ec = e.code();
    std::cerr << ec.value() << '\n';
    std::cerr << ec.category().name() << '\n';
  }
}
```

In Example 55.5, the free-standing function `fail()` has been changed to throw an exception of type `boost::system::system_error` in case of an error. This exception can transport an error code of type `boost::system::error_code`. The exception is caught in `main()`, which writes the error code and the error category to standard error. There is a second variant of the function `boost::asio::ip::host_name()` that works just like this.

Chapter 56. Boost.Exception

The library Boost.Exception provides a new exception type, `boost::exception`, that lets you add data to an exception after it has been thrown. This type is defined in `boost/exception/exception.hpp`. Because Boost.Exception spreads its classes and functions over multiple header files, the following examples access the master header file `boost/exception/all.hpp` to avoid including header files one by one.

Boost.Exception supports the mechanism from the C++11 standard that transports an exception from one thread to another. `boost::exception_ptr` is similar to `std::exception_ptr`. However, Boost.Exception isn't a full replacement for the header file `exception` from the standard library. For example, Boost.Exception is missing support for nested exceptions of type `std::nested_exception`.

Note

To compile the examples in this chapter with Visual C++ 2013, remove the keyword `noexcept`. This version of the Microsoft compiler doesn't support `noexcept` yet.

Example 56.1. Using `boost::exception`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public boost::exception, public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    throw allocation_failed{};
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}
```

```
int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Example 56.1 calls the function `write_lots_of_zeros()`, which in turn calls `allocate_memory()`. `allocate_memory()` allocates memory dynamically. The function passes `std::nothrow` to `new` and checks whether the return value is 0. If memory allocation fails, an exception of type `allocation_failed` is thrown. `allocation_failed` replaces the exception `std::bad_alloc` thrown by default if `new` fails to allocate memory.

`write_lots_of_zeros()` calls `allocate_memory()` to try and allocate a memory block with the greatest possible size. This is done with the help of `max()` from `std::numeric_limits`. The example intentionally tries to allocate that much memory to make the allocation fail.

`allocation_failed` is derived from `boost::exception` and `std::exception`. Deriving the class from `std::exception` is not necessary. `allocation_failed` could have also been derived from a class from a different class hierarchy in order to embed it in an existing framework. While Example 56.1 uses the class hierarchy defined by the standard, deriving `allocation_failed` solely from `boost::exception` would have been sufficient.

If an exception of type `allocation_failed` is caught, `allocate_memory()` must be the origin of the exception, since it is the only function that throws exceptions of this type. In programs that have many functions calling `allocate_memory()`, knowing the type of the exception is no longer sufficient to debug the program effectively. In those cases, it would help to know which function tried to allocate more memory than `allocate_memory()` could provide.

The challenge is that `allocate_memory()` does not have any additional information, such as the caller name, to add to the exception. `allocate_memory()` can't enrich the exception. This can only be done in the calling context.

With Boost.Exception, data can be added to an exception at any time. You just need to define a type based on `boost::error_info` for each bit of data you need to add.

`boost::error_info` is a template that expects two parameters. The first parameter is a *tag* that uniquely identifies the newly created type. This is typically a structure with a unique name. The second parameter refers to the type of the value stored inside the exception. Example 56.1 defines a new type, `errmsg_info` – uniquely identifiable via the structure `tag_errmsg` – that stores a string of type `std::string`.

In the `catch` handler of `write_lots_of_zeros()`, `errmsg_info` is used to create an object that is initialized with the string "writing lots of zeros failed". This object is then added to the exception of type `boost::exception` using `operator<<`. Then the exception is re-thrown.

Now, the exception doesn't just denote a failed memory allocation. It also says that the memory allocation failed when the program tried to write lots of zeros in the function `write_lots_of_zeros()`. Knowing which function called `allocate_memory()` makes debugging larger programs easier.

To retrieve all available data from an exception, the function `boost::diagnostic_information()` can be called in the `catch` handler of `main()`. `boost::diagnostic_information()` calls the member function `what()` for each exception passed to it and accesses all of the additional data stored inside the exception. `boost::diagnostic_information()` returns a string of type `std::string`, which, for example, can be written to standard error.

When compiled with Visual C++ 2013, Example 56.1 will display the following message:

```
Throw location unknown (consider using BOOST_THROW_EXCEPTION)
Dynamic exception type: struct allocation_failed
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

The message contains the type of the exception, the error message retrieved from `what()`, and the description, including the name of the structure.

`boost::diagnostic_information()` checks at run time whether or not a given exception is derived from `std::exception`. `what()` will only be called if that is the case.

The name of the function that threw the exception of type `allocation_failed` is unknown.

Boost.Exception provides a macro to throw an exception that contains not only the name of the function, but also additional data such as the file name and the line number.

Example 56.2. More data with `BOOST_THROW_EXCEPTION`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
```

```
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << boost::diagnostic_information(e);
  }
}
```

Using the macro `BOOST_THROW_EXCEPTION` instead of `throw`, data such as function name, file name, and line number are automatically added to the exception. But this only works if the compiler supports macros for the additional data. While macros such as `__FILE__` and `__LINE__` have been standardized since C++98, the macro `__func__`, which gets the name of the current function, only became standard with C++11. Because many compilers provided such a macro before C++11, `BOOST_THROW_EXCEPTION` tries to identify the underlying compiler and use the corresponding macro if it exists.

Compiled with Visual C++ 2013, Example 56.2 displays the following message:

```
main.cpp(20): Throw in function char *__cdecl allocate_memory(unsigned int)
Dynamic exception type: class boost::exception_detail::clone_impl<struct boos
std::exception::what: allocation failed
[struct tag_errmsg *] = writing lots of zeros failed
```

In Example 56.2, `allocation_failed` is no longer derived from `boost::exception`. `BOOST_THROW_EXCEPTION` accesses the function `boost::enable_error_info()`, which identifies whether or not an exception is derived from `boost::exception`. If not, it creates a new exception type derived from the specified type and `boost::exception`. This is why the message shown above contains a different exception type than `allocation_failed`.

Example 56.3. Selectively accessing data with `boost::get_error_info()`

```
#include <boost/exception/all.hpp>
#include <exception>
#include <new>
#include <string>
#include <algorithm>
#include <limits>
#include <iostream>

typedef boost::error_info<struct tag_errmsg, std::string> errmsg_info;

struct allocation_failed : public std::exception
{
```

```
  const char *what() const noexcept { return "allocation failed"; }
};

char *allocate_memory(std::size_t size)
{
  char *c = new (std::nothrow) char[size];
  if (!c)
    BOOST_THROW_EXCEPTION(allocation_failed{});
  return c;
}

char *write_lots_of_zeros()
{
  try
  {
    char *c = allocate_memory(std::numeric_limits<std::size_t>::max());
    std::fill_n(c, std::numeric_limits<std::size_t>::max(), 0);
    return c;
  }
  catch (boost::exception &e)
  {
    e << errmsg_info{"writing lots of zeros failed"};
    throw;
  }
}

int main()
{
  try
  {
    char *c = write_lots_of_zeros();
    delete[] c;
  }
  catch (boost::exception &e)
  {
    std::cerr << *boost::get_error_info<errmsg_info>(e);
  }
}
```

Example 56.3 does not use `boost::diagnostic_information()`, it uses `boost::get_error_info()` to directly access the error message of type `errmsg_info`. Because `boost::get_error_info()` returns a smart pointer of type `boost::shared_ptr`, `operator*` is used to fetch the error message. If the parameter passed to `boost::get_error_info()` is not of type `boost::exception`, a null pointer is returned. If the macro `BOOST_THROW_EXCEPTION` is always used to throw an exception, the exception will always be derived from `boost::exception` – there is no need to check the returned smart pointer for null in that case.

Part XIV. Number Handling

The following libraries are all about working with numbers.

- Boost.Integer provides integral types to, for example, specify the exact number of bytes used by a variable.
- Boost.Accumulators provides accumulators that you can pass numbers to when you are calculating values like the mean or standard deviation.
- Boost.MinMax lets you get the smallest and largest number in a container with one function call.
- Boost.Random provides random number generators.
- Boost.NumericConversion provides a cast operator that protects against unintended overflows.

Table of Contents

57. Boost.Integer
58. Boost.Accumulators
59. Boost.MinMax
60. Boost.Random
61. Boost.NumericConversion

Chapter 57. Boost.Integer

Boost.Integer provides the header file `boost/cstdint.hpp`, which defines specialized types for integers. These definitions originate from the C99 standard. This is a version of the standard for the C programming language that was released in 1999. Because the first version of the C++ standard was released in 1998, it does not include the specialized integer types defined in C99.

C99 defines types in the header file `stdint.h`. This header file was taken into C++11. In C++, it is called `cstdint`. If your development environment supports C++11, you can access `cstdint`, and you don't have to use `boost/cstdint.hpp`.

Example 57.1. Types for integers with number of bits

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

int main()
{
  boost::int8_t i8 = 1;
  std::cout << sizeof(i8) << '\n';

#ifndef BOOST_NO_INT64_T
  boost::uint64_t ui64 = 1;
  std::cout << sizeof(ui64) << '\n';
#endif

  boost::int_least8_t il8 = 1;
  std::cout << sizeof(il8) << '\n';

  boost::uint_least32_t uil32 = 1;
  std::cout << sizeof(uil32) << '\n';

  boost::int_fast8_t if8 = 1;
  std::cout << sizeof(if8) << '\n';

  boost::uint_fast16_t uif16 = 1;
  std::cout << sizeof(uif16) << '\n';
}
```

The types from `boost/cstdint.hpp` are defined in the namespace `boost`. They can be divided into three categories:

- Types such as `boost::int8_t` and `boost::uint64_t` carry the exact memory size in their names. Thus, `boost::int8_t` contains exactly 8 bits, and `boost::uint64_t` contains exactly 64 bits.
- Types such as `boost::int_least8_t` and `boost::uint_least32_t` contain at least as many bits as their names say. It is possible that the memory size of `boost::int_least8_t` will be greater than 8 bits and that of `boost::uint_least32_t` will be greater than 32 bits.
- Types such as `boost::int_fast8_t` and `boost::uint_fast16_t` also have a minimum size. Their actual size is set to a value that guarantees the best performance. Example 57.1 compiled with Visual C++ 2013 and run on a 64-bit Windows 7 system displays `4` for `sizeof(uif16)`.

Please note that 64-bit types aren't available on all platforms. You can check with the macro `BOOST_NO_INT64_T` whether 64-bit types are available or not.

Example 57.2. More specialized types for integers

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

int main()
{
  boost::intmax_t imax = 1;
  std::cout << sizeof(imax) << '\n';

  std::cout << sizeof(UINT8_C(1)) << '\n';

#ifndef BOOST_NO_INT64_T
  std::cout << sizeof(INT64_C(1)) << '\n';
#endif
}
```

Boost.Integer defines two types, `boost::intmax_t` and `boost::uintmax_t`, for the maximum width integer types available on a platform. Example 57.2 compiled with Visual C++ 2013 and run on a 64-bit Windows 7 system displays `8` for `sizeof(imax)`. Thus, the biggest type for integers contains 64 bits.

Furthermore, Boost.Integer provides macros to use integers as literals with certain types. If an integer is written in C++ code, by default it uses the type `int` and allocates at least 4 bytes. Macros like `UINT8_C` and `INT64_C` make it possible to set a minimum size for integers as literals. Example 57.2 returns at least 1 for `sizeof(UINT8_C(1))` and at least 8 for `sizeof(INT64_C(1))`.

Boost.Integer provides additional header files that mainly define classes used for template meta programming.

Chapter 58. Boost.Accumulators

Boost.Accumulators provides classes to process samples. For example, you can find the largest or smallest sample, or calculate the total of all samples. While the standard library supports some of these operations, Boost.Accumulators also supports statistical calculations, such as mean and standard deviation.

The library is called Boost.Accumulators because the *accumulator* is an essential concept. An accumulator is a container that calculates a new result every time a value is inserted. The value isn't necessarily stored in the accumulator. Instead the accumulator continuously updates intermediary results as it is fed new values.

Boost.Accumulators contains three parts:

- The framework provides the overall structure of the library. It provides the class `boost::accumulators::accumulator_set`, which is always used with Boost.Accumulators. While you need to know about this and a few other classes from the framework, the details don't matter unless you want to develop your own accumulators. The header file `boost/accumulators/accumulators.hpp` gives you access to `boost::accumulators::accumulator_set` and other classes from the framework.
- Boost.Accumulators provides numerous accumulators that perform calculations. You can access and use all of these accumulators once you include `boost/accumulators/statistics.hpp`.
- Boost.Accumulators provides operators to, for example, multiply a complex number of type `std::complex` with an `int` value or add two vectors. The header file `boost/accumulators/numeric/functional.hpp` defines operators for `std::complex`, `std::valarray`, and `std::vector`. You don't need to include the header file yourself because it is included in the header files for the accumulators. However, you have to define the macros `BOOST_NUMERIC_FUNCTIONAL_STD_COMPLEX_SUPPORT`, `BOOST_NUMERIC_FUNCTIONAL_STD_VALARRAY_SUPPORT`, and `BOOST_NUMERIC_FUNCTIONAL_STD_VECTOR_SUPPORT` to make the operators available.

All classes and functions provided by Boost.Accumulators are defined in `boost::accumulators` or nested namespaces. For example, all accumulators are defined in `boost::accumulators::tag`.

Example 58.1. Counting with `boost::accumulators::tag::count`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<int, features<tag::count>> acc;
  acc(4);
  acc(-6);
```

```
  acc(9);
  std::cout << count(acc) << '\n';
}
```

Example 58.1 uses `boost::accumulators::tag::count`, a simple accumulator that counts the number of values passed to it. Thus, since three values are passed, this example writes `3` to standard output. To use an accumulator, you access the class `boost::accumulators::accumulator_set`, which is a template that expects as its first parameter the type of the values that will be processed. Example 58.1 passes `int` as the first parameter.

The second parameter specifies the accumulators you want to use. You can use multiple accumulators. The class name `boost::accumulators::accumulator_set` indicates that any number of accumulators can be managed.

Strictly speaking, you specify *features*, not accumulators. Features define what should be calculated. You determine the what, not the how. There can be different implementations for features. The implementations are the accumulators.

Example 58.1 uses `boost::accumulators::tag::count` to select an accumulator that counts values. If several accumulators exist that can count values, Boost.Accumulators selects the default accumulator.

Please note that you can't pass features directly to `boost::accumulators::accumulator_set`. You need to use `boost::accumulators::features`.

An object of type `boost::accumulators::accumulator_set` can be used like a function. Values can be passed by calling `operator()`. They are immediately processed. The values passed must have the same type as was passed as the first template parameter to `boost::accumulators::accumulator_set`.

For every feature, there is an identically named *extractor*. An extractor receives the current result of an accumulator. Example 58.1 uses the extractor `boost::accumulators::count()`. The only parameter passed is **`acc`**. `boost::accumulators::count()` returns 3.

Example 58.2. Using `mean` and `variance`

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>> acc;
  acc(8);
  acc(9);
  acc(10);
  acc(11);
  acc(12);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Example 58.2 uses the two features `boost::accumulators::tag::mean` and `boost::accumulators::tag::variance` to calculate the mean and the variance of five values. The example writes `10` and `2` to standard output.

The variance is 2 because Boost.Accumulators assigns a weight of 0.2 to each of the five values. The accumulator selected with `boost::accumulators::tag::variance` uses weights. If weights are not set explicitly, all values are given the same weight.

Example 58.3. Calculating the weighted variance

```
#include <boost/accumulators/accumulators.hpp>
#include <boost/accumulators/statistics.hpp>
#include <iostream>

using namespace boost::accumulators;

int main()
{
  accumulator_set<double, features<tag::mean, tag::variance>, int> acc;
  acc(8, weight = 1);
  acc(9, weight = 1);
  acc(10, weight = 4);
  acc(11, weight = 1);
  acc(12, weight = 1);
  std::cout << mean(acc) << '\n';
  std::cout << variance(acc) << '\n';
}
```

Example 58.3 passes `int` as a third template parameter to `boost::accumulators::accumulator_set`. This parameter specifies the data type of the weights. In this example, weights are assigned to every value.

Boost.Accumulators uses Boost.Parameter to pass additional parameters, such as weights, as name/value pairs. The parameter name for weights is **`weight`**. You can treat the parameter like a variable and assign a value. The name/value pair is passed as an additional parameter after every value to the accumulator.

In Example 58.3, the value 10 has a weight of 4 while all other values have a weight of 1. The mean is still 10 since weights don't matter for means. However, the variance is now 1.25. It has decreased compared to the previous example because the middle value has a higher weight than the other values.

Boost.Accumulators provides many more accumulators. They are used like the accumulators introduced in this chapter. The documentation of the library contains an overview on all available accumulators.

Chapter 59. Boost.MinMax

Boost.MinMax provides an algorithm to find the minimum and the maximum of two values using only one function call, which is more efficient than calling `std::min()` and `std::max()`.

Boost.MinMax is part of C++11. You find the algorithms from this Boost library in the header file `algorithm` if your development environment supports C++11.

Example 59.1. Using `boost::minmax()`

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

int main()
{
  int i = 2;
  int j = 1;

  boost::tuples::tuple<const int&, const int&> t = boost::minmax(i, j);

  std::cout << t.get<0>() << '\n';
  std::cout << t.get<1>() << '\n';
}
```

`boost::minmax()` computes the minimum and maximum of two objects. While both `std::min()` and `std::max()` return only one value, `boost::minmax()` returns two values as a tuple. The first reference in the tuple points to the minimum and the second to the maximum. Example 59.1 writes 1 and 2 to the standard output stream.

`boost::minmax()` is defined in `boost/algorithm/minmax.hpp`.

Example 59.2. Using `boost::minmax_element()`

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

int main()
{
  typedef std::array<int, 4> array;
  array a{{2, 3, 0, 1}};

  std::pair<array::iterator, array::iterator> p =
    boost::minmax_element(a.begin(), a.end());

  std::cout << *p.first << '\n';
  std::cout << *p.second << '\n';
}
```

Just as the standard library offers algorithms to find the minimum and maximum values in a container, Boost.MinMax offers the same functionality with only one call to the function `boost::minmax_element()`.

Unlike `boost::minmax()`, `boost::minmax_element()` returns a `std::pair` containing two iterators. The first iterator points to the minimum and the second points to the maximum. Thus,

Example 59.2 writes `0` and `3` to the standard output stream.

`boost::minmax_element()` is defined in `boost/algorithm/minmax_element.hpp`.

Both `boost::minmax()` and `boost::minmax_element()` can be called with a third parameter that specifies how objects should be compared. Thus, these functions can be used like the algorithms from the standard library.

Chapter 60. Boost.Random

The library Boost.Random provides numerous random number generators that allow you to decide how random numbers should be generated. It was always possible in C++ to generate random numbers with `std::rand()` from `cstdlib`. However, with `std::rand()` the way random numbers are generated depends on how the standard library was implemented.

You can use all of the random number generators and other classes and functions from Boost.Random when you include the header file `boost/random.hpp`.

Large parts of this library were added to the standard library with C++11. If your development environment supports C++11, you can rewrite the Boost.Random examples in this chapter by including the header file `random` and accessing the namespace `std`.

Example 60.1. Pseudo-random numbers with `boost::random::mt19937`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  std::cout << gen() << '\n';
}
```

Example 60.1 accesses the random number generator `boost::random::mt19937`. The operator `operator()` generates a random number, which is written to standard output.

The random numbers generated by `boost::random::mt19937` are integers. Whether integers or floating point numbers are generated depends on the particular generator you use. All random number generators define the type `result_type` to determine the type of the random numbers. The `result_type` for `boost::random::mt19937` is `boost::uint32_t`.

All random number generators provide two member functions: `min()` and `max()`. These functions return the smallest and largest number that can be generated by that random number generator.

Nearly all of the random number generators provided by Boost.Random are *pseudo-random number generators*. Pseudo-random number generators don't generate real random numbers. They are based on algorithms that generate seemingly random numbers. `boost::random::mt19937` is one of these pseudo-random number generators.

Pseudo-random number generators typically have to be initialized. If they are initialized with the same values, they return the same random numbers. That's why in Example 60.1 the return value of `std::time()` is passed to the constructor of `boost::random::mt19937`. This should ensure that when the program is run at different times, different random numbers will be generated.

Pseudo-random numbers are good enough for most use cases. `std::rand()` is also based on a pseudo-random number generator, which must be initialized with `std::srand()`. However, Boost.Random provides a random number generator that can generate real random numbers, as long as the operating system has a source to generate real random numbers.

Example 60.2. Real random numbers with `boost::random::random_device`

```
#include <boost/random/random_device.hpp>
#include <iostream>

int main()
{
  boost::random::random_device gen;
  std::cout << gen() << '\n';
}
```

`boost::random::random_device` is a *non-deterministic random number generator*, which is a random number generator that can generate real random numbers. There is no algorithm that needs to be initialized. Thus, predicting the random numbers is impossible. Non-deterministic random number generators are often used in security-related applications.

`boost::random::random_device` calls operating system functions to generate random numbers. If, as in Example 60.2, the default constructor is called, `boost::random::random_device` uses the cryptographic service provider MS_DEF_PROV on Windows and `/dev/urandom` on Linux as a source.

If you want to use another source, call the constructor of `boost::random::random_device`, which expects a parameter of type `std::string`. How this parameter is interpreted depends on the operating system. On Windows, it must be the name of a cryptographic service provider, on Linux a path to a device.

Please note that `boost/random/random_device.hpp` must be included if you want to use the class `boost::random::random_device`. This class is not made available by `boost/random.hpp`.

Example 60.3. The random numbers 0 and 1 with `bernoulli_distribution`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  boost::random::bernoulli_distribution<> dist;
  std::cout << dist(gen) << '\n';
}
```

Example 60.3 uses the pseudo-random number generator `boost::random::mt19937`. In addition, a *distribution* is used. Distributions are Boost.Random classes that map the range of random numbers from a random number generator to another range. While random number generators like `boost::random::mt19937` have a built-in lower and upper limit for random

numbers that can be seen using `min()` and `max()`, you may need random numbers in a different range.

Example 60.3 simulates throwing a coin. Because a coin has only two sides, the random number generator should return 0 or 1. `boost::random::bernoulli_distribution` is a distribution that returns one of two possible results.

Distributions are used like random number generators: you call the operator `operator()` to receive a random number. However, you must pass a random number generator as a parameter to a distribution. In Example 60.3, **`dist`** uses the random number generator **`gen`** to return either 0 or 1.

Example 60.4. Random numbers between 1 and 100 with `uniform_int_distribution`

```
#include <boost/random.hpp>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  std::time_t now = std::time(0);
  boost::random::mt19937 gen{static_cast<std::uint32_t>(now)};
  boost::random::uniform_int_distribution<> dist{1, 100};
  std::cout << dist(gen) << '\n';
}
```

Boost.Random provides numerous distributions. Example 60.4 uses a distribution that is often needed: `boost::random::uniform_int_distribution`. This distribution lets you define the range of random numbers you need. In Example 60.4, **`dist`** returns a number between 1 and 100.

Please note that the values 1 and 100 can be returned by **`dist`**. The lower and upper limits of distributions are inclusive.

There are many distributions in Boost.Random besides `boost::random::bernoulli_distribution` and `boost::random::uniform_int_distribution`. For example, there are distributions like `boost::random::normal_distribution` and `boost::random::chi_squared_distribution`, which are used in statistics.

Chapter 61. Boost.NumericConversion

The library Boost.NumericConversion can be used to convert numbers of one numeric type to a different numeric type. In C++, such a conversion can also take place implicitly, as shown in Example 61.1.

Example 61.1. Implicit conversion from `int` to `short`

```
#include <iostream>

int main()
{
  int i = 0x10000;
  short s = i;
  std::cout << s << '\n';
}
```

Example 61.1 will compile cleanly because the type conversion from `int` to `short` takes place automatically. However, even though the program will run, the result of the conversion depends on the compiler used. The number `0x10000` in the variable **`i`** is too big to be stored in a variable of type `short`. According to the standard, the result of this operation is implementation specific. Compiled with Visual C++ 2013, the program displays `0`, which clearly differs from the value in **`i`**.

To avoid these kind of problems, you can use the cast operator `boost::numeric_cast` (see Example 61.2).

Example 61.2. Overflow detection with `boost::numeric_cast`

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

int main()
{
  try
  {
    int i = 0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << '\n';
  }
  catch (boost::numeric::bad_numeric_cast &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

`boost::numeric_cast` is used exactly like the existing C++ cast operators. The correct header file must be included; in this case, the header file `boost/numeric/conversion/cast.hpp`.

`boost::numeric_cast` does the same conversion as C++, but it verifies whether the conversion can take place without changing the value being converted. In Example 61.2, this verification fails, and an exception of type `boost::numeric::bad_numeric_cast` is thrown because `0x10000` is too big to be placed in a variable of type `short`.

Strictly speaking, an exception of type `boost::numeric::positive_overflow` will be thrown. This type specifies an overflow – in this case for positive numbers. There is also `boost::numeric::negative_overflow`, which specifies an overflow for negative numbers (see Example 61.3).

Example 61.3. Overflow detection for negative numbers

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

int main()
{
  try
  {
    int i = -0x10000;
    short s = boost::numeric_cast<short>(i);
    std::cout << s << '\n';
  }
  catch (boost::numeric::negative_overflow &e)
  {
    std::cerr << e.what() << '\n';
  }
}
```

Boost.NumericConversion defines additional exception types, all derived from `boost::numeric::bad_numeric_cast`. Because `boost::numeric::bad_numeric_cast` is derived from `std::bad_cast`, a `catch` handler can also catch exceptions of this type.

Part XV. Application Libraries

Application libraries refers to libraries that are typically used exclusively in the development of stand-alone applications and not in the development of libraries.

- Boost.Log is a logging library.
- Boost.ProgramOptions is a library to define and parse command line options.
- Boost.Serialization lets you serialize objects to, for example, save them to and load them from files.
- Boost.Uuid supports working with UUIDs.

Table of Contents

62. Boost.Log
63. Boost.ProgramOptions
64. Boost.Serialization
65. Boost.Uuid

Chapter 62. Boost.Log

Boost.Log is the logging library in Boost. It supports numerous back-ends to log data in various formats. Back-ends are accessed through front-ends that bundle services and forward log entries in different ways. For example, there is a front-end that uses a thread to forward log entries asynchronously. Front-ends can have filters to ignore certain log entries. And they define how log entries are formatted as strings. All these functions are extensible, which makes Boost.Log a powerful library.

Example 62.1. Back-end, front-end, core, and logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  sources::logger lg;

  BOOST_LOG(lg) << "note";
  sink->flush();
}
```

Example 62.1 introduces the essential components of Boost.Log. Boost.Log gives you access to back-ends, front-ends, the core, and loggers:

- Back-ends decide where data is written. `boost::log::sinks::text_ostream_backend` is initialized with a stream of type `std::ostream` and writes log entries to it.

- Front-ends are the connection between the core and a back-end. They implement various functions that don't need to be implemented by each individual back-end. For example, filters can be added to a front-end to choose which log entries get forwarded to the back-end and which don't.

  Example 62.1 uses the front-end `boost::log::sinks::asynchronous_sink`. You must use a front-end even if you don't use filters. `boost::log::sinks::asynchronous_sink` uses a thread that forwards log entries to a back-end asynchronously. This can improve the performance but defers write operations.

- The core is the central component that all log entries are routed through. It is implemented as a singleton. To get a pointer to the core, call `boost::log::core::get()`.

Front-ends must be added to the core to receive log entries. Whether log entries are forwarded to front-ends depends on the filter in the core. Filters can be registered either in front-ends or in the core. Filters registered in the core are global, and filters registered in front-ends are local. If a log entry is filtered out by the core, it isn't forwarded to any front-end. If it is filtered by a front-end, it can still be processed by other front-ends and forwarded to their back-ends.

- The logger is the component in Boost.Log you will use most often. While you access back-ends, front-ends, and the core only when you initialize the logging library, you use a logger every time you write a log entry. The logger forwards the entry to the core.

  The logger in Example 62.1 is of the type `boost::log::sources::logger`. This is the simplest logger. When you want to write a log entry, use the macro `BOOST_LOG` and pass the logger as a parameter. The log entry is created by writing data into the macro as if it is a stream of type `std::ostream`.

Back-end, front-end, core, and logger work together. `boost::log::sinks::asynchronous_sink`, a front-end, is a template that receives the back-end `boost::log::sinks::text_ostream_backend` as a parameter. Afterwards, the front-end is instantiated with `boost::shared_ptr`. The smart pointer is required to register the front-end in the core: the call to `boost::log::core::add_sink()` expects a `boost::shared_ptr`.

Because the back-end is a template parameter of the front-end, it can only be configured after the front-end has been instantiated. The back-end determines how this is done. The member function `add_stream()` is provided by the back-end `boost::log::sinks::text_ostream_backend` to add streams. You can add more than one stream to `boost::log::sinks::text_ostream_backend`. Other back-ends provide different member functions for configuration. Consult the documentation for details.

To get access to a back-end, all front-ends provide the member function `locked_backend()`. This member function is called `locked_backend()` because it returns a pointer that provides synchronized access to the back-end as long as the pointer exists. You can access a back-end through pointers returned by `locked_backend()` from multiple threads without having to synchronize access yourself.

You can instantiate a logger like `boost::log::sources::logger` with the default constructor. The logger automatically calls `boost::log::core::get()` to forward log entries to the core.

You can access loggers without macros. Loggers are objects with member functions you can call. However, macros like `BOOST_LOG` make it easier to write log entries. Without macros it wouldn't be possible to write a log entry in one line of code.

Example 62.1 calls `boost::log::sinks::asynchronous_sink::flush()` at the end of `main()`. This call is required because the front-end is asynchronous and uses a thread to forward log entries. The call makes sure that all buffered log entries are passed to the back-end and are written. Without the call to `flush()`, the example could terminate without displaying `note`.

Example 62.2. `boost::sources::severity_logger` with a filter

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

bool only_warnings(const attribute_value_set &set)
{
  return set["Severity"].extract<int>() > 0;
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(&only_warnings);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG(lg) << "note";
  BOOST_LOG_SEV(lg, 0) << "another note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Example 62.2 is based on Example 62.1, but it replaces `boost::sources::logger` with the logger `boost::sources::severity_logger`. This logger adds an attribute for a log level to every log entry. You can use the macro `BOOST_LOG_SEV` to set the log level.

The type of the log level depends on a template parameter passed to `boost::sources::severity_logger`. Example 62.2 uses `int`. That's why numbers like 0 and 1 are passed to `BOOST_LOG_SEV`. If `BOOST_LOG` is used, the log level is set to 0.

Example 62.2 also calls `set_filter()` to register a filter at the front-end. The filter function is called for every log entry. If the function returns `true`, the log entry is forwarded to the back-end. Example 62.2 defines the function `only_warnings()` with a return value of type `bool`.

`only_warnings()` expects a parameter of type `boost::log::attribute_value_set`. This type represents log entries while they are being passed around in the logging framework. `boost::log::record` is another type for log entries that is like a wrapper for `boost::log::attribute_value_set`. This type provides the member function `attribute_values()`, which retrieves a reference to the `boost::log::attribute_value_set`. Filter functions receive a `boost::log::attribute_value_set` directly and no `boost::log::record`. `boost::log::attribute_value_set` stores key/value pairs. Think of it as a `std::unordered_map`.

Log entries consist of attributes. Attributes have a name and a value. You can create attributes yourself. They can also be created automatically – for example by loggers. In fact, that's why Boost.Log provides multiple loggers. `boost::log::sources::severity_logger` adds an attribute called Severity to every log entry. This attribute stores the log level. That way a filter can check whether the log level of a log entry is greater than 0.

`boost::log::attribute_value_set` provides several member functions to access attributes. The member functions are similar to the ones provided by `std::unordered_map`. For example, `boost::log::attribute_value_set` overloads the operator `operator[]`. This operator returns the value of an attribute whose name is passed as a parameter. If the attribute doesn't exist, it is created.

The type of attribute names is `boost::log::attribute_name`. This class provides a constructor that accepts a string, so you can pass a string directly to `operator[]`, as in Example 62.2.

The type of attribute values is `boost::log::attribute_value`. This class provides member functions to receive the value in the attribute's original type. Because the log level is an `int` value, `int` is passed as a template parameter to `extract()`.

`boost::log::attribute_value` also defines the member functions `extract_or_default()` and `extract_or_throw()`. `extract()` returns a value created with the default constructor if a type conversion fails – for example 0 in case of an `int`. `extract_or_default()` returns a default value which is passed as another parameter to that member function. `extract_or_throw()` throws an exception of type `boost::log::runtime_error` in the event of an error.

For type-safe conversions, Boost.Log provides the visitor function `boost::log::visit()`, which you can use instead of `extract()`.

Example 62.2 displays `warning`. This log entry has a log level greater than 0 and thus isn't filtered.

Example 62.3. Changing the format of a log entry with `set_formatter()`

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

void severity_and_message(const record_view &view, formatting_ostream &os)
{
  os << view.attribute_values()["Severity"].extract<int>() << ": " <<
    view.attribute_values()["Message"].extract<std::string>();
}

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();
```

```
  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_formatter(&severity_and_message);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  sink->flush();
}
```

Example 62.3 is based on Example 62.2. This time the log level is displayed.

Front-ends provide the member function `set_formatter()`, which can be passed a format function. If a log entry isn't filtered by a front-end, it is forwarded to the format function. This function formats the log entry as a string that is then passed from the front-end to the back-end. If you don't call `set_formatter()`, by default the back-end only receives what is on the right side of a macro like `BOOST_LOG`.

Example 62.3 passes the function `severity_and_message()` to `set_formatter()`. `severity_and_message()` expects parameters of type `boost::log::record_view` and `boost::log::formatting_ostream`. `boost::log::record_view` is a view on a log entry. It's similar to `boost::log::record`. However, `boost::log::record_view` is an immutable log entry.

`boost::log::record_view` provides the member function `attribute_values()`, which returns a constant reference to `boost::log::attribute_value_set`. `boost::log::formatting_ostream` is the stream used to create the string that is passed to the back-end.

`severity_and_message()` accesses the attributes Severity and Message. `extract()` is called to get the attribute values, which are then written to the stream. Severity returns the log level as an `int` value. Message provides access to what is on the right side of a macro like `BOOST_LOG`. Consult the documentation for a complete list of available attribute names.

Example 62.3 uses no filter. The example writes two log entries: `0: note` and `1: warning`.

Example 62.4. Filtering log entries and formatting them with lambda functions

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();
```

```
  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::attr<int>("Severity") > 0);
  sink->set_formatter(expressions::stream <<
    expressions::attr<int>("Severity") << ": " << expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Example 62.4 uses both a filter and a format function. This time the functions are implemented as lambda functions – not as C++11 lambda functions but as Boost.Phoenix lambda functions.

Boost.Log provides helpers for lambda functions in the namespace `boost::log::expressions`. For example, **`boost::log::expressions::stream`** represents the stream. **`boost::log::expressions::smessage`** provides access to everything on the right side of a macro like `BOOST_LOG`. You can use `boost::log::expressions::attr()` to access any attribute. Instead of **`smessage`** Example 62.4 could use `attr<std::string>("Message")`.

Example 62.4 displays `1: warning` and `2: error`.

Example 62.5. Defining keywords for attributes

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << severity << ": " <<
    expressions::smessage);

  core::get()->add_sink(sink);

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  BOOST_LOG_SEV(lg, 2) << "error";
  sink->flush();
}
```

Boost.Log supports user-defined keywords. You can use the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define keywords to access attributes without having to repeatedly pass attribute names as strings to `boost::log::expressions::attr()`.

Example 62.5 uses the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define a keyword **`severity`**. The macro expects three parameters: the name of the keyword, the attribute name as a string, and the type of the attribute. The new keyword can be used in filter and format lambda functions. This means you are not restricted to using keywords, such as **`boost::log::expressions::smessage`**, that are provided by Boost.Log – you can also define new keywords.

In all of the examples so far, the attributes used are the ones defined in Boost.Log. Example 62.6 shows how to create user-defined attributes.

Example 62.6. Defining attributes

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(severity > 0);
  sink->set_formatter(expressions::stream << counter << " - " << severity <<
    ": " << expressions::smessage << " (" << timestamp << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

You create a global attribute by calling `add_global_attribute()` on the core. The attribute is global because it is added to every log entry automatically.

`add_global_attribute()` expects two parameters: the name and the type of the new attribute. The name is passed as a string. For the type you use a class from the namespace `boost::log::attributes`, which provides classes to define different attributes. Example 62.6 uses `boost::log::attributes::counter` to define the attribute LineCounter, which adds a line number to every log entry. This attribute will number log entries starting at 1.

`add_global_attribute()` is not a function template. `boost::log::attributes::counter` isn't passed as a template parameter. The attribute type must be instantiated and passed as an object.

Example 62.6 uses a second attribute called Timestamp. This is a scoped attribute that is created with `BOOST_LOG_SCOPED_LOGGER_ATTR`. This macro adds an attribute to a logger. The first parameter is the logger, the second is the attribute name, and the third is the attribute object. The type of the attribute object is `boost::log::attribute::local_clock`. The attribute is set to the current time for each log entry.

The attribute Timestamp is added to the log entry "error" only. Timestamp exists only in the scope where `BOOST_LOG_SCOPED_LOGGER_ATTR` is used. When the scope ends, the attribute is removed. `BOOST_LOG_SCOPED_LOGGER_ATTR` is similar to a call to `add_attribute()` and `remove_attribute()`.

As in Example 62.5, Example 62.6 uses the macro `BOOST_LOG_ATTRIBUTE_KEYWORD` to define keywords for the new attributes. The format function accesses the keywords to write the line number and current time. The value of **`timestamp`** will be an empty string for those log entries where the attribute Timestamp is undefined.

Example 62.7. Helper functions for filters and formats

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/support/date_time.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <iomanip>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(counter, "LineCounter", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(timestamp, "Timestamp",
  boost::posix_time::ptime)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
```

```
  sink->locked_backend()->add_stream(stream);
  sink->set_filter(expressions::is_in_range(severity, 1, 3));
  sink->set_formatter(expressions::stream << std::setw(5) << counter <<
    " - " << severity << ": " << expressions::smessage << " (" <<
    expressions::format_date_time(timestamp, "%H:%M:%S") << ")");

  core::get()->add_sink(sink);
  core::get()->add_global_attribute("LineCounter",
    attributes::counter<int>{});

  sources::severity_logger<int> lg;

  BOOST_LOG_SEV(lg, 0) << "note";
  BOOST_LOG_SEV(lg, 1) << "warning";
  {
    BOOST_LOG_SCOPED_LOGGER_ATTR(lg, "Timestamp", attributes::local_clock{})
    BOOST_LOG_SEV(lg, 2) << "error";
  }
  BOOST_LOG_SEV(lg, 2) << "another error";
  sink->flush();
}
```

Boost.Log provides numerous helper functions for filters and formats. Example 62.7 calls the helper `boost::log::expressions::is_in_range()` to filter log entries whose log level is outside a range. `boost::log::expressions::is_in_range()` expects the attribute as its first parameter and lower and upper bounds as its second and third parameters. As with iterators, the upper bound is exclusive and doesn't belong to the range.

`boost::log::expressions::format_date_time()` is called in the format function. It is used to format a timepoint. Example 62.7 uses `boost::log::expressions::format_date_time()` to write the time without a date. You can also use manipulators from the standard library in format functions. Example 62.7 uses `std::setw()` to set the width for the counter.

Example 62.8. Several loggers, front-ends, and back-ends

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/severity_logger.hpp>
#include <boost/log/sources/channel_logger.hpp>
#include <boost/log/expressions.hpp>
#include <boost/log/attributes.hpp>
#include <boost/log/utility/string_literal.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <string>

using namespace boost::log;

BOOST_LOG_ATTRIBUTE_KEYWORD(severity, "Severity", int)
BOOST_LOG_ATTRIBUTE_KEYWORD(channel, "Channel", std::string)

int main()
{
  typedef sinks::asynchronous_sink<sinks::text_ostream_backend>
    ostream_sink;
  boost::shared_ptr<ostream_sink> ostream =
    boost::make_shared<ostream_sink>();
  boost::shared_ptr<std::ostream> clog{&std::clog,
    boost::empty_deleter{}};
  ostream->locked_backend()->add_stream(clog);
  core::get()->add_sink(ostream);
```

```
  typedef sinks::synchronous_sink<sinks::text_multifile_backend>
    multifile_sink;
  boost::shared_ptr<multifile_sink> multifile =
    boost::make_shared<multifile_sink>();
  multifile->locked_backend()->set_file_name_composer(
    sinks::file::as_file_name_composer(expressions::stream <<
    channel.or_default<std::string>("None") << "-" <<
    severity.or_default(0) << ".log"));
  core::get()->add_sink(multifile);

  sources::severity_logger<int> severity_lg;
  sources::channel_logger<> channel_lg{keywords::channel = "Main"};

  BOOST_LOG_SEV(severity_lg, 1) << "severity message";
  BOOST_LOG(channel_lg) << "channel message";
  ostream->flush();
}
```

Example 62.8 uses several loggers, front-ends, and back-ends. In addition to using the classes `boost::log::sinks::asynchronous_sink`, `boost::log::sinks::text_ostream_backend` and `boost::log::sources::severity_logger`, the example also uses the front-end `boost::log::sinks::synchronous_sink`, the back-end `boost::log::sinks::text_multifile_backend`, and the logger `boost::log::sources::channel_logger`.

The front-end `boost::log::sinks::synchronous_sink` provides synchronous access to a back-end, which lets you use a back-end in a multithreaded application even if the back-end isn't thread safe.

The difference between the two front-ends `boost::log::sinks::asynchronous_sink` and `boost::log::sinks::synchronous_sink` is that the latter isn't based on a thread. Log entries are passed to the back-end in the same thread.

Example 62.8 uses the front-end `boost::log::sinks::synchronous_sink` with the back-end `boost::log::sinks::text_multifile_backend`. This back-end writes log entries to one or more files. File names are created according to a rule passed by `set_file_name_composer()` to the back-end. If you use the free-standing function `boost::log::sinks::file::as_file_name_composer()`, as in the example, the rule can be created as a lambda function with the same building blocks used for format functions. However, the attributes aren't used to create the string that is written to a back-end. Instead, the string will be the name of the file that log entries will be written to.

Example 62.8 uses the keywords **`channel`** and **`severity`**, which are defined with the macro `BOOST_LOG_ATTRIBUTE_KEYWORD`. They refer to the attributes Channel and Severity. The member function `or_default()` is called on the keywords to pass a default value if an attribute isn't set. If a log entry is written and Channel and Severity are not set, the entry is written to the file `None-0.log`. If a log entry is written with the log level 1, it is stored in the file `None-1.log`. If the log level is 1 and the channel is called Main, the log entry is saved in the file `Main-1.log`.

The attribute Channel is defined by the logger `boost::log::sources::channel_logger`. The constructor expects a channel name. The name can't be passed directly as a string. Instead, it must be passed as a named parameter. That's why the example uses `keywords::channel =`

"Main" even though boost::log::sources::channel_logger doesn't accept any other parameters.

Please note that the named parameter boost::log::keywords::channel has nothing to do with the keywords you create with the macro BOOST_LOG_ATTRIBUTE_KEYWORD.

boost::log::sources::channel_logger identifies log entries from different components of a program. Components can use their own objects of type boost::log::sources::channel_logger, giving them unique names. If components only access their own loggers, it's clear which component a particular log entry came from.

Example 62.9. Handling exceptions centrally

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/log/utility/exception_handler.hpp>
#include <boost/log/exceptions.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

struct handler
{
  void operator()(const runtime_error &ex) const
  {
    std::cerr << "boost::log::runtime_error: " << ex.what() << '\n';
  }

  void operator()(const std::exception &ex) const
  {
    std::cerr << "std::exception: " << ex.what() << '\n';
  }
};

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);
  core::get()->set_exception_handler(
    make_exception_handler<runtime_error, std::exception>(handler{}));

  sources::logger lg;

  BOOST_LOG(lg) << "note";
}
```

Boost.Log provides the option to handle exceptions in the logging framework centrally. This means you don't need to wrap every BOOST_LOG in a try block to handle exceptions in catch.

Example 62.9 calls the member function set_exception_handler(). The core provides this member function to register a handler. All exceptions in the logging framework will be passed to

that handler. The handler is implemented as a function object. It has to overload `operator()` for every exception type expected. An instance of that function object is passed to `set_exception_handler()` through the function template `boost::log::make_exception_handler()`. All exception types you want to handle must be passed as template parameters to `boost::log::make_exception_handler()`.

The function `boost::log::make_exception_suppressor()` let's you discard all exceptions in the logging framework. You call this function instead of `boost::log::make_exception_handler()`.

Example 62.10. A macro to define a global logger

```
#include <boost/log/common.hpp>
#include <boost/log/sinks.hpp>
#include <boost/log/sources/logger.hpp>
#include <boost/utility/empty_deleter.hpp>
#include <boost/shared_ptr.hpp>
#include <iostream>
#include <exception>

using namespace boost::log;

BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT(lg, sources::wlogger_mt)

int main()
{
  typedef sinks::synchronous_sink<sinks::text_ostream_backend> text_sink;
  boost::shared_ptr<text_sink> sink = boost::make_shared<text_sink>();

  boost::shared_ptr<std::ostream> stream{&std::clog,
    boost::empty_deleter{}};
  sink->locked_backend()->add_stream(stream);

  core::get()->add_sink(sink);

  BOOST_LOG(lg::get()) << L"note";
}
```

All of the examples in this chapter use local loggers. If you want to define a global logger, use the macro `BOOST_LOG_INLINE_GLOBAL_LOGGER_DEFAULT` as in Example 62.10. You pass the name of the logger as the first parameter and the type as the second. You don't access the logger through its name. Instead, you call `get()`, which returns a pointer to a singleton.

Boost.Log provides additional macros such as `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS`. They let you initialize global loggers. `BOOST_LOG_INLINE_GLOBAL_LOGGER_CTOR_ARGS` lets you pass parameters to the constructor of a global logger. All of these macros guarantee that global loggers will be correctly initialized.

Boost.Log provides many more functions that are worth a look. For example, you can configure the logging framework through a container with key/value pairs as strings. Then, you don't need to instantiate classes and call member functions. For example, a key Destination can be set to Console, which will automatically make the logging framework use the back-end `boost::log::sinks::text_ostream_backend`. The back-end can be configured through additional key/value pairs. Because the container can also be serialized in an INI-file, it is possible to store the configuration in a text file and initialize the logging framework with that file.

Chapter 63. Boost.ProgramOptions

Boost.ProgramOptions is a library that makes it easy to parse command-line options, for example, for console applications. If you develop applications with a graphical user interface, command-line options are usually not important.

To parse command-line options with Boost.ProgramOptions, the following three steps are required:

1. Define command-line options. You give them names and specify which ones can be set to a value. If a command-line option is parsed as a key/value pair, you also set the type of the value – for example, whether it is a string or a number.

2. Use a parser to evaluate the command line. You get the command line from the two parameters of `main()`, which are usually called **`argc`** and **`argv`**.

3. Store the command-line options evaluated by the parser. Boost.ProgramOptions offers a class derived from `std::map` that saves command-line options as name/value pairs. Afterwards, you can check which options have been stored and what their values are.

Example 63.1 shows the basic approach for parsing command-line options with Boost.ProgramOptions.

Example 63.1. Basic approach with Boost.ProgramOptions

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

using namespace boost::program_options;

void on_age(int age)
{
  std::cout << "On age: " << age << '\n';
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->default_value(3.14f), "Pi")
      ("age", value<int>()->notifier(on_age), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, desc), vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << vm["age"].as<int>() << '\n';
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
```

```
    std::cerr << ex.what() << '\n';
  }
}
```

To use Boost.ProgramOptions, include the header file `boost/program_options.hpp`. You can access all classes and functions from this library in the namespace `boost::program_options`.

Use the class `boost::program_options::options_description` to describe command-line options. An object of this type can be written to a stream such as **`std::cout`** to display an overview of available command-line options. The string passed to the constructor gives the overview a name that acts as a title for the command-line options.

`boost::program_options::options_description` defines a member function `add()` that expects a parameter of type `boost::program_options::option_description`. You call this function to describe each command-line option. Instead of calling this function for every command-line option, Example 63.1 calls the member function `add_options()`, which makes that task easier.

`add_options()` returns a proxy object representing an object of type `boost::program_options::options_description`. The type of the proxy object doesn't matter. It's more interesting that the proxy object simplifies defining many command-line options. It uses the overloaded operator `operator()`, which you can call to pass the required data to define a command-line option. This operator returns a reference to the same proxy object, which allows you to call `operator()` multiple times.

Example 63.1 defines three command-line options with the help of the proxy object. The first command-line option is `--help`. The description of this option is set to "Help screen". The option is a switch, not a name/value pair. You set `--help` on the command line or omit it. It's not possible to set `--help` to a value.

Please note that the first string passed to `operator()` is "help,h". You can specify short names for command-line options. A short name must consist of just one letter and is set after a comma. Now the help can be displayed with either `--help` or `-h`.

Besides `--help`, two more command-line options are defined: `--pi` and `--age`. These options aren't switches, they're name/value pairs. Both `--pi` and `--age` expect to be set to a value.

You pass a pointer to an object of type `boost::program_options::value_semantic` as the second parameter to `operator()` to define an option as a name/value pair. You don't need to access `boost::program_options::value_semantic` directly. You can use the helper function `boost::program_options::value()`, which creates an object of type `boost::program_options::value_semantic`. `boost::program_options::value()` returns the object's address, which you then can pass to the proxy object using `operator()`.

`boost::program_options::value()` is a function template that takes the type of the command-line option value as a template parameter. Thus, the command-line option `--age` expects an integer and `--pi` expects a floating point number.

The object returned from `boost::program_options::value()` provides some useful member functions. For example, you can call `default_value()` to provide a default value. Example 63.1 sets `--pi` to 3.14 if that option isn't used on the command line.

`notifier()` links a function to a command-line option's value. That function is then called with the value of the command-line option. In Example 63.1, the function `on_age()` is linked to `--age`. If the command-line option `--age` is used to set an age, the age is passed to `on_age()` which writes it to standard output.

Processing values with functions like `on_age()` is optional. You don't have to use `notifier()` because it's possible to access values in other ways.

After all command-line options have been defined, you use a parser. In Example 63.1, the helper function `boost::program_options::parse_command_line()` is called to parse the command line. This function takes **`argc`** and **`argv`**, which define the command line, and **`desc`**, which contains the option descriptions. `boost::program_options::parse_command_line()` returns the parsed options in an object of type `boost::program_options::parsed_options`. You usually don't access this object directly. Instead you pass it to `boost::program_options::store()`, which stores the parsed options in a container.

Example 63.1 passes **`vm`** as a second parameter to `boost::program_options::store()`. **`vm`** is an object of type `boost::program_options::variables_map`. This class is derived from the class `std::map<std::string, boost::program_options::variable_value>` and, thus, provides the same member functions as `std::map`. For example, you can call `count()` to check whether a certain command-line option has been used and is stored in the container.

In Example 63.1, before **`vm`** is accessed and `count()` is called, `boost::program_options::notify()` is called. This function triggers functions, such as `on_age()`, that are linked to a value using `notifier()`. Without `boost::program_options::notify()`, `on_age()` would not be called.

`vm` lets you check whether a certain command-line option exists, and it also lets you access the value the command-line option is set to. The value's type is `boost::program_options::variable_value`, a class that uses `boost::any` internally. You can get the object of type `boost::any` from the member function `value()`.

Example 63.1 calls `as()`, not `value()`. This member function converts the value of a command-line option to the type passed as a template parameter. `as()` uses `boost::any_cast()` for the type conversion.

Be sure the type you pass to `as()` matches the type of the command-line option. For example, Example 63.1 expects the command-line option `--age` to be set to a number of type `int`, so `int` must be passed as a template parameter to `as()`.

You can start Example 63.1 in many ways. Here is one example:

test

In this case `Pi: 3.14` is displayed. Because `--pi` isn't set on the command line, the default value is displayed.

This example sets a value using `--pi`:

test --pi 3.1415

The program now displays `Pi: 3.1415`.

This example also passes an age:

test --pi 3.1415 --age 29

The output is now `On age: 29` and `Age: 29`. The first line is written when `boost::program_options::notify()` is called; this triggers the execution of `on_age()`. There is no output for `--pi` because the program uses `else if` statements that only display the value set with `--pi` if `--age` is not set.

This example shows the help:

test -h

You get a complete overview on all command-line options:

```
Options: -h [ --help ]          Help screen --pi arg (=3.1400001) Pi --age arg
```

As you can see, the help can be shown in two different ways because a short name for that command-line option was defined. For `--pi` the default value is displayed. The command-line options and their descriptions are formatted automatically. You only need to write the object of type `boost::program_options::options_description` to standard output as in Example 63.1.

Now, start the example like this:

test --age

The output is the following:

```
the required argument for option '--age' is missing.
```

Because `--age` isn't set, the parser used in `boost::program_options::parse_command_line()` throws an exception of type `boost::program_options::error`. The exception is caught, and an error message is written to standard output.

`boost::program_options::error` is derived from `std::logic_error`. Boost.ProgramOptions defines additional exceptions, which are all derived from `boost::program_options::error`. One of those exceptions is `boost::program_options::invalid_syntax`, which is the exact exception thrown in Example 63.1 if you don't supply a value for `--age`.

Example 63.2 introduces more configuration settings available with Boost.ProgramOptions.

Example 63.2. Special configuration settings with Boost.ProgramOptions

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

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(), std::ostream_iterator<std::string>{
    std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    int age;

    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("pi", value<float>()->implicit_value(3.14f), "Pi")
      ("age", value<int>(&age), "Age")
      ("phone", value<std::vector<std::string>>()->multitoken()->
        zero_tokens()->composing(), "Phone")
      ("unreg", "Unrecognized options");

    command_line_parser parser{argc, argv};
    parser.options(desc).allow_unregistered().style(
      command_line_style::default_style |
      command_line_style::allow_slash_for_short);
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("age"))
      std::cout << "Age: " << age << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
    else if (vm.count("unreg"))
      to_cout(collect_unrecognized(parsed_options.options,
        exclude_positional));
    else if (vm.count("pi"))
      std::cout << "Pi: " << vm["pi"].as<float>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.2 parses command-line options like the previous example does. However, there are some notable differences. For example, `implicit_value()` is called, rather than `default_value()`, when defining the `--pi` command-line option. This means that pi isn't set to

3.14 by default. `--pi` must be set on the command line for pi to be available. However, you don't need to supply a value to the `--pi` command-line option if you use `implicit_value()`. It's sufficient to pass `--pi` without setting a value. In that case, pi is set to 3.14 implicitly.

For the command-line option `--age`, a pointer to the variable **age** is passed to `boost::program_options::value()`. This stores the value of a command-line option in a variable. Of course, the value is still available in the container **vm**.

Please note that a value is only stored in **age** if `boost::program_options::notify()` is called. Even though `notifier()` isn't used in this example, `boost::program_options::notify()` still must be used. To avoid problems, it's a good idea to always call `boost::program_options::notify()` after parsed command-line options have been stored with `boost::program_options::store()`.

Example 63.2 supports a new command-line option `--phone` to pass a phone number to the program. In fact, you can pass multiple phone numbers on the command line. For example, the following command line starts the program with the phone numbers 123 and 456:

test --phone 123 456

Example 63.2 supports multiple phone numbers because `multitoken()` is called on this command-line option's value. And, since `zero_tokens()` is called, `--phone` can also be used without passing a phone number.

You can also pass multiple phone numbers by repeating the `--phone` option, as shown in the following command line:

test --phone 123 --phone 456

In this case, both phone numbers, 123 and 456, are parsed. The call to `composing()` makes it possible to use a command-line option multiple times – the values are composed.

The value of the argument to `--phone` is of type `std::vector<std::string>`. You need to use a container to store multiple phone numbers.

Example 63.2 defines another command-line option, `--unreg`. This is a switch that can't be set to a value. It is used later in the example to decide whether command-line options that aren't defined in **desc** should be displayed.

While Example 63.1 calls the function `boost::program_options::parse_command_line()` to parse command-line options, Example 63.2 uses a parser of type `boost::program_options::command_line_parser`. **argc** and **argv** are passed to the constructor.

`boost::program_options::command_line_parser` provides several member functions. You must call `options()` to pass the definition of command-line options to the parser.

Like other member functions, `options()` returns a reference to the same parser. That way, member functions can be easily called one after another. Example 63.2 calls

`allow_unregistered()` after `options()` to tell the parser not to throw an exception if unknown command-line options are detected. Finally, `style()` is called to tell the parser that short names can be used with a slash. Thus, the short name for the `--help` option can be either `-h` or `/h`.

Please note that `boost::program_options::parse_command_line()` supports a fourth parameter, which is forwarded to `style()`. If you want to use an option like **`boost::program_options::command_line_style::allow_slash_for_short`**, you can still use the function `boost::program_options::parse_command_line()`.

After the configuration has been set, call `run()` on the parser. This member function returns the parsed command-line options in an object of type `boost::program_options::parsed_options`, which you can pass to `boost::program_options::store()` to store the options in **`vm`**.

Later in the code, Example 63.2 accesses **`vm`** again to evaluate command-line options. Only the call to `boost::program_options::collect_unrecognized()` is new. This function is called for the command-line option `--unreg`. The function expects an object of type `boost::program_options::parsed_options`, which is returned by `run()`. It returns all unknown command-line options in a `std::vector<std::string>`. For example, if you start the program with **test --unreg --abc**, `--abc` will be written to standard output.

When **`boost::program_options::exclude_positional`** is passed as the second parameter to `boost::program_options::collect_unrecognized()`, positional options are ignored. For Example 63.2, this doesn't matter because no positional options are defined. However, `boost::program_options::collect_unrecognized()` requires this parameter.

Example 63.3 illustrates positional options.

Example 63.3. Positional options with Boost.ProgramOptions

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

using namespace boost::program_options;

void to_cout(const std::vector<std::string> &v)
{
  std::copy(v.begin(), v.end(),
    std::ostream_iterator<std::string>{std::cout, "\n"});
}

int main(int argc, const char *argv[])
{
  try
  {
    options_description desc{"Options"};
    desc.add_options()
      ("help,h", "Help screen")
      ("phone", value<std::vector<std::string>>()->
        multitoken()->zero_tokens()->composing(), "Phone");

    positional_options_description pos_desc;
    pos_desc.add("phone", -1);
```

```
    command_line_parser parser{argc, argv};
    parser.options(desc).positional(pos_desc).allow_unregistered();
    parsed_options parsed_options = parser.run();

    variables_map vm;
    store(parsed_options, vm);
    notify(vm);

    if (vm.count("help"))
      std::cout << desc << '\n';
    else if (vm.count("phone"))
      to_cout(vm["phone"].as<std::vector<std::string>>());
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.3 defines `--phone` as a positional option using the class `boost::program_options::positional_options_description`. This class provides the member function `add()`, which expects the name of the command-line option and a position to be passed. The example passes “phone” and -1.

With positional options, values can be set on the command line without using command-line options. You can start Example 63.3 like this:

test 123 456

Even though `--phone` isn’t used, 123 and 456 are recognized as phone numbers.

Calling `add()` on an object of type `boost::program_options::positional_options_description` assigns values on the command line to command-line options using position numbers. When Example 63.3 is called using the command line **test 123 456**, 123 has the position number 0 and 456 has the position number 1. Example 63.3 passes -1 to `add()`, which assigns all of the values – 123 and 456 – to `--phone`. If you changed Example 63.3 to pass the value 0 to `add()`, only 123 would be recognized as a phone number. And if 1 was passed to `add()`, only 456 would be recognized.

`pos_desc` is passed with `positional()` to the parser. That’s how the parser knows which command-line options are positional.

Please note that you have to make sure that positional options are defined. In Example 63.3, for example, “phone” could only be passed to `add()` because a definition for `--phone` already existed in **`desc`**.

In all previous examples, Boost.ProgramOptions was used to parse command-line options. However, the library supports loading configuration options from a file, too. This can be useful if the same command-line options have to be set repeatedly.

Example 63.4. Loading options from a configuration file

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

```
#include <fstream>
#include <iostream>

using namespace boost::program_options;

int main(int argc, const char *argv[])
{
  try
  {
    options_description generalOptions{"General"};
    generalOptions.add_options()
      ("help,h", "Help screen")
      ("config", value<std::string>(), "Config file");

    options_description fileOptions{"File"};
    fileOptions.add_options()
      ("age", value<int>(), "Age");

    variables_map vm;
    store(parse_command_line(argc, argv, generalOptions), vm);
    if (vm.count("config"))
    {
      std::ifstream ifs{vm["config"].as<std::string>().c_str()};
      if (ifs)
        store(parse_config_file(ifs, fileOptions), vm);
    }
    notify(vm);

    if (vm.count("help"))
      std::cout << generalOptions << '\n';
    else if (vm.count("age"))
      std::cout << "Your age is: " << vm["age"].as<int>() << '\n';
  }
  catch (const error &ex)
  {
    std::cerr << ex.what() << '\n';
  }
}
```

Example 63.4 uses two objects of type `boost::program_options::options_description`. **`generalOptions`** defines options that must be set on the command line. **`fileOptions`** defines options that can be loaded from a configuration file.

It's not mandatory to define options with two different objects of type `boost::program_options::options_description`. You can use just one if the set of options is the same for both command line and file. In Example 63.4, separating options makes sense because you don't want to allow `--help` to be set in the configuration file. If that was allowed and the user put that option in the configuration file, the program would display the help screen every time.

Example 63.4 loads `--age` from a configuration file. You can pass the name of the configuration file as a command-line option. In this example, `--config` is defined in **`generalOptions`** for that reason.

After the command-line options have been parsed with `boost::program_options::parse_command_line()` and stored in **`vm`**, the example checks whether `--config` is set. If it is, the configuration file is opened with `std::ifstream`. The `std::ifstream` object is passed to the function `boost::program_options::parse_config_file()` along with **`fileOptions`**, which describes

the options. `boost::program_options::parse_config_file()` does the same thing as `boost::program_options::parse_command_line()` and returns parsed options in an object of type `boost::program_options::parsed_options`. This object is passed to `boost::program_options::store()` to store the parsed options in **`vm`**.

If you create a file called `config.txt`, put age=29 in that file, and execute the command line below, you will get the result shown.

test --config config.txt

The output is the following:

```
Your age is: 29
```

If you support the same options on the command line and in a configuration file, your program may parse the same option twice – once with `boost::program_options::parse_command_line()` and once with `boost::program_options::parse_config_file()`. The order of the function calls determines which value you will find in **`vm`**. Once a command-line option's value has been stored in **`vm`**, that value will not be overwritten. Whether the value is set by an option on the command line or in a configuration file depends only on the order in which you call the `store()` function.

Boost.ProgramOptions also defines the function `boost::program_options::parse_environment()`, which can be used to load options from environment variables. The class `boost::environment_iterator` lets you iterate over environment variables.

Chapter 64. Boost.Serialization

Table of Contents

Archive
Pointers and References
Serialization of Class Hierarchy Objects
Wrapper Functions for Optimization

The library Boost.Serialization makes it possible to convert objects in a C++ program to a sequence of bytes that can be saved and loaded to restore the objects. There are different data formats available to define the rules for generating sequences of bytes. All of the formats supported by Boost.Serialization are only intended for use with this library. For example, the XML format developed for Boost.Serialization should not be used to exchange data with programs that do not use Boost.Serialization. The only advantage of the XML format is that it can make debugging easier since C++ objects are saved in a readable format.

Note

As outlined in the release notes of version 1.55.0 of the Boost libraries, a missing include causes a compiler error with Visual C++ 2013. This bug has been fixed in Boost 1.56.0.

Chapter 65. Boost.Uuid

Boost.Uuid provides generators for *UUIDs*. UUIDs are universally unique identifiers that don't depend on a central coordinating instance. There is, for example, no database storing all generated UUIDs that can be checked to see whether a new UUID has been used.

UUIDs are used by distributed systems that have to uniquely identify components. For example, Microsoft uses UUIDs to identify interfaces in the COM world. For new interfaces developed for COM, unique identifiers can be easily assigned.

UUIDs are 128-bit numbers. Various methods exist to generate UUIDs. For example, a computer's network address can be used to generate a UUID. The generators provided by Boost.Uuid are based on a random number generator to avoid generating UUIDs that can be traced back to the computer generating them.

All classes and functions from Boost.Uuid are defined in the namespace `boost::uuids`. There is no master header file to get access to all of them.

Example 65.1. Generating random UUIDs with `boost::uuids::random_generator`

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();
  std::cout << id << '\n';
}
```

Example 65.1 generates a random UUID. It uses the class `boost::uuids::random_generator`, which is defined in `boost/uuid/uuid_generators.hpp`. This header file provides access to all generators provided by Boost.Uuid.

`boost::uuids::random_generator` is used like the generators from the C++11 standard library or from Boost.Random. This class overloads `operator()` to generate random UUIDs.

The type of a UUID is `boost::uuids::uuid`. `boost::uuids::uuid` is a *POD* – plain old data. You can't create objects of type `boost::uuids::uuid` without a generator. But then, it's a lean type that allocates exactly 128 bits. The class is defined in `boost/uuid/uuid.hpp`.

An object of type `boost::uuids::uuid` can be written to the standard output stream. However, you must include `boost/uuid/uuid_io.hpp`. This header file provides the overloaded operator to write objects of type `boost::uuids::uuid` to an output stream.

Example 65.1 displays output that looks like the following: `0cb6f61f-be68-5afc-8686-c52e3fc7a50d`. Using dashes is the preferred way of displaying UUIDs.

Example 65.2. Member functions of `boost::uuids::uuid`

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();
  std::cout << id.size() << '\n';
  std::cout << std::boolalpha << id.is_nil() << '\n';
  std::cout << id.variant() << '\n';
  std::cout << id.version() << '\n';
}
```

`boost::uuids::uuid` provides only a few member functions, some of which are introduced in Example 65.2. `size()` returns the size of a UUID in bytes. Because a UUID is always 128 bits, `size()` always returns 16. `is_nil()` returns `true` if the UUID is a nil UUID. The nil UUID is 00000000-0000-0000-0000-000000000000. `variant()` and `version()` specify the kind of UUID and how it was generated. In Example 65.2, `variant()` returns 1, which means the UUID conforms to RFC 4122. `version()` returns 4, which means that the UUID was created by a random number generator.

`boost::uuids::uuid` also provides member functions like `begin()`, `end()`, and `swap()` .

Example 65.3. Generators from Boost.Uuid

```
#include <boost/uuid/uuid.hpp>
#include <boost/uuid/uuid_generators.hpp>
#include <boost/uuid/uuid_io.hpp>
#include <iostream>

using namespace boost::uuids;

int main()
{
  nil_generator nil_gen;
  uuid id = nil_gen();
  std::cout << std::boolalpha << id.is_nil() << '\n';

  string_generator string_gen;
  id = string_gen("CF77C981-F61B-7817-10FF-D916FCC3EAA4");
  std::cout << id.variant() << '\n';

  name_generator name_gen(id);
  std::cout << name_gen("theboostcpplibraries.com") << '\n';
}
```

Example 65.3 contains more generators from Boost.Uuid. `nil_generator` generates a nil UUID. `is_nil()` returns `true` only if the UUID is nil.

You use `string_generator` if you want to use an existing UUID. You can generate UUIDs at sites such as http://www.uuidgenerator.net/. For the UUID in Example 65.3, `variant()` returns 0, which means that the UUID conforms to the backwards compatible NCS standard. `name_generator` is used to generate UUIDs in namespaces.

Please note the spelling of UUIDs when using `string_generator`. You can pass a UUID without dashes, but if you use dashes, they must be in the right places. Case (upper or lower) is ignored.

Example 65.4. Conversion to strings

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

using namespace boost::uuids;

int main()
{
  random_generator gen;
  uuid id = gen();

  std::string s = to_string(id);
  std::cout << s << '\n';

  std::cout << boost::lexical_cast<std::string>(id) << '\n';
}
```

Boost.Uuid provides the functions `boost::uuids::to_string()` and `boost::uuids::to_wstring()` to convert a UUID to a string (see Example 65.4). It is also possible to use `boost::lexical_cast()` for the conversion.

Part XVI. Design Patterns

The following libraries are for design patterns.

- Boost.Flyweight helps in situations where many identical objects are used in a program and memory consumption needs to be reduced.
- Boost.Signals2 makes it easy to use the observer design pattern. This library is called Boost.Signals2 because it implements the signal/slot concept.
- Boost.MetaStateMachine makes it possible to transfer state machines from UML to C++.

Table of Contents

66. Boost.Flyweight
67. Boost.Signals2
68. Boost.MetaStateMachine

Chapter 66. Boost.Flyweight

Boost.Flyweight is a library that makes it easy to use the design pattern of the same name. Flyweight helps save memory when many objects share data. With this design pattern, instead of storing the same data multiple times in objects, shared data is kept in just one place, and all objects refer to that data. While you can implement this design pattern with, for example, pointers, it is easier to use Boost.Flyweight.

Example 66.1. A hundred thousand identical strings without Boost.Flyweight

```
#include <string>
#include <vector>

struct person
{
  int id_;
  std::string city_;
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Example 66.1 creates a hundred thousand objects of type `person`. `person` defines two member variables: **`id_`** identifies persons, and **`city_`** stores the city people live in. In this example, all people live in Berlin. That's why **`city_`** is set to "Berlin" in all hundred thousand objects. Thus, the example uses a hundred thousand strings all set to the same value. With Boost.Flyweight, one string – instead of thousands – can be used and memory consumption reduced.

Example 66.2. One string instead of a hundred thousand strings with Boost.Flyweight

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

To use Boost.Flyweight, include `boost/flyweight.hpp`, as in Example 66.2. Boost.Flyweight provides additional header files that only need to be included if you need to change the detailed

library settings.

All classes and functions are in the namespace `boost::flyweights`. Example 66.2 only uses the class `boost::flyweights::flyweight`, which is the most important class in this library. The member variable **`city_`** uses the type `flyweight<std::string>` rather than `std::string`. This is all you need to change to use this design pattern and reduce the memory requirements of the program.

Example 66.3. Using `boost::flyweights::flyweight` multiple times

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string> city_;
  flyweight<std::string> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Example 66.3 adds a second member variable, **`country_`**, to the class `person`. This member variable contains the names of the countries people live in. Since, in this example, all people live in Berlin, they all live in the same country. That's why `boost::flyweights::flyweight` is used in the definition of the member variable **`country_`**, too.

Boost.Flyweight uses an internal container to store objects. It makes sure there can't be multiple objects with same values. By default, Boost.Flyweight uses a hash container such as `std::unordered_set`. For different types, different hash containers are used. As in Example 66.3, both member variables **`city_`** and **`country_`** are strings; therefore, only one container is used. In this example, this is not a problem because the container only stores two strings: "Berlin" and "Germany." If many different cities and countries must be stored, it would be better to store cities in one container and countries in another.

Example 66.4. Using `boost::flyweights::flyweight` multiple times with tags

```
#include <boost/flyweight.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct city {};
struct country {};
```

```
struct person
{
  int id_;
  flyweight<std::string, tag<city>> city_;
  flyweight<std::string, tag<country>> country_;
  person(int id, std::string city, std::string country)
    : id_{id}, city_{std::move(city)}, country_{std::move(country)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin", "Germany"});
}
```

Example 66.4 passes a second template parameter to `boost::flyweights::flyweight`. This is a *tag*. Tags are arbitrary types only used to differentiate the types on which **`city_`** and **`country_`** are based. Example 66.4 defines two empty structures `city` and `country`, which are used as tags. However, the example could have instead used `int`, `bool`, or any type.

The tags make **`city_`** and **`country_`** use different types. Now two hash containers are used by Boost.Flyweight – one stores cities, the other stores countries.

Example 66.5. Template parameters of `boost::flyweights::flyweight`

```
#include <boost/flyweight.hpp>
#include <boost/flyweight/set_factory.hpp>
#include <boost/flyweight/no_locking.hpp>
#include <boost/flyweight/no_tracking.hpp>
#include <string>
#include <vector>
#include <utility>

using namespace boost::flyweights;

struct person
{
  int id_;
  flyweight<std::string, set_factory<>, no_locking, no_tracking> city_;
  person(int id, std::string city) : id_{id}, city_{std::move(city)} {}
};

int main()
{
  std::vector<person> persons;
  for (int i = 0; i < 100000; ++i)
    persons.push_back({i, "Berlin"});
}
```

Template parameters other than tags can be passed to `boost::flyweights::flyweight`. Example 66.5 passes `boost::flyweights::set_factory`, `boost::flyweights::no_locking`, and `boost::flyweights::no_tracking`. Additional header files are included to make use of these classes.

`boost::flyweights::set_factory` tells Boost.Flyweight to use a sorted container, such as `std::set`, rather than a hash container. With `boost::flyweights::no_locking`, support for multithreading, which is normally activated by default, is deactivated. `boost::flyweights::no_tracking` tells Boost.Flyweight to not track objects stored in internal

containers. By default, when objects are no longer used, Boost.Flyweight detects this and removes them from the containers. When `boost::flyweights::no_tracking` is set, the detection mechanism is disabled. This improves performance. However, containers can only grow and will never shrink.

Boost.Flyweight supports additional settings. Check the official documentation if you are interested in more details on tuning.

Exercise

Improve this program with Boost.Flyweight. Use Boost.Flyweight with disabled support for multithreading:

```
#include <string>
#include <vector>
#include <memory>

int main()
{
    std::vector<std::shared_ptr<std::string>> countries;
    auto germany = std::make_shared<std::string>("Germany");
    for (int i = 0; i < 500; ++i)
        countries.push_back(germany);
    auto netherlands = std::make_shared<std::string>("Netherlands");
    for (int i = 0; i < 500; ++i)
        countries.push_back(netherlands);
}
```

Chapter 67. Boost.Signals2

Table of Contents

Signals
Connections
Multithreading

Boost.Signals2 implements the signal/slot concept. One or multiple functions – called *slots* – are linked with an object that can emit a signal. Every time the signal is emitted, the linked functions are called.

The signal/slot concept can be useful when, for example, developing applications with graphical user interfaces. Buttons can be modelled so they emit a signal when a user clicks on them. They can support links to many functions to handle user input. That way it is possible to process events flexibly.

`std::function` can also be used for event handling. One crucial difference between `std::function` and Boost.Signals2, is that Boost.Signals2 can associate more than one event handler with a single event. Therefore, Boost.Signals2 is better for supporting event-driven development and should be the first choice whenever events need to be handled.

Boost.Signals2 succeeds the library Boost.Signals, which is deprecated and not discussed in this book.

Chapter 68. Boost.MetaStateMachine

Boost.MetaStateMachine is used to define state machines. State machines describe objects through their states. They describe what states exist and what transitions between states are possible.

Boost.MetaStateMachine provides three different ways to define state machines. The code you need to write to create a state machine depends on the front-end.

If you go with the basic front-end or the function front-end, you define state machines in the conventional way: you create classes, derive them from other classes provided by Boost.MetaStateMachine, define required member variables, and write the required C++ code yourself. The fundamental difference between the basic front-end and the function front-end is that the basic front-end expects function pointers, while the function front-end lets you use function objects.

The third front-end is called eUML and is based on a domain-specific language. This front-end makes it possible to define state machines by reusing definitions of a UML state machine. Developers familiar with UML can copy definitions from a UML behavior diagram to C++ code. You don't need to translate UML definitions to C++ code.

eUML is based on a set of macros that you must use with this front-end. The advantage of the macros is that you don't need to work directly with many of the classes provided by Boost.MetaStateMachine. You just need to know which macros to use. This means you can't forget to derive your state machine from a class, which can happen with the basic front-end or the function front-end. This chapter introduces Boost.MetaStateMachine with eUML.

Example 68.1. Simple state machine with eUML

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
```

```
  std::cout << *light.current_state() << '\n';
  light.process_event(press);
  std::cout << *light.current_state() << '\n';
}
```

Example 68.1 uses the simplest state machine possible: A lamp has exactly two states. It is either on or off. If it is off, it can be switched on. If it is on, it can be switched off. It is possible to switch from every state to every other state.

Example 68.1 uses the eUML front-end to describe the state machine of a lamp. Boost.MetaStateMachine doesn't have a master header file. Therefore, the required header files have to be included one by one. `boost/msm/front/euml/euml.hpp` and `boost/msm/front/euml/state_grammar.hpp` provide access to eUML macros. `boost/msm/back/state_machine.hpp` is required to link a state machine from the front-end to a state-machine from the back-end. While front-ends provide various possibilities to define state machines, the actual implementation of a state machine is found in the back-end. Since Boost.MetaStateMachine contains only one back-end, you don't need to select an implementation.

All of the definitions from Boost.MetaStateMachine are in the namespace `boost::msm`. Unfortunately, many eUML macros don't refer explicitly to classes in this namespace. They use either the namespace `msm` or no namespace at all. That's why Example 68.1 creates an alias for the namespace `boost::msm` and makes the definitions in `boost::msm::front::euml` available with a `using` directive. Otherwise the eUML macros lead to compiler errors.

To use the state machine of a lamp, first define the states for off and on. States are defined with the macro `BOOST_MSM_EUML_STATE`, which expects the name of the state as its second parameter. The first parameter describes the state. You'll see later how these descriptions look like. The two states defined in Example 68.1 are called **`Off`** and **`On`**.

To switch between states, events are required. Events are defined with the macro `BOOST_MSM_EUML_EVENT`, which expects the name of the event as its sole parameter. Example 68.1 defines an event called **`press`**, which represents the action of pressing the light switch. Since the same event switches a light on and off, only one event is defined.

When the required states and events are defined, the macro `BOOST_MSM_EUML_TRANSITION_TABLE` is used to create a *transition table*. This tables defines valid transitions between states and which events trigger which state transitions.

`BOOST_MSM_EUML_TRANSITION_TABLE` expects two parameters. The first parameter defines the transition table, and the second is the name of the transition table. The syntax of the first parameter is designed to make it easy to recognize how states and events relate to each other. For example, `Off + press == On` means that the machine in the state **`Off`** switches to the state **`On`** with the event **`press`**. The intuitive and self-explanatory syntax of a transition table definition is one of the strengths of the eUML front-end.

After the transition table has been created, the state machine is defined with the macro `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`. The second parameter is again the simpler one: it

sets the name of the state machine. The state machine in Example 68.1 is named `light_state_machine`.

The first parameter of `BOOST_MSM_EUML_DECLARE_STATE_MACHINE` is a tuple. The first value is the name of the transition table. The second value is an expression using **`init_`**, which is an attribute provided by Boost.MetaStateMachine. You'll learn more about attributes later. The expression `init_ << Off` is required to set the initial state of the state machine to **`Off`**.

The state machine `light_state_machine`, defined with `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`, is a class. You use this class to instantiate a state machine from the back-end. In Example 68.1 this is done by passing `light_state_machine` to the class template `boost::msm::back::state_machine` as a parameter. This creates a state machine called **`light`**.

State machines provide a member function `process_event()` to process events. If you pass an event to `process_event()`, the state machines changes its state depending on its transition table.

To make it easier to see what happens in Example 68.1 when `process_event()` is called multiple times, `current_state()` is called. This member function should only be used for debugging purposes. It returns a pointer to an `int`. Every state is an `int` value assigned in the order the states have been accessed in `BOOST_MSM_EUML_TRANSITION_TABLE`. In Example 68.1 **`Off`** is assigned the value 0 and **`On`** is assigned the value 1. The example writes `0`, `1`, and `0` to standard output. The light switch is pressed two times, which switches the light on and off.

Example 68.2. State machine with state, event, and action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    BOOST_MSM_EUML_STATE_NAME(Off) &sourceState,
    BOOST_MSM_EUML_STATE_NAME(On) &targetState) const
  {
    std::cout << "Switching on\n";
  }

  template <class Event, class Fsm>
  void operator()(const Event &ev, Fsm &fsm,
    decltype(On) &sourceState,
    decltype(Off) &targetState) const
  {
    std::cout << "Switching off\n";
  }
};
```

```
BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.2 extends the state machine for the lamp by an action. An action is executed by an event triggering a state transition. Because actions are optional, a state machine could be defined without them.

Actions are defined with `BOOST_MSM_EUML_ACTION`. Strictly speaking, a function object is defined. You must overload the operator `operator()`. The operator must accept four parameters. The parameters reference an event, a state machine and two states. You are free to define a template or use concrete types for all of the parameters. In Example 68.2, concrete types are only set for the last two parameters. Because these parameters describe the beginning and ending states, you can overload `operator()` so that different member functions are executed for different switches.

Please note that the states **`On`** and **`Off`** are objects. Boost.MetaStateMachine provides a macro `BOOST_MSM_EUML_STATE_NAME` to get the type of a state. If you use C++11, you can use the operator `decltype` instead of the macro.

The action **`switch_light`**, which has been defined with `BOOST_MSM_EUML_ACTION`, is executed when the light switch is pressed. The transition table has been changed accordingly. The first transition is now `Off + press / switch_light == On`. You pass actions after a slash after the event. This transition means that the operator `operator()` of **`switch_light`** is called if the current state is **`Off`** and the event **`press`** happens. After the action has been executed, the new state is **`On`**.

Example 68.2 writes `Switching on` and then `Switching off` to standard output.

Example 68.3. State machine with state, event, guard, and action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)
```

```
BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return true;
  }
};

BOOST_MSM_EUML_ACTION(switch_light)
{
  template <class Event, class Fsm, class Source, class Target>
  void operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    std::cout << "Switching\n";
  }
};

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] / switch_light == On,
  On + press / switch_light == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.3 uses a guard in the transition table. The definition of the first transition is `Off + press [!is_broken] / switch_light == On`. Passing **`is_broken`** in brackets means that the state machine checks before the action **`switch_light`** is called whether the transition may occur. This is called a guard. A guard must return a result of type `bool`.

A guard like **`is_broken`** is defined with `BOOST_MSM_EUML_ACTION` in the same way as actions. Thus, the operator `operator()` has to be overloaded for the same four parameters. `operator()` must have a return value of type `bool` to be used as a guard.

Please note that you can use logical operators like `operator!` on guards inside brackets.

If you run the example, you'll notice that nothing is written to standard output. The action **`switch_light`** is not executed – the light stays off. The guard **`is_broken`** returns `true`. However, because the operator `operator!` is used, the expression in brackets evaluates to `false`.

You can use guards to check whether a state transition can occur. Example 68.3 uses **`is_broken`** to check whether the lamp is broken. While a transition from off to on is usually possible and the transition table describes lamps correctly, in this example, the lamp cannot be switched on. Despite two calls to `process_event()`, the state of **`light`** is **`Off`**.

Example 68.4. State machine with state, event, entry action, and exit action

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
```

```
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Entering\n";
  }
};

BOOST_MSM_EUML_ACTION(state_exit)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Exiting\n";
  }
};

BOOST_MSM_EUML_STATE((state_entry, state_exit), Off)
BOOST_MSM_EUML_STATE((state_entry, state_exit), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off),
light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
}
```

In Example 68.4, the first parameter passed to `BOOST_MSM_EUML_STATE` is a tuple consisting of `state_entry` and `state_exit`. `state_entry` is an entry action, and `state_exit` is an exit action. These actions are executed when a state is entered or exited.

Like actions, entry and exit actions are defined with `BOOST_MSM_EUML_ACTION`. However, the overloaded operator `operator()` expects only three parameters: references to an event, a state machine, and a state. Transitions between states don't matter for entry and exit actions, so only one state needs to be passed to `operator()`. For entry actions, this state is entered. For exit actions, this state is exited.

In Example 68.4, both states **`Off`** and **`On`** have entry and exit actions. Because the event **`press`** occurs twice, `Entering` and `Exiting` is displayed twice. Please note that `Exiting` is displayed first and `Entering` afterwards because the first action executed is an exit action.

The first event **`press`** triggers a transition from **`Off`** to **`On`**, and `Exiting` and `Entering` are each displayed once. The second event **`press`** switches the state to **`Off`**. Again `Exiting` and

`Entering` are each displayed once. Thus, state transitions execute the exit action first, then the entry action of the new state.

Example 68.5. Attributes in a state machine

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

BOOST_MSM_EUML_ACTION(state_entry)
{
  template <class Event, class Fsm, class State>
  void operator()(const Event &ev, Fsm &fsm, State &state) const
  {
    std::cout << "Switched on\n";
    ++fsm.get_attribute(switched_on);
  }
};
BOOST_MSM_EUML_ACTION(is_broken)
{
  template <class Event, class Fsm, class Source, class Target>
  bool operator()(const Event &ev, Fsm &fsm, Source &src, Target &trg) const
  {
    return fsm.get_attribute(switched_on) > 1;
  }
};

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((state_entry), On)
BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [!is_broken] == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.5 uses the guard **`is_broken`** to check whether a state transition from **`Off`** to **`On`** is possible. This time the return value of **`is_broken`** depends on how often the light switch has been pressed. It is possible to switch the light on two times before the lamp is broken. To count how often the light has been switched on, an attribute is used.

Attributes are variables that can be attached to objects. They let you adapt the behavior of state machines at run time. Because data such as how often the light has been switched on has to be

stored somewhere, it makes sense to store it directly in the state machine, in a state, or in an event.

Before an attribute can be used, it has to be defined. This is done with the macro `BOOST_MSM_EUML_DECLARE_ATTRIBUTE`. The first parameter passed to `BOOST_MSM_EUML_DECLARE_ATTRIBUTE` is the type, and the second is the name of the attribute. Example 68.5 defines the attribute **`switched_on`** of type `int`.

After the attribute has been defined, it must be attached to an object. The example attaches the attribute **`switched_on`** to the state machine. This is done via the fifth value in the tuple, which is passed as the first parameter to `BOOST_MSM_EUML_DECLARE_STATE_MACHINE`. With **`attributes_`**, a keyword from Boost.MetaStateMachine is used to create a lambda function. To attach the attribute **`switched_on`** to the state machine, write **`switched_on`** to **`attributes_`** as though it were a stream, using `operator<<`.

The third and fourth values in the tuples are both set to **`no_action`**. The attribute is passed as the fifth value in the tuple. The third and fourth values can be used to define entry and exit actions for the state machine. If no entry and exit actions are defined, use **`no_action`**.

After the attribute has been attached to the state machine, it can be accessed with `get_attribute()`. In Example 68.5, this member function is called in the entry action `state_entry` to increment the value of the attribute. Because `state_entry` is only linked to the state **`On`**, **`switched_on`** is only incremented when the light is switched on.

`switched_on` is also accessed from the guard **`is_broken`**, which checks whether the value of the attribute is greater than 1. If it is, the guard returns `true`. Because attributes are initialized with the default constructor and **`switched_on`** is set to 0, **`is_broken`** returns `true` if the light has been switched on two times.

In Example 68.5, the event **`press`** occurs five times. The light is switched on and off two times and then switched on again. The first two times the light is switched on, `Switched on` is displayed. However, the third time the light is switched on there is no output. This happens because **`is_broken`** returns `true` after the light has been switched on two times, and therefore, there is no state transition from **`Off`** to **`On`**. This means the entry action for the state **`On`** is not executed, and the example does not write to standard output.

Example 68.6. Accessing attributes in transition tables

```
#include <boost/msm/front/euml/euml.hpp>
#include <boost/msm/front/euml/state_grammar.hpp>
#include <boost/msm/back/state_machine.hpp>
#include <iostream>

namespace msm = boost::msm;
using namespace boost::msm::front::euml;

BOOST_MSM_EUML_DECLARE_ATTRIBUTE(int, switched_on)

void write_message()
{
  std::cout << "Switched on\n";
}
```

```
BOOST_MSM_EUML_FUNCTION(WriteMessage_, write_message, write_message_,
  void, void)

BOOST_MSM_EUML_STATE((), Off)
BOOST_MSM_EUML_STATE((), On)

BOOST_MSM_EUML_EVENT(press)

BOOST_MSM_EUML_TRANSITION_TABLE((
  Off + press [fsm_(switched_on) < Int_<2>()] / (++fsm_(switched_on),
    write_message_()) == On,
  On + press == Off
), light_transition_table)

BOOST_MSM_EUML_DECLARE_STATE_MACHINE(
(light_transition_table, init_ << Off, no_action, no_action,
attributes_ << switched_on), light_state_machine)

int main()
{
  msm::back::state_machine<light_state_machine> light;
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
  light.process_event(press);
}
```

Example 68.6 does the same thing as Example 68.5: after switching the light on two times, the light is broken and can't be switched on anymore. While the previous example accessed the attribute **`switched_on`** in actions, this example uses attributes in the transition table.

Boost.MetaStateMachine provides the function `fsm_()` to access an attribute in a state machine. That way a guard is defined that checks whether **`switched_on`** is smaller than 2. And an action is defined that increments **`switched_on`** every time the state switches from **`Off`** to **`On`**.

Please note that the smaller-than comparison in the guard is done with `Int_<2>()`. The number 2 must be passed as a template parameter to `Int_` to create an instance of this class. That creates a function object that has the type needed by Boost.MetaStateMachine.

Example 68.6 also uses the macro `BOOST_MSM_EUML_FUNCTION` to make a function an action. The first parameter passed to `BOOST_MSM_EUML_FUNCTION` is the name of the action that can be used in the function front-end. The second parameter is the name of the function. The third parameter is the name of the action as it is used in eUML. The fourth and fifth parameters are the return values for the function – one for the case where the action is used for a state transition, and the other for the case where the action describes an entry or exit action. After `write_message()` has been turned into an action this way, an object of type `write_message_` is created and used following `++fsm_(switched_on)` in the transition table. In a state transition from **`Off`** to **`On`**, the attribute **`switched_on`** is incremented and then `write_message()` is called.

Example 68.6 displays `Switched on` twice, as in Example 68.5.

Boost.MetaStateMachine provides additional functions, such as `state_()` and `event_()`, to access attributes attached to other objects. Other classes, such as `Char_` and `String_`, can also be used like `Int_`.

Tip As you can see in the examples, the front-end eUML requires you to use many macros. The header file `boost/msm/front/euml/common.hpp` contains definitions for all of the eUML macros, which makes it a useful reference.

Exercises

1. Create a state machine for a window that can be closed, opened or tilted. A closed window can be opened or tilted. An open window can't be tilted though without closing it first. Nor can a tilted window be opened without closing it first. Test your state machine by opening and tilting your window a couple of times. Use `current_state()` to write states to standard output.

2. Extend the state machine: The window should be part of a smart home. The state machine should now count how often the window was opened and tilted. To test your state machine open and tilt your window a couple of times. At the end of your program write to standard output how often the window was opened and how often it was tilted.

Part XVII. Other Libraries

The following libraries provide small but helpful utilities.

- Boost.Utility collects everything that doesn't fit somewhere else in the Boost libraries.
- Boost.Assign provides helper functions that make it easier to perform operations such as adding multiple values to a container without having to call `push_back()` repeatedly.
- Boost.Swap provides a variant of `std::swap()` that is optimized for the Boost libraries.
- Boost.Operators makes it easy to define operators based on other operators.

Table of Contents

69. Boost.Utility
70. Boost.Assign
71. Boost.Swap
72. Boost.Operators

Chapter 69. Boost.Utility

The library Boost.Utility is a conglomeration of miscellaneous, useful classes and functions that are too small to justify being maintained in stand-alone libraries. While the utilities are small and can be learned quickly, they are completely unrelated. Unlike the examples in other chapters, the code samples here do not build on each other, since they are independent utilities.

While most utilities are defined in `boost/utility.hpp`, some have their own header files. The following examples include the appropriate header file for the utility being introduced.

Example 69.1. Using `boost::checked_delete()`

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

struct animal : public boost::intrusive::list_base_hook<>
{
  std::string name_;
  int legs_;

  animal(std::string name, int legs) : name_{std::move(name)},
    legs_{legs} {}
};

int main()
{
  animal *a = new animal{"cat", 4};

  typedef boost::intrusive::list<animal> animal_list;
  animal_list al;

  al.push_back(*a);

  al.pop_back_and_dispose(boost::checked_delete<animal>);
  std::cout << al.size() << '\n';
}
```

Example 69.1 passes the function `boost::checked_delete()` as a parameter to the member function `pop_back_and_dispose()`, which is provided by the class `boost::intrusive::list` from Boost.Intrusive. `boost::intrusive::list` and `pop_back_and_dispose()` are introduced in Chapter 18, while `boost::checked_delete()` is provided by Boost.Utility and defined in `boost/checked_delete.hpp`.

`boost::checked_delete()` expects as its sole parameter a pointer to the object that will be deleted by `delete`. Because `pop_back_and_dispose()` expects a function that takes a pointer to destroy the corresponding object, it makes sense to pass in `boost::checked_delete()` – that way, you don't need to define a similar function.

Unlike `delete`, `boost::checked_delete()` ensures that the type of the object to be destroyed is complete. `delete` will accept a pointer to an object with an incomplete type. While this concerns a detail of the C++ standard that you can usually ignore, you should note that

`boost::checked_delete()` is not completely identical to a call to `delete` because it puts higher demands on its parameter.

Boost.Utility also provides `boost::checked_array_delete()`, which can be used to destroy arrays. It calls `delete[]` rather than `delete`.

Additionally, two classes, `boost::checked_deleter` and `boost::checked_array_deleter`, are available to create function objects that behave like `boost::checked_delete()` and `boost::checked_array_delete()`, respectively.

Example 69.2. Using `BOOST_CURRENT_FUNCTION`

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

int main()
{
  const char *funcname = BOOST_CURRENT_FUNCTION;
  std::cout << funcname << '\n';
}
```

Example 69.2 uses the macro `BOOST_CURRENT_FUNCTION`, defined in `boost/current_function.hpp`, to return the name of the surrounding function as a string.

`BOOST_CURRENT_FUNCTION` provides a platform-independent way to retrieve the name of a function. Starting with C++11, you can do the same thing with the standardized macro `__func__`. Before C++11, compilers like Visual C++ and GCC supported the macro `__FUNCTION__` as an extension. `BOOST_CURRENT_FUNCTION` uses whatever macro is supported by the compiler.

If compiled with Visual C++ 2013, Example 69.2 displays `int __cdecl main(void)`.

Example 69.3. Using `boost::prior()` and `boost::next()`

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

int main()
{
  std::array<char, 4> a{{'a', 'c', 'b', 'd'}};

  auto it = std::find(a.begin(), a.end(), 'b');
  auto prior = boost::prior(it, 2);
  auto next = boost::next(it);

  std::cout << *prior << '\n';
  std::cout << *it << '\n';
  std::cout << *next << '\n';
}
```

Boost.Utility provides two functions, `boost::prior()` and `boost::next()`, that return an iterator relative to another iterator. In Example 69.3, **`it`** points to “b” in the array, **`prior`** points to “a”, and **`next`** to “d”.

Unlike `std::advance()`, `boost::prior()` and `boost::next()` return a new iterator and do not modify the iterator that was passed in.

In addition to the iterator, both functions accept a second parameter that indicates the number of steps to move forward or backward. In Example 69.3, the iterator is moved two steps backward in the call to `boost::prior()` and one step forward in the call to `boost::next()`.

The number of steps is always a positive number, even for `boost::prior()`, which moves backwards.

To use `boost::prior()` and `boost::next()`, include the header file `boost/next_prior.hpp`.

Both functions were added to the standard library in C++11, where they are called `std::prev()` and `std::next()`. They are defined in the header file `iterator`.

Example 69.4. Using `boost::noncopyable`

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

struct animal : boost::noncopyable
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}
};

void print(const animal &a)
{
  std::cout << a.name << '\n';
  std::cout << a.legs << '\n';
}

int main()
{
  animal a{"cat", 4};
  print(a);
}
```

Boost.Utility provides the class `boost::noncopyable`, which is defined in `boost/noncopyable.hpp`. This class makes it impossible to copy (and move) objects.

The same effect can be achieved by defining the copy constructor and assignment operator as private member functions or – since C++11 – by removing the copy constructor and assignment operator with `delete`. However, deriving from `boost::noncopyable` explicitly states the intention that objects of a class should be non-copyable.

Note

Some developers prefer `boost::noncopyable` while others prefer to remove member functions explicitly with `delete`. You will find arguments for both approaches at Stack Overflow, among other places.

Example 69.4 can be compiled and executed. However, if the signature of the `print()` function is modified to take an object of type `animal` by value rather than by reference, the resulting code will no longer compile.

Example 69.5. Using `boost::addressof()`

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

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

  int operator&() const { return legs; }
};

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

To retrieve the address of a particular object, even if `operator&` has been overloaded, Boost.Utility provides the function `boost::addressof()`, which is defined in `boost/utility/addressof.hpp` (see Example 69.5). With C++11, this function became part of the standard library and is available as `std::addressof()` in the header file `memory`.

Example 69.6. Using `BOOST_BINARY`

```
#include <boost/utility/binary.hpp>
#include <iostream>

int main()
{
  int i = BOOST_BINARY(1001 0001);
  std::cout << i << '\n';

  short s = BOOST_BINARY(1000 0000 0000 0000);
  std::cout << s << '\n';
}
```

The macro `BOOST_BINARY` lets you create numbers in binary form. Standard C++ only supports hexadecimal and octal forms, using the prefixes `0x` and `0`. C++11 introduced user-defined literals, which allows you to define custom suffixes, but there still is no standard way of using numbers in binary form in C++11.

Example 69.6 displays `145` and `-32768`. The bit sequence stored in **s** represents a negative number because the 16-bit type `short` uses the 16th bit – the most significant bit in `short` – as the sign bit.

`BOOST_BINARY` simply offers another option to write numbers. Because, in C++, the default type for numbers is `int`, `BOOST_BINARY` also uses `int`. To define a number of type `long`, use the macro `BOOST_BINARY_L`, which generates the equivalent of a number suffixed with the letter L.

Boost.Utility includes additional macros such as `BOOST_BINARY_U`, which initializes a variable without a sign bit. All of these macros are defined in the header file `boost/utility/binary.hpp`.

Example 69.7. Using `boost::string_ref`

```
#include <boost/utility/string_ref.hpp>
#include <iostream>

boost::string_ref start_at_boost(boost::string_ref s)
{
  auto idx = s.find("Boost");
  return (idx != boost::string_ref::npos) ? s.substr(idx) : "";
}

int main()
{
  boost::string_ref s = "The Boost C++ Libraries";
  std::cout << start_at_boost(s) << '\n';
}
```

Example 69.7 introduces the class `boost::string_ref`, which is a reference to a string that only supports read access. To a certain extent, the reference is comparable with `const std::string&`. However, `const std::string&` requires the existence of an object of type `std::string`. `boost::string_ref` can also be used without `std::string`. The benefit of `boost::string_ref` is that, unlike `std::string`, it requires no memory to be allocated.

Example 69.7 looks for the word “Boost” in a string. If found, a string starting with that word is displayed. If the word “Boost” isn’t found, an empty string is displayed. The type of the string **s** in `main()` isn’t `std::string`, it’s `boost::string_ref`. Thus no memory is allocated with `new` and no copy is created. **s** points to the literal string “The Boost C++ Libraries” directly.

The type of the return value of `start_at_boost()` is `boost::string_ref`, not `std::string`. The function doesn’t return a new string, it returns a reference. This reference is to either a substring of the parameter or an empty string. `start_at_boost()` requires that the original string remains valid as long as references of type `boost::string_ref` are in use. If this is guaranteed, as in Example 69.7, memory allocations can be avoided.

Additional utilities are also available, but they are beyond the scope of this book because they are mostly used by the developers of Boost libraries or for template meta programming. The documentation of Boost.Utility provides a fairly comprehensive overview of these additional utilities and can serve as a starting point if you are interested.

Chapter 70. Boost.Assign

The library Boost.Assign provides helper functions to initialize containers or add elements to containers. These functions are especially helpful if many elements need to be stored in a container. Thanks to the functions offered by Boost.Assign, you don't need to call a member function like `push_back()` repeatedly to insert elements one by one into a container.

If you work with a development environment that supports C++11, you can benefit from initializer lists. Usually you can pass any number of values to the constructor to initialize containers. Thanks to initializer lists, you don't have to depend on Boost.Assign with C++11. However, Boost.Assign provides helper functions to add multiple values to an existing container. These helper functions can be useful in C++11 development environments.

Example 70.1 introduces a few functions that containers can be initialized with. To use the functions defined by Boost.Assign, include the header file `boost/assign.hpp`.

Example 70.1. Initializing containers

```
#include <boost/assign.hpp>
#include <boost/tuple/tuple.hpp>
#include <vector>
#include <stack>
#include <map>
#include <string>
#include <utility>

using namespace boost::assign;

int main()
{
  std::vector<int> v = list_of(1)(2)(3);

  std::stack<int> s = list_of(1)(2)(3).to_adapter();

  std::vector<std::pair<std::string, int>> v2 =
    list_of<std::pair<std::string, int>>("a", 1)("b", 2)("c", 3);

  std::map<std::string, int> m =
    map_list_of("a", 1)("b", 2)("c", 3);

  std::vector<boost::tuple<std::string, int, double>> v3 =
    tuple_list_of("a", 1, 9.9)("b", 2, 8.8)("c", 3, 7.7);
}
```

Boost.Assign provides three functions to initialize containers. The most important, and the one you usually work with, is `boost::assign::list_of()`. You use `boost::assign::map_list_of()` with `std::map` and `boost::assign::tuple_list_of()` to initialize tuples in a container.

You don't have to use `boost::assign::map_list_of()` or `boost::assign::tuple_list_of()`. You can initialize any container with `boost::assign::list_of()`. However, if you use `std::map` or a container with tuples, you must pass a template parameter to `boost::assign::list_of()` that tells the function how

elements are stored in the container. This template parameter is not required for `boost::assign::map_list_of()` and `boost::assign::tuple_list_of()`.

All three functions return a proxy object. This object overloads the operator `operator()`. You can call this operator multiple times to save values in the container. Even though you access another object, and not the container, the container is changed through this proxy object.

If you want to initialize adapters like `std::stack`, call the member function `to_adapter()` on the proxy. The proxy then calls the member function `push()`, which is provided by all adapters.

`boost::assign::tuple_list_of()` supports tuples of type `boost::tuple` only. You cannot use this function to initialize containers with tuples from the standard library.

Example 70.2 illustrates how values can be added to existing containers.

Example 70.2. Adding values to containers

```
#include <boost/assign.hpp>
#include <vector>
#include <deque>
#include <set>
#include <queue>

int main()
{
  std::vector<int> v;
  boost::assign::push_back(v)(1)(2)(3);

  std::deque<int> d;
  boost::assign::push_front(d)(1)(2)(3);

  std::set<int> s;
  boost::assign::insert(s)(1)(2)(3);

  std::queue<int> q;
  boost::assign::push(q)(1)(2)(3);
}
```

The `boost::assign::push_back()`, `boost::assign::push_front()`, `boost::assign::insert()`, and `boost::assign::push()` functions of Boost.Assign return a proxy. You pass these functions the container you want to add new elements to. Then, you call the operator `operator()` on the proxy and pass the values you want to store in the container.

The four functions `boost::assign::push_back()`, `boost::assign::push_front()`, `boost::assign::insert()`, and `boost::assign::push()` are called in this manner because the proxies returned call the identically named member functions on the container. Example 70.2 adds the three numbers 1, 2, and 3 to the vector **v** with three calls to `push_back()`.

Boost.Assign provides additional helper functions you can use to add values to a container, including `boost::assign::add_edge()`, which you can use to get a proxy for a graph from Boost.Graph.

Exercise

Improve this program with Boost.Assign:

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

int main()
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(2);
    v.push_back(3);
    std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout));
    v.push_back(4);
    v.push_back(5);
    v.push_back(6);
    std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout));
}
```

Chapter 71. Boost.Swap

If you use many Boost libraries and also use `std::swap()` to swap data, consider using `boost::swap()` as an alternative. `boost::swap()` is provided by Boost.Swap and is defined in `boost/swap.hpp`.

Example 71.1. Using `boost::swap()`

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

int main()
{
  char c1 = 'a';
  char c2 = 'b';

  boost::swap(c1, c2);

  std::cout << c1 << c2 << '\n';

  boost::array<int, 1> a1{{1}};
  boost::array<int, 1> a2{{2}};

  boost::swap(a1, a2);

  std::cout << a1[0] << a2[0] << '\n';
}
```

`boost::swap()` does nothing different from `std::swap()`. However, because many Boost libraries offer specializations for swapping data that are defined in the namespace `boost`, `boost::swap()` can take advantage of them. In Example 71.1, `boost::swap()` accesses `std::swap()` to swap the values of the two `char` variables and uses the optimized function `boost::swap()` from Boost.Array to swap data in the arrays.

Example 71.1 writes `ba` and `21` to the standard output stream.

Chapter 72. Boost.Operators

Boost.Operators provides numerous classes to automatically overload operators. In Example 72.1, a greater-than operator is automatically added, even though there is no declaration, because the greater-than operator can be implemented using the already defined less-than operator.

Example 72.1. Greater-than operator with `boost::less_than_comparable`

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

struct animal : public boost::less_than_comparable<animal>
{
  std::string name;
  int legs;

  animal(std::string n, int l) : name{std::move(n)}, legs{l} {}

  bool operator<(const animal &a) const { return legs < a.legs; }
};

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

  std::cout << std::boolalpha << (a2 > a1) << '\n';
}
```

To automatically add operators, derive a class from classes defined by Boost.Operators in `boost/operators.hpp`. If a class is derived from `boost::less_than_comparable`, then `operator>`, `operator<=`, and `operator>=` are automatically defined.

Because many operators can be expressed in terms of other operators, automatic overloading is possible. For example, `boost::less_than_comparable` implements the greater-than operator as the opposite of the less-than operator; if an object isn't less than another, it must be greater, assuming they aren't equal.

If two objects can be equal, use `boost::partially_ordered` as the base class. By defining `operator==`, `boost::partially_ordered` can determine whether less than really means greater than or equal.

In addition to `boost::less_than_comparable` and `boost::partially_ordered`, classes are provided that allow you to overload arithmetic and logical operators. Classes are also available to overload operators usually provided by iterators, pointers, or arrays. Because automatic overloading is only possible once other operators have been defined, the particular operators that must be provided will vary depending on the situation. Consult the documentation for more information.