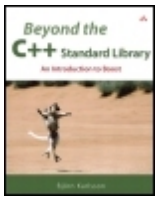


Beyond the C++ Standard Library: An Introduction to Boost
By Björn Karlsson

.....
Publisher: **Addison Wesley Professional**
Pub Date: **August 31, 2005**
ISBN: **0321133544**
Pages: **432**

Table of Contents | Index

Overview

Introducing the Boost libraries: the next breakthrough in C++ programming

Boost takes you far beyond the C++ Standard Library, making C++ programming more elegant, robust, and productive. Now, for the first time, a leading Boost expert systematically introduces the broad set of Boost libraries and teaches best practices for their use.

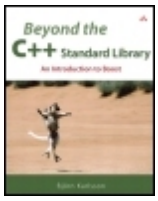
Writing for intermediate-to-advanced C++ developers, Björn Karlsson briefly outlines all 58 Boost libraries, and then presents comprehensive coverage of 12 libraries you're likely to find especially useful. Karlsson's topics range from smart pointers and conversions to containers and data structures, explaining exactly how using each library can improve your code. He offers detailed coverage of higher-order function objects that enable you to write code that is more concise, expressive, and readable. He even takes you "behind the scenes" with Boost, revealing tools and techniques for creating your own generic libraries.

Coverage includes

- Smart pointers that provide automatic lifetime management of objects and simplify resource sharing
- Consistent, best-practice solutions for performing type conversions and lexical conversions
- Utility classes that make programming simpler and clearer
- Flexible container libraries that solve common problems not covered by the C++ Standard Library
- Powerful support for regular expressions with Boost.Regex
- Function objects defined at the call site with Boost.Bind and Boost.Lambda
- More flexible callbacks with Boost.Function
- Managed signals and slots (a.k.a. the Observer pattern) with Boost.Signals

The Boost libraries are proving so useful that many of them are planned for inclusion in the next version of the C++ Standard Library. Get your head start now, with Beyond the C++ Standard Library.

NEXT



Beyond the C++ Standard Library: An Introduction to Boost
By Björn Karlsson
.....
Publisher: **Addison Wesley Professional**
Pub Date: **August 31, 2005**
ISBN: **0321133544**
Pages: **432**

Table of Contents | Index

PREV NEXT

Copyright

Library of Congress Catalog Number: 2005927496

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.

First printing, August 2005

Dedication

In memory of the dead, in honor of the living.

Foreword

Good things are happening in the C++ community. Although C++ remains the most widely used programming language in the world, it is becoming even more powerful and yet easier to use. Skeptical? Bear with me.

The current version of standard C++, which was finalized in 1998, offers robust support for traditional procedural programming as well as object-oriented and generic programming. Just as old (pre-1998) C++ was single-handedly responsible for putting object-oriented within the reach of the workaday software developer, C++98 has done the same for generic programming. The integration of the Standard Template Library (STL) into standard C++ in the mid-1990s represented as much a paradigm shift as did Bjarne Stroustrup's adding classes to C in the early 1980s. Now that the majority of C++ practitioners are proficient with concepts of STL, it's once again time to raise the bar.

Applications of the power of C++ are still being discovered. Many of today's C++ libraries, and mathematical libraries in particular, take routine advantage of template metaprogramming, a fortuitous but unforeseen result of the brilliant design of C++ templates. As higher-level tools and techniques come to light in the C++ community, developing increasingly complex applications is becoming more straightforward and enjoyable.

It is difficult to overstate the importance of Boost to the world of C++. Since the ratification of C++98, no entity outside of the ISO Committee for Standard C++ (called WG21) has done more to influence the direction of C++ than has Boost (and many Boost subscribers are prominent members of WG21, including its founder, my friend Beman Dawes). The thousands of experienced Boost volunteers have, in unselfish, peer-reviewed fashion, developed many useful library solutions not provided by C++98. Ten of its offerings have already been accepted to be integrated into the upcoming C++0x library, and more are under consideration. Where a library approach has been shown to be wanting, the wisdom gained from the cross-pollination of Boost and WG21 has suggested a few modest language enhancements, which are now being entertained.

In the rare case that you haven't heard of Boost, let me ask…do you need to convert between text and numbers or (better yet) between any streamable types? No problemuse Boost.lexical_cast. Oh, you have more sophisticated text processing requirements? Then Boost.Tokenizer or Boost.Regex might be for you, or Boost.Spirit, if you need full-blown parsing. Boost.Bind will amaze you with its function projection and composition capabilities. For functional programming there is Boost.Lambda. Static assertions? Got 'em. If you're mathematically inclined, get your pencil out: You have Boost.Math, Graph, Quaternion, Octonion, MultiArray, Random, and Rational. If you are fortunate enough to have discovered the joy of Python, you can use it and C++ together with the help of Boost.Python. And you can practically pick your platform for all of the above.

Björn Karlsson is a Boost enthusiast and a heartfelt supporter of the C++ community. He has published useful and well-written articles in the C/C++ Users Journal and, more recently, for The C++ Source, a new online voice for the C++ community (see www.artima.com/cppsource). In this volume, he motivates and illustrates key Boost components, and shows how they work with and extend the C++ Standard Library. Consider this not only an in-depth tutorial on Boost, but also a foretaste of the future version of Standard C++. Enjoy!

Chuck Allison, Editor, The C++ Source

Preface

Dear Reader,

Welcome to Beyond the C++ Standard Library: An Introduction to Boost.

If you are interested in generic programming, library design, and the C++ Standard Library, this book is for you. Because the intended audience for the book is intermediate to advanced C++ programmers, there is little coverage of basic C++ concepts. As the title suggests, the focus of this book is on the Boost librariesgeneral usage, best practices, implementation techniques, and design rationale.

Almost from the day I discovered Boost, the people behind it, and the extraordinary libraries in it, I've wanted to write this book. It is amazing that a language as mature as C++ still offers room for exploration into higher-level abstractions as well as technical detail, all without requiring changes to the language. Of course, this is what sets C++ apart from many other programming languages: It is specifically and intentionally designed for extension, and the language's facilities for generic constructs are extremely powerful. This exploration is at the core of the Boost libraries and the Boost community itself. Boost is about making programming in C++ more elegant, more robust, and more productive. As discoveries are made and best practices are shaped, a great challenge faces the C++ community; to share this knowledge with others. In isolation, there is limited value to these remarkable findings, but when exposed to a larger audience, a whole industry will evolve.

This book shows how to use a selection of the wonderfully useful Boost libraries, teaches best practices for their use, and even goes behind the scenes to see how they actually work. The Boost libraries' license grants permission to copy, use, and modify the software for any use (commercial and non-commercial), so all you need to do is visit www.boost.org and download the latest version.

For all the C++ Standard Library aficionados out there, it is well known that a new revision of the Standard Library is in progress. From a standardization point of view, there are three primary areas where the C++ Standard Library is likely to change:

- Fixing broken libraries
- Augmenting missing features to existing libraries
- Adding libraries that provide functionality that is missing in the Standard Library

The Boost libraries address all of these areas in one way or another. Of the 12 libraries covered in this book, six have already been accepted for inclusion in the upcoming Library Technical Report, which means that they will most likely be part of the next version of the Standard Library. Thus, learning about these libraries has excellent long-term value. I hope that you will find this book to be a valuable tool for using, understanding, and extending the Boost libraries. From that vantage, you'll want to incorporate those libraries and the knowledge enshrined within them into your own designs and implementations. That's what I call reuse.

Thank you for reading.

Björn Karlsson

Acknowledgments

A number of people have made all the difference for this book, and for my ability to write it. First of all, I wish to thank the Boost community for these astonishing libraries. Theythe libraries and the Boostersmake a very real difference for the C++ community and the whole software industry. Then, there are people who have very actively supported this effort of mine, and I wish to thank them personally. It's inevitable that I will fail to mention some: To all of you, please accept my sincere apologies. Beman Dawes, thank you for creating Boost in the first place, and for hooking me up with Addison-Wesley. Bjarne Stroustrup, thank you for providing guidance and pointing out important omissions from the nearly finished manuscript. Robert Stewart, thank you for the careful technical and general editing of this book. Rob has made this book much more consistent, more readable, and more accurateand all of this on his free time! The technical errors that remain are mine, not his. Rob has also been instrumental in finding ways to help the reader stay on track even when the author strays. Chuck Allison, thank you for your continuous encouragement and support for my authoring goals. David Abrahams, thank you for supporting this effort and for helping out with reviewing. Matthew Wilson, thank you for reviewing parts of this book and for being a good friend. Gary Powell, thank you for the excellent reviews and for your outstanding enthusiasm for this endeavor. All of the authors of Boost libraries have created online documentation for them: Without this great source of information, it nearly would have been impossible to write this book. Thanks to all of you. Many Boosters have helped out in different ways, and special thanks go to those who have reviewed various chapters of this book. Without their help, important points would not have been made and errors would have prevailed. Aleksey Gurtovoy, David Brownell, Douglas Gregor, Duane Murphy, Eric Friedman, Eric Niebler, Fernando Cacciola, Frank Griswold, Jaakko Järvi, James Curran, Jeremy Siek, John Maddock, Kevlin Henney, Michiel Salters, Paul Grenyer, Peter Dimov, Ronald Garcia, Phil Boyd, Thorsten Ottosen, Tommy Svensson, and Vladimir Prusthank you all so much!

Special thanks go to Microsoft Corporation and Comeau Computing for providing me with their excellent compilers.

I have also had the pleasure of working with two excellent editors from Addison-Wesley. Deborah Lafferty helped me with all of the initial work, such as creating the proposal for the book, and basically made sure that I came to grips with many of the authoring details that I was previously oblivious to. Peter Gordon, skillfully assisted by Kim Boedigheimer, took over the editing of the book and led it through to publishing. Further assistance was given by Lori Lyons, project editor, and Kelli Brooks, copy editor. I wish to thank them allfor making the book possible and for seeing it through to completion.

Friends and family have supported my obsession with C++ for many years now; thank you so much for being there, always.

And finally, many thanks to my wife Jeanette and our son SimonI am forever grateful for your love and support. I will always do my best to deserve it.

About the Author

Björn Karlsson works as a Senior Software Engineer at ReadSoft, where he spends most of his time designing and programming in C++. He has written a number of articles about C++ and the Boost libraries for publications such as C/C++ Users Journal, Overload, and the online journal The C++ Source.

Karlsson is a member of the advisory board for The C++ Source and has been a member of the editorial board of C/C++ Users Journal, where he is also one of the columnists in the Experts Forum. He participates in the Boost newsgroups and is one of the Boost-Users moderators.

Organization of This Book

This book is divided into three main parts, each containing libraries pertaining to a certain domain, but there is definitely overlap. These divisions exist to make it easier to find relevant information for your task at hand or to read the book and find related topics grouped together. Most of the chapters cover a single library, but a few consist of small collections.

The typesetting and coding style is intentionally kept simple. There are a number of popular best practices in this area, and I've just picked one I feel that most people are accustomed to, and that will convey information easily. Furthermore, the coding style in this book purposely tries to save some vertical space by avoiding curly braces on separate lines.

Although the examples in most books make heavy use of using declarations and using directives, this is not the case here. I have done my best to qualify names in the interest of clarity. There is an additional benefit to doing so in this book, and that is to show where the types and functions come from. If something is from the Standard Library, it will be prefixed with std::. If it's from Boost, it will be prefixed with boost::.

Some of the libraries covered by this book are very extensive, which makes it impossible to include detailed explanations of all aspects of the library. When this is the case, there's typically a note stating that there is more to know, with references to the online documentation, related literature, or both. Also, I have tried to focus on the things that are of the most immediate use, and that have a strong relation with the C++ Standard Library.

The first part of this book covers general libraries, which are libraries that are eminently useful, but have no other obvious affinity. The second discusses important data structures and containers. The third is about higher-order programming. There's no requirement to read about the libraries in a specific order, but it certainly doesn't hurt to follow tradition and start from the beginning.

Before getting to the in-depth look at the covered Boost libraries, a survey of each of the currently available Boost libraries will introduce you to the Boost libraries and give context for those that I'll address in the rest of the book. It gives an interesting overview of the versatility of this world-class collection of C++ libraries.

Introduction to Boost

Because you are reading this book, I expect that you are somewhat familiar with the Boost libraries, or that you at least have heard of Boost. There are a great number of libraries in Boost, and there are few, if any, that will not be of at least some interest to you. As a result, you will most definitely find libraries you can put to immediate use. The Boost libraries range over a wide variety of domainsfrom numeric libraries to smart pointers, from a library for template metaprogramming to a preprocessor library, from threading to lambda expressions, and so on. All of the Boost libraries are compatible with a very generous license, which ensures that the libraries can be freely used in commercial applications. Support is available through newsgroups, where much of the activity of the Boost community takes place, and there is at least one company that specializes in consulting related to the Boost libraries. For an online introduction to the Boost community, I strongly suggest that you visit Boost on the Web at www.boost.org.

As of the time of this writing, the current Boost release is 1.32.0. In it, there are 58 separate libraries. The following pages introduce all 58 of those libraries sorted by category and give a short description of what the libraries have to offer. For the libraries not covered in detail in this book, have a look at the documentation provided at www.boost.org, which is also where you go to download the Boost libraries.

String and Text Processing

Boost.Regex

Regular expressions are essential for solving a great number of pattern-matching problems. They are often used to process large strings, find inexact substrings, tokenize a string depending on some format, or modify a string based on certain criteria. The lack of regular expressions support in C++ has sometimes forced users to look at other languages known for their powerful regular expression support, such as Perl, awk, and sed. Regex provides efficient and powerful regular expression support, designed on the same premises as the Standard Template Library (STL), which makes it intuitive to use. Regex has been accepted for the upcoming Library Technical Report. For more information, see "Library 5: Regex."

The author of Regex is Dr. John Maddock.

Boost.Spirit

The Spirit library is a functional, recursive-decent parser generator framework. With it, you can create command-line parsers, even a language preprocessor.[1] It allows the programmer to specify the grammar rules directly in C++ code, using (an approximation of) EBNF syntax. Parsers are typically hard to write properly, and when targeted at a specific problem, they quickly become hard to maintain and understand. Spirit avoids these problems, while giving the same or nearly the same performance as a hand-tuned parser.

[1] The Wave library illustrates this point by using Spirit to implement a highly conformant C++ preprocessor.

The author of Spirit is Joel de Guzman, together with a team of skilled programmers.

Boost.String_algo

This is a collection of string-related algorithms. There are a number of useful algorithms for converting case, trimming strings, splitting strings, finding/replacing, and so forth. This collection of algorithms is an extension to those in the C++ Standard Library.

The author of String_algo is Pavol Droba.

Boost.Tokenizer

This library offers ways of separating character sequences into tokens. Common parsing tasks include finding the data in delimited text streams. It is beneficial to be able to treat such a sequence as a container of elements, where the elements are delimited according to user-defined criteria. Parsing is a separate task from operating on the elements, and it is exactly this abstraction that is offered by Tokenizer. The user determines how the character sequence is delimited, and the library finds the tokens as the user requests new elements.

The author of Tokenizer is John Bandela.

Data Structures, Containers, Iterators, and Algorithms

Boost.Any

The Any library supports typesafe storage and retrieval of values of any type. When the need for a variant type arises, there are three possible solutions:

- Indiscriminate types, such as void*. This solution can almost never be made typesafe; avoid it like the plague.
- Variant typesthat is, types that support the storage and retrieval of a set of types.
- Types that support conversions, such as between string types and integral types.

Any implements the second solutiona value-based variant type, with an unbounded set of possible types. The library is often used for storing heterogeneous types in Standard Library containers. Read more in "Library 6: Any."

The author of Any is Kevlin Henney.

Boost.Array

This library is a wrapper around ordinary C-style arrays, augmenting them with the functions and typedefs from the Standard Library containers. In effect, this makes it possible to treat ordinary arrays as Standard Library containers. This is useful because it adds safety without impeding efficiency and it enables uniform syntax for Standard Library containers and ordinary arrays. The latter means that it enables the use of ordinary arrays with most functions that require a container type to operate on. Array is typically used when performance issues mandate that ordinary arrays be used rather than std::vector.

The author of Array is Nicolai Josuttis, who built the library upon ideas brought forth by Matt Austern and Bjarne Stroustrup.

Boost.Compressed_pair

This library consists of a single parameterized type, compressed_pair, which is very similar to the Standard Library's std::pair. The difference from std::pair is that boost::compressed_pair evaluates the template arguments to see if one of them is empty and, if so, uses the empty base optimization to compress the size of the pair.

Boost.Compressed_pair is used for storing a pair, where one or both of the types is possibly empty.

The authors of Compressed_pair are Steve Cleary, Beman Dawes, Howard Hinnant, and John Maddock.

Boost.Dynamic_bitset

The Dynamic_bitset library very closely resembles std::bitset, except that whereas std::bitset is parameterized on the number of bits (that is, the size of the container), boost::dynamic_bitset supports runtime size configuration. Although dynamic_bitset supports the same interface as std::bitset, it adds functions that support runtime-specific functionality and some that aren't available in std::bitset. The library is typically used instead of std::bitset, in scenarios where the size of the bitset isn't necessarily known at compile time, or may change during program execution.

The authors of Dynamic_bitset are Jeremy Siek and Chuck Allison.

Boost.Graph

Function Objects and Higher-Order Programming

Boost.Bind

Bind is a generalization of the Standard Library binders, bind1st and bind2nd. The library supports binding arguments to anything that behaves like a functionfunction pointers, function objects, and member function pointers with a uniform syntax. It also enables functional composition by means of nested binders. This library does not have all of the requirements that are imposed by the Standard Library binders, most notably that there is often no need to provide the typedefs result_type, first_argument_type, and second_argument_type for your classes. This library also makes it unnecessary to use the adaptors ptr_fun, mem_fun, and mem_fun_ref. The Bind library is thoroughly covered in "Library 9: Bind 9." It's an important and very useful addition to the C++ Standard Library. Bind is typically used with the Standard Library algorithms, and is often used together with Boost.Function, yielding a powerful tool for storing arbitrary functions and function objects for subsequent invocation. Bind has been accepted for the upcoming Library Technical Report.

The author of Bind is Peter Dimov.

Boost.Function

The Function library implements a generalized callback mechanism. It provides for the storage and subsequent invocation of function pointers, function objects, and member function pointers. Of course, it works with binder libraries such as Boost.Bind and Boost.Lambda, which greatly increases the number of use cases for callbacks (including stateful callback functions). The library is covered in detail in "Library 11: Function 11." Function is typically used where a function pointer would otherwise be employed to provide callbacks. Examples of usage are in signal/slot implementations, separation of GUIs from business logic, and storage of heterogeneous function-like types in Standard Library containers. Function has been accepted for the upcoming Library Technical Report.

The author of Function is Douglas Gregor.

Boost.Functional

The Functional library provides enhanced versions of the adapters in the C++ Standard Library. The major advantage is that it helps solve the problem with references to references (which are illegal) that arise when using the Standard Library binders with functions taking one or more arguments by reference. Functional also obviates the use of ptr_fun for using function pointers with the Standard Library algorithms.

The author of Functional is Mark Rodgers.

Boost.Lambda

Lambda provides lambda expressionsunnamed functionsfor C++. Especially useful when using the Standard Library algorithms, Lambda allows functions to be created at the call site, which avoids the creation of many small function objects. Using lambdas means writing less code, and writing it in the location where it's to be used, which is much clearer and maintainable than scattering function objects around the code base. "Library 10: Lambda 10" covers this library in detail.

The authors of Lambda are Jaakko Järvi and Gary Powell.

Boost.Ref

Many function templates, including a large number from the Standard C++ Library, take their arguments by value, which is sometimes problematic. It may be expensive or impossible to copy an object, or the state may be tied to a particular instance, so copying is unwanted. In these situations, one needs a way to pass by reference rather than by value. Ref wraps a reference to an object and turns it into an object that may be copied. This permits calling functions taking their arguments by value with a reference. Ref has been accepted for the upcoming Library Technical Report.

Generic Programming and Template Metaprogramming

Boost.Call_traits

This library provides automatic deduction of the best way of passing arguments to functions, based upon on the argument type. For example, when passing built-in types such as int and double, it is most efficient to pass them by value. For user-defined types, passing them by reference to const is generally preferable. Call_traits automatically selects the right argument type for you. The library also helps in declaring arguments as references, without imposing restrictions or risking references to references (which are illegal in C++). Call_traits is typically used with generic functions that require the most efficient way of passing arguments without knowing much about the argument types beforehand, and to avoid the reference-to-reference problem.

The authors of Call_traits are Steve Cleary, Beman Dawes, Howard Hinnant, and John Maddock.

Boost.Concept_check

Concept_check supplies class templates that are used to test certain concepts (set of requirements). Generic (as in parameterized) code typically requires that the types with which it is instantiated model some abstraction, such as LessThanComparable. This library provides the means to explicitly state the requirements of the parameterizing types for templates. Clients of the code benefit because the requirements are documented and because the compiler can produce an error message that explicitly states how a type failed to meet them. Boost.Concept_check provides more than 30 concepts that can be used for generic code, and several archetypes that may be used to verify that component implementations include all relevant concepts. It is used to assert and document the requirements for concepts in generic code.

The author of Concept_check is Jeremy Siek, who was inspired by previous work by Alexander Stepanov and Matt Austern.

Boost.Enable_if

Enable_if allows function templates or class template specializations to include or exclude themselves from a set of matching functions or specializations. The main use cases are to include or exclude based on some property of the parameterizing typefor example, enabling a function template only when instantiated with an integral type. The library also offers a very useful studying opportunity of SFINAE (substitution failure is not an error).

The authors of Enable_if are Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine.

Boost.In_place_factory

The In_place_factory library is a framework for direct construction of contained objects, including variadic argument lists for initialization. This can reduce the typical requirement that contained types be CopyConstructible, and alleviates the need to create unnecessary temporaries used only for the purpose of providing a source object to be copied from. The library helps minimize the work needed to forward the arguments used for initialization of the contained object.

The author of In_place_factory is Fernando Cacciola.

Boost.Mpl

Mpl is a library for template metaprogramming. It includes data structures and algorithms that closely resemble those from the C++ Standard Library, but here they are used at compile time. There is even support for compile-time lambda expressions! Performing compile-time operations, such as generating types or manipulating sequences of types, is increasingly common in modern C++, and a library that offers such functionality is an extremely important tool. To the best of my knowledge, there is nothing quite like the Mpl library in existence. It fills an important void in the world of C++ metaprogramming. I should tell you that there's a book for Boost.Mpl in the worksby the time you read this, it will be available. C++ Template Metaprogramming is written by Aleksey Gurtovoy and David Abrahams.

Math and Numerics

Boost.Integer

This library provides useful functionality for integer types, such as compile-time constants for the minimum and maximum values,[3] suitably sized types based on the number of required bits, static binary logarithm calculations, and more. Also included are typedefs from the 1999 C Standard header <stdint.h>.

[3] std::numeric_limits only provide these as functions.

The authors of Integer are Beman Dawes and Daryle Walker.

Boost.Interval

The Interval library helps when working with mathematical intervals. It provides arithmetic operators for the class template interval. A common use case for working with intervals (besides the obvious case of computations including intervals) is when computations provide inexact results; intervals make it possible to quantify the propagation of rounding errors.

The authors of Interval are Guillaume Melquiond, Sylvain Pion, and Hervé Brönniman, and the library is inspired by previous work from Jens Maurer.

Boost.Math

Math is a collection of mathematics templates: quaternions and octonions (generalizations of complex numbers); numerical functions such as acosh, asinh, and sinhc; functionality for calculating the greatest common divisor (GCD) and least common multiple (LCM); and more.

The authors of Math are Hubert Holin, Daryle Walker, and Eric Ford.

Boost.Minmax

Minmax simultaneously computes the minimum and maximum values, rather than requiring two comparisons when using std::min and std::max. For a range of n elements, only 3n/2+1 comparisons are performed, rather than the 2n required when using std::min_element and std::max_element.

The author of Minmax is Hervé Brönniman.

Boost.Numeric Conversion

The Numeric Conversion library is a collection of tools used to perform safe and predictable conversions between values of different numeric types. For example, there is a tool called numeric_cast (originally from Boost.Conversion), which performs range-checked conversions and ensures that the value can be represented in the destination type; otherwise, it throws an exception.

The author of Numeric Conversion is Fernando Cacciola.

Boost.Operators

The Operators library provides implementations of related operators and concepts (LessThanComparable, Arithmetic, and so on). When defining operators for a type, it is both tedious and error prone to add all of the operators that should be defined. For example, when providing operator< (LessThanComparable), operator<=, operator>, and operator>= should also be defined in most cases. Operators automatically declare and define all relevant operators in terms of a minimum set of user-defined operators for a given type. There is detailed coverage of the library in "Library 4: Operators 4."

Input/Output

Boost.Assign

Assign assists in assigning series of values into containers. It gives the user an easy way of assigning data, by means of overloaded operator, (the comma operator) and operator()() (function call operator). Although being especially useful for a prototyping-style of code, the functionality of the library is useful at other times too, due to the readable code that results from using the library. It is also possible to use this library to create anonymous arrays on-the-fly using list_of.

The author of Assign is Thorsten Ottosen.

Boost.Filesystem

The Filesystem library offers portable manipulation of paths, directories, and files. The high-level abstractions enable C++ programmers to write code similar to script-like operations that are often available in other programming languages. For iterating thorough directories and files, convenient algorithms are provided. The difficult task of writing code that is portable between platforms with different filesystems becomes feasible with the help of this library.

The author of Filesystem is Beman Dawes.

Boost.Format

This library adds functionality for formatting arguments according to format strings, similar to printf, but with the addition of type safety. One of the primary arguments against using printf and similar formatting facilities is that they are inherently dangerous; there is no assurance that the types that are specified in the format string are matched by the actual arguments. Besides eliminating the opportunity for such mismatches, Format also enables custom formatting of user-defined types.[4]

[4] This is not possible with formatting functions using a variable number of arguments through use of ellipsis.

The author of Format is Samuel Krempp.

Boost.Io_state_savers

The Io_state_savers library allows the state of IOStream objects to be saved, and later restored, to undo any intervening state changes that may occur. Many manipulators permanently change the state of the stream on which they operate, and it can be cumbersome at best and error prone at worst to manually reset the state. There are state savers for control flags, precision, width, exception masks, locale for the stream, and more.

The author of Io_state_savers is Daryle Walker.

Boost.Serialization

This library allows arbitrary C++ data structures to be saved to, and restored from, archives. An archive could be, for example, a text file or XML file. Boost.Serialization is highly portable and offers a very mature set of features, such as class versioning, serialization of common classes from the C++ Standard Library, serialization of shared data, and more.

The author of Serialization is Robert Ramey.

Miscellaneous

Boost.Conversion

The Conversion library contains functions that augment the existing cast operators (static_cast, const_cast, and dynamic_cast). Conversion adds polymorphic_cast and polymorphic_downcast for safe polymorphic casts, numeric_cast for safe conversions among numeric types, and lexical_cast for lexical conversions (for example, between string and double). You can customize these casts to work optimally with your own typessomething that isn't possible with the casts provided by the language. The library is covered in detail in "Library 2: Conversion."

The authors of Conversion are Dave Abrahams and Kevlin Henney.

Boost.Crc

The Crc library provides calculations of cyclic redundancy codes (CRC), a commonly used checksum type. A CRC is attached to a stream of data (from which it is computed), so the checksum can be used later to validate the data. The library includes four sample CRC types: crc_16_type, crc_ccitt_type, crc_xmodem_type, and crc_32_type5.

The author of Crc is Daryle Walker.

Boost.Date_time

The Date_time library provides extensive support for date and time types and operations upon them. Without library support for dates and time, temporal programming tasks are complicated and error prone. Using Date_time, the natural abstractions that one would expect are supported: days, weeks, months, durations (and intervals thereof), addition and subtraction, and so on. The library addresses issues commonly omitted from other date/time libraries, such as handling leap seconds and supporting high-resolution time sources. The library's design is extensible, allowing for customized behavior or added functionality.

The author of Date_time is Jeff Garland.

5. CRC32 is used in PKZip, for example.

Boost.Optional

It is common for functions to indicate that the returned value is invalid, but often the returned type does not have a state to indicate that it's not valid. Optional offers the class template optional, which is a type that semantically has an additional state, one that is in effect when instances of optional are not containing instances of the wrapped object.

The author of Optional is Fernando Cacciola.

Boost.Pool

The Pool library provides a pool memory allocatorthat is, a tool for managing dynamic memory in a single, large allocation. Using memory pools is a good solution when allocating and deallocating many small objects, or when memory control needs to be made more efficient.

The author of Pool is Steve Cleary.

Boost.Preprocessor

Using the preprocessor is hard when you need to express common constructs such as recursion, it doesn't have containers, doesn't provide means for iteration, and so forth. Nevertheless, the preprocessor is a powerful and portable tool. The Preprocessor library provides abstractions on top of the preprocessor. These include lists, tuples, and arrays, as well as algorithms that operate on the elements of those types. The library helps eliminate repetitive

Part I: General Libraries

It is not obvious what a suitable name for this part of the book should be. With a structure of the book that encompasses distinct domains (such as containers and higher-order programming), names are often palpable; except for what's covered in this partthose little things that we use all of the time: smart pointers, conversion utilities, and so on.

You can't really begin with a division called Miscellaneous, or Ubiquitous, or Frequently Used Libraries. It's truethey are all of these things, but it just doesn't convey their importance properly. Ergo, General Libraries, which I'm also hoping will focus on their omnipresence.

One thing that strikes me as odd is the way that we often regard these "simple" componentsutilities, if you likethat are of so much use to us. They get a lot of attention in books and articles, but it is surprisingly common to underestimate their value when it comes to selecting them (or creating them) for production code. Is it because we consider small components uncomplicated? Do we will ingly sacrifice flexibility on the basis that it's easy to create another small component just like it, but manually adapted to the exact problem at hand? If these are indeed the arguments, we are thoroughly deceiving ourselves. Two million instances of smart pointers in a program make the smart pointers critical, both in terms of efficiency and reliability. Twenty different implementations of common conversions in a program affects the time it takes to code them, but more importantly it also impedes maintainability. Systems are built on layers of abstraction, and the lower levels are often referred to as being comprised of data structures, algorithms, and utilities. If you agree with that, consider the impact of a change, or a bug, or unwarranted inflexibility in any of these small, insignificant, their-importance-forgotten utilities. Shiver. Utilities are vessels that traffic the veins of our programs. They are the oil in our engines of logic and the glue between our barriers of insulation. Enough of crummy analogies; let's just give them the credit they deserve, shall we? We will cover a wide variety of general libraries here, including smart pointers, conversions (both type conversions and lexical conversions), regular expressions, operators, static assertions, and more.

Library 1. Smart_ptr

How Does the Smart_ptr Library Improve Your Programs?

- Automatic lifetime management of objects with shared_ptr makes shared ownership of resources effective and safe.
- Safe observation of shared resources through weak_ptr avoids dangling pointers.
- Scoped resources using scoped_ptr and scoped_array make the code easier to write and maintain, and helps in writing exception-safe code.

Smart pointers solve the problem of managing the lifetime of resources (typically dynamically allocated objects[1]). Smart pointers come in different flavors. Most share one key featureautomatic resource management. This feature is manifested in different ways, such as lifetime control over dynamically allocated objects, and acquisition and release of resources (files, network connections). The Boost smart pointers primarily cover the first casethey store pointers to dynamically allocated objects, and delete those objects at the right time. You might wonder why these smart pointers don't do more. Couldn't they just as easily cover all types of resource management? Well, they could (and to some extent they do), but not without a price. General solutions often imply increased complexity, and with the Boost smart pointers, usability is of even higher priority than flexibility. However, through the support for custom deleters, Boost's arguably smartest smart pointer (boost::shared_ptr) supports resources that need other destruction code than delete. The five smart pointer types in Boost.Smart_ptr are tailor-made to fit the most common needs that arise in everyday programming.

[1] Just about any type of resource can be handled by a generic smart pointer type.

When Do We Need Smart Pointers?

There are three typical scenarios when smart pointers are appropriate:

- Shared ownership of resources
- When writing exception-safe code
- Avoiding common errors, such as resource leaks

Shared ownership is the case when two or more objects must use a third object. How (or rather when) should that third object be deallocated? To be sure that the timing of deallocation is right, every object referring to the shared resource would have to know about each other to be able to correctly time the release of that resource. That coupling is not viable from a design or a maintenance point of view. The better approach is for the owners to delegate responsibility for lifetime management to a smart pointer. When no more shared owners exist, the smart pointer can safely free the resource.

Exception safety at its simplest means not leaking resources and preserving program invariants when an exception is thrown. When an object is dynamically allocated, it won't be deleted when an exception is thrown. As the stack unwinds and the pointer goes out of scope, the resource is possibly lost until the program is terminated (and even resource reclamation upon termination isn't guaranteed by the language). Not only can the program run out of resources due to memory leaks, but the program state can easily become corrupt. Smart pointers can automatically release those resources for you, even in the face of exceptions.

Avoiding common errors. Forgetting to call delete is the oldest mistake in the book (at least in this book). A smart pointer doesn't care about the control paths in a program; it only cares about deleting a pointed-to object at the end of its lifetime. Using a smart pointer eliminates your need to keep track of when to delete objects. Also, smart pointers can hide the deallocation details, so that clients don't need to know whether to call delete, some special cleanup function, or not delete the resource at all.

Safe and efficient smart pointers are vital weapons in the programmer's arsenal. Although the C++ Standard Library offers std::auto_ptr, that's not nearly enough to fulfill our smart pointer needs. For example, auto_ptrs cannot be used as elements of STL containers. The Boost smart pointer classes fill a gap currently left open by the Standard.

The main focus of this chapter is on scoped_ptr, shared_ptr, intrusive_ptr, and weak_ptr. Although the complementary scoped_array and shared_array are sometimes useful, they are not used nearly as frequently, and they are so similar to those covered that it would be too repetitive to cover them at the same level of detail.

How Does Smart_ptr Fit with the Standard Library?

The Smart_ptr library has been proposed for inclusion in the Standard Library, and there are primarily three reasons for this:

- The Standard Library currently offers only auto_ptr, which is but one type of smart pointer, covering only one part of the smart pointer spectrum. shared_ptr offers different, arguably even more important, functionality.
- The Boost smart pointers are specifically designed to work well with, and be a natural extension to, the Standard Library. For example, before shared_ptr, there were no standard smart pointers that could be used as elements in containers.
- Real-world programmers have proven these smart pointer classes through heavy use in their own programs for a long time.

The preceding reasons make the Smart_ptr library a very useful addition to the C++ Standard Library. Boost.Smart_ptr's shared_ptr (and the accompanying helper enable_shared_from_this) and weak_ptr have been accepted for the upcoming Library Technical Report.

scoped_ptr

Header: "boost/scoped_ptr.hpp"

boost::scoped_ptr is used to ensure the proper deletion of a dynamically allocated object. scoped_ptr has similar characteristics to std::auto_ptr, with the important difference that it doesn't transfer ownership the way an auto_ptr does. In fact, a scoped_ptr cannot be copied or assigned at all! A scoped_ptr assumes ownership of the resource to which it points, and never accidentally surrenders that ownership. This property of scoped_ptr improves expressiveness in our code, as we can select the smart pointer (scoped_ptr or auto_ptr) that best fits our needs.

When deciding whether to use std::auto_ptr or boost::scoped_ptr, consider whether transfer of ownership is a desirable property of the smart pointer. If it isn't, use scoped_ptr. It is a lightweight smart pointer; using it doesn't make your program larger or run slower. It only makes your code safer and more maintainable.

Next is the synopsis for scoped_ptr, followed by a short description of the class members:

```
namespace boost {

  template<typename T> class scoped_ptr : noncopyable {
  public:
    explicit scoped_ptr(T* p = 0);
    ~scoped_ptr();

    void reset(T* p = 0);

    T& operator*() const;
    T* operator->() const;
    T* get() const;

    void swap(scoped_ptr& b);
  };

  template<typename T>
    void swap(scoped_ptr<T> & a, scoped_ptr<T> & b);
}
```

Members

```
explicit scoped_ptr(T* p=0)
```

The constructor stores a copy of p. Note that p must be allocated using operator new, or be null. There is no requirement on T to be a complete type at the time of construction. This is useful when the pointer p is the result of calling some allocation function rather than calling new directly: Because the type needn't be complete, a forward declaration of the type T is enough. This constructor never throws.

```
~scoped_ptr()
```

Deletes the pointee. The type T must be a complete type when it is destroyed. If the scoped_ptr holds no resource at the time of its destruction, this does nothing. The destructor never throws.

```
void reset(T* p=0);
```

Resetting a scoped_ptr deletes the stored pointer it already owns, if any, and then saves p. Often, the lifetime management of a resource is completely left to be handled by the scoped_ptr, but on rare occasions the resource needs to be freed prior to the scoped_ptr's destruction, or another resource needs to be handled by the scoped_ptr instead of the original. In those cases, reset is useful, but use it sparingly. (Excessive use probably indicates a design

scoped_array

Header: "boost/scoped_array.hpp"

The need for dynamically allocated arrays is usually best handled by std:: vector, but there are two cases when it makes good sense to use arrays: for optimization, as there is some overhead in size and speed for vector; and for expression of intent, making it clear that bounds are fixed.[5] Dynamically allocated arrays are exposed to the same dangers as ordinary pointers, with the added (and all too common) mistake of invoking the delete operator instead of the delete[] operator. I've seen that mistake in places one could hardly imagine, such as in widely used, proprietary container classes! scoped_array does for arrays what scoped_ptr does for pointers to single objects: It deletes the memory. The difference is that scoped_array does it using the delete[] operator.

[5] These are not clear-cut advantages. Indeed, it is usually best to use std::vector until performance measurements suggest the benefits of scoped_array are warranted.

The reason that scoped_array is a separate class rather than being a specialization of scoped_ptr is because it is not possible to distinguish between pointers to single objects and pointers to arrays using metaprogramming techniques. Despite efforts to make that distinction, no one has found a reliable way to do that because arrays decay so easily into pointers that carry no type information indicating that they point to arrays. As a result, the onus is on you to use scoped_array rather than scoped_ptr, just as you must otherwise choose to use the delete[] operator rather than the delete operator. The benefits are that scoped_array handles deletion for you, and that scoped_array conveys that we are dealing with an array, whereas a raw pointer doesn't.

scoped_array is very similar to scoped_ptr, with the differences that it provides operator[] to mimic a raw array.

scoped_array is a superior alternative to ordinary, dynamically allocated arrays. It handles lifetime management of dynamically allocated arrays, similar to how scoped_ptr manages lifetime for pointers to objects. Remember though, in most cases, std::vector is preferable as it is more flexible and powerful. When you need to clearly state that the size of the array is constant, use scoped_array rather than std::vector.

shared_ptr

Header: "boost/shared_ptr.hpp"

Almost all non-trivial programs need some form of reference-counted smart pointers. These smart pointers eliminate the need to write complicated logic to control the lifetime of objects shared among two or more other objects. When the reference count drops to zero, no more objects are interested in the shared object, and so it is deleted automatically. Reference-counted smart pointers can be categorized as intrusive or non-intrusive. The former expects the classes that it manages to provide certain functionality or data members with which to manage the reference count. That means designing classes with the foresight to work with an intrusive, reference-counted smart pointer class, or retrofitting. Non-intrusive, reference-counted smart pointers don't require anything of the types they manage. Reference-counted smart pointers assume ownership of the memory associated with their stored pointers. The problem with sharing objects without the help of smart pointers is that someone must, eventually, delete the shared memory. Who, and when? Without reference-counted smart pointers, one must impose lifetime management externally to the memory being managed, which typically means stronger dependencies among the collective owners. That, in turn, impedes reusability and adds complexity.

The class to be managed may have properties that make it a good candidate for use with a reference-counted smart pointer. For example, the fact that it is expensive to copy, or that part of its representation needs to be shared between instances, make shared ownership desirable. There are also situations in which there is no explicit owner of a shared resource. Using reference-counted smart pointers makes possible sharing ownership among the objects that need access to the shared resource. Reference-counted smart pointers also make it possible to store pointers to objects in Standard Library containers without risk of leaks, especially in the face of exceptions or when removing elements from the containers. When you store pointers in containers, you can take advantage of polymorphism, improved efficiency (if copying is expensive), and the ability to store the same objects in multiple, associated containers for specialized lookups.

After you've determined that the use of a reference-counted smart pointer is warranted, how do you choose whether to use an intrusive or non-intrusive design? Non-intrusive smart pointers are almost always the better choice on account of their general applicability, lack of impact on existing code, and flexibility. You can use non-intrusive, reference-counted smart pointers with classes that you cannot or don't wish to change. The usual way to adapt a class to work with an intrusive, reference-counted smart pointer is to derive from a reference-counted base class. That change may be more expensive than appears at first glance. At the very least, it adds dependencies and decreases reusability.[6] It also typically increases object size, which may limit usability in some contexts.[7]

[6] Consider the need to use more than one reference-counted smart pointer class with the same type. If both are intrusive designs, the different base classes may not be compatible and will certainly be wasteful. If only one is an intrusive design, the overhead of the base class is for naught when using the non-intrusive smart pointer.

[7] On the other hand, non-intrusive smart pointers require additional storage for the actual smart pointer.

A shared_ptr can be constructed from a raw pointer, another shared_ptr, a std::auto_ptr, or a boost::weak_ptr. It is also possible to pass a second argument to the constructor of shared_ptr, known as a deleter. The deleter is later called upon to handle deletion of the shared resource. This is useful for resource management where the resource is not allocated with new and destroyed with delete (we shall see examples of creating custom deleters later). After the shared_ptr has been constructed, it is used just like an ordinary pointer, with the obvious exception that it must not be explicitly deleted.

This is a partial synopsis for shared_ptr; the most important members and accompanying free functions are shown and subsequently briefly discussed.

```
namespace boost {

  template<typename T> class shared_ptr {
  public:
    template <class Y> explicit shared_ptr(Y* p);
    template <class Y,class D> shared_ptr(Y* p,D d);
```

shared_array

Header: "boost/shared_array.hpp"

shared_array is a smart pointer that enables shared ownership of arrays. It is to shared_ptr what scoped_array is to scoped_ptr. shared_array differs from shared_ptr mainly in that it is used with arrays rather than a single object. When we discussed scoped_array, I mentioned that std::vector was often a better choice. But shared_array adds some value over vector, because it offers shared ownership of arrays. The shared_array interface is similar to that of shared_ptr, but with the addition of a subscript operator and without support for custom deleters.

Because a shared_ptr to std::vector offers much more flexibility than shared_array, there's no usage section on shared_array in this chapter. If you find that you need boost::shared_array, refer to the online documentation.

intrusive_ptr

Header: "boost/intrusive_ptr.hpp"

intrusive_ptr is the intrusive analogue to shared_ptr. Sometimes, there's no other choice than using an intrusive, reference-counted smart pointer. The typical scenario is for code that has already been written with an internal reference counter, and where there's no time to rewrite it (or where the code's not available). Another case is when the size of a smart pointer must be exactly the size of a raw pointer, or when performance is hurt by the allocation of the reference count for shared_ptr (a rare case, I'm sure!). The only case where it would seem that an intrusive smart pointer is required, from a functional perspective, is when a member function of a pointed-to class needs to return this, such that it can be used in another smart pointer. (Actually, there are ways to solve that problem with non-intrusive smart pointers too, as we saw earlier in this chapter.) intrusive_ptr is different from the other smart pointers because it requires you to provide the reference counter that it manipulates.

When intrusive_ptr increments or decrements a reference count on a non-null pointer, it does so by making unqualified calls to the functions intrusive_ptr_add_ref and intrusive_ptr_release, respectively. These functions are responsible for making sure that the reference count is always correct and, if the reference counter drops to zero, to delete the pointer. Therefore, you must overload those functions for your type, as we shall see later.

This is a partial synopsis for intrusive_ptr, showing the most important functions.

```
namespace boost {

  template<class T> class intrusive_ptr {
  public:
    intrusive_ptr(T* p,bool add_ref=true);

    intrusive_ptr(const intrusive_ptr& r);

    ~intrusive_ptr();

    T& operator*() const;
    T* operator->() const;
    T* get() const;

    operator unspecified-bool-type() const;
  };

  template <class T> T* get_pointer(const intrusive_ptr<T>& p);

  template <class T,class U> intrusive_ptr<T>
    static_pointer_cast(const intrusive_ptr<U>& r);
}
```

Members

```
intrusive_ptr(T* p,bool add_ref=true);
```

This constructor stores the pointer p in *this. If p isn't null, and if add_ref is true, the constructor makes an unqualified call to intrusive_ptr_add_ref(p). If add_ref is false, the constructor makes no call to intrusive_ptr_add_ref. This constructor can throw an exception if intrusive_ptr_add_ref can throw.

```
intrusive_ptr(const intrusive_ptr& r);
```

The copy constructor saves a copy of r.get() and, if that pointer is not null, calls intrusive_ptr_add_ref with it. This constructor never throws.

```
~intrusive_ptr();
```

weak_ptr

Header: "boost/weak_ptr.hpp"

A weak_ptr is an observer of a shared_ptr. It does not interfere with the ownership of what a shared_ptr shares. When a shared_ptr that is being observed by a weak_ptr must release its resource, it sets the observing weak_ptr's pointer to null. That prevents the weak_ptr from holding a dangling pointer. Why would you need a weak_ptr? There are many situations where one needs to observe and use a shared resource without accepting ownership, such as to break cyclic dependencies, to observe a shared resource without assuming ownership of it, or to avoid dangling pointers. It's possible to construct a shared_ptr from a weak_ptr, thereby gaining access to the shared resource.

This is a partial synopsis for weak_ptr, showing and then briefly discussing the most important functions.

```
namespace boost {

  template<typename T> class weak_ptr {
  public:
    template <typename Y>
      weak_ptr(const shared_ptr<Y>& r);

    weak_ptr(const weak_ptr& r);

    ~weak_ptr();

    T* get() const;
    bool expired() const;
    shared_ptr<T> lock() const;
  };
}
```

Members

```
template <typename Y> weak_ptr(const shared_ptr<Y>& r);
```

This constructor creates a weak_ptr from a shared_ptr, provided there is an implicit conversion from Y* to T*. The new weak_ptr is configured to observe the resource referred to by r. r's reference count remains unchanged. This implies that the resource referenced by r may be deleted despite the existence of the new weak_ptr referring to it. This constructor never throws.

```
weak_ptr(const weak_ptr& r);
```

The copy constructor makes the new weak_ptr observe the resource referenced by shared_ptr r. The reference count of the shared_ptr is unchanged. This constructor never throws.

```
~weak_ptr();
```

The weak_ptr destructor, similarly to the constructor, does not change the reference count. If needed, the destructor detaches *this as an observer for the shared resource. This destructor never throws.

```
bool expired() const;
```

Returns TRue if the observed resource has "expired," which means that it has been released. If the stored pointer is non-null, expired returns false. This function never throws.

```
shared_ptr<T> lock() const
```

Smart_ptr **Summary**

This chapter has introduced the Boost smart pointers, a contribution to the C++ community that can hardly be overestimated. For a smart pointer library to be successful, it must take into consideration and correctly handle a great number of factors. I'm sure you have seen quite a number of smart pointers, and you might have even been involved in their creation, so you are aware of the effort involved to get things right. Not many smart pointers are as smart as they should be, and that makes the value of a proven library such as Boost.Smart_ptr immense.

Being such a central component of software engineering, the smart pointers in Boost have obviously received a lot of attention and thorough review. It is therefore hard to give credit to all who deserve it. Many have contributed valuable opinions and have been part of shaping the current smart pointer library. However, a few exceptional people and efforts must be mentioned here:

- Greg Colvin, the father of auto_ptr, also suggested counted_ptr, which later became what we now call shared_ptr.
- Beman Dawes revived the discussion about smart pointers and proposed that the original semantics as suggested by Greg Colvin be considered.
- Peter Dimov redesigned the smart pointer classes, adding thread safety, intrusive_ptr, and weak_ptr.

It is intriguing that such a well-known concept continues to evolve. There will undoubtedly be more progress in the domain of smart pointers or maybe, smart resources, but just as important is the quality of smart pointers that are used today. It's survival of the fittest, and that's why people are using Smart_ptr. The Boost smart pointers are a fine, assorted selection of delicious software chocolate, and I eat them regularly (you should, too). We'll soon see some of them become part of the C++ Standard Library, as they have been accepted into the Library Technical Report.

Endnotes

15. Boost.Bind is just such a library.

Library 2. Conversion

PREV NEXT

How Does the Conversion Library Improve Your Programs?

- Understandable, maintainable, and consistent polymorphic conversions
- Static downcasting using safer constructs than static_cast
- Range-preserving numeric conversions that ensure correct value logic and less time debugging
- Correct and reusable lexical conversions that lead to less time coding

The versatility of C++ is one of the primary reasons for its success, but sometimes also a formidable source of headaches because of the complexity of certain parts of the language. For instance, the rules for numeric conversions and type promotions are far from trivial. Other conversions are trivial, but tedious; how many times do we need to write a safe function[1] for converting between strings and ints, doubles and strings, and so on? Conversions can be problematic in every library and program you write, and that's how and why the Conversion library can help. It provides facilities that prevent dangerous conversions and simplify recurring conversion tasks.

[1] To avoid using sprintf and its ilk.

The Conversion library consists of four cast functions that provide better type safety (polymorphic_cast), better efficiency with preserved type safety (polymorphic_downcast), range-checked numeric conversions (numeric_cast), and lexical conversions (lexical_cast). These cast-like functions share the semantics of the C++ cast operators. Like the C++ cast operators, these functions have an important quality that, together with type safety, sets them apart from C-style casts: They unambiguously state the programmer's intent.[2] The importance of the code we write goes far further than its implementation and present behavior. More important is to clearly convey our intents when writing it. This library makes it somewhat easier by extending our C++ vocabulary.

[2] They can also be overloaded, which sometimes makes them superior to the C++ cast operators.

polymorphic_cast

Header: "boost/cast.hpp"

Polymorphic conversions in C++ are performed via dynamic_cast. A feature of dynamic_cast, which is sometimes also the cause of erroneous code, is that it behaves differently depending on the type with which it is used. dynamic_cast tHRows an exceptionstd::bad_castif the conversion is not possible when used on a reference type. The reason for the exception is simple. There is no such thing as a null reference in C++, so either the conversion succeeds and the result is a valid reference or it fails and you get an exception instead. Of course, when using dynamic_cast to convert a pointer type, failure is indicated by returning the null pointer.

dynamic_cast's different behavior depending on whether pointer or reference types are used is a valuable property, because it allows the programmer to express intent. Typically, if a failed conversion doesn't constitute a logical error, the pointer conversion is used, and if it is an error, the reference version is used. Unfortunately, the difference is quite subtleit boils down to an asterisk or an ampersandand it isn't always a natural choice. What if a failed cast to a pointer type is an error? To make that clear by having an exception thrown automatically, and to make the code consistent, Boost offers polymorphic_cast. It always throws a std::bad_cast exception if the conversion fails.

In The C++ Programming Language 3rd Edition, Stroustrup has the following to say about dynamic_cast with pointer types, and the fact that it can return the null pointer:

"Explicit tests against 0 can beand therefore occasionally will beaccidentally omitted. If that worries you, you can write a conversion function that throws an exception in case of failure."

polymorphic_cast is precisely that conversion function.

Usage

polymorphic_cast is used just like dynamic_cast, except (pun intended) that it always throws a std::bad_cast on failure to convert. Another feature of polymorphic_cast is that it is a function, and can be overloaded, if necessary. As a natural extension to our C++ vocabulary, it makes code clearer and casts less error prone. To use it, include the header "boost/cast.hpp". The function is parameterized on the type to convert to, and accepts one argument to be converted.

```
template <class Target, class Source>
  polymorphic_cast(Source* p);
```

It should be mentioned that there is no version of polymorphic_cast for reference types. The reason for this is that the implementation would do exactly what dynamic_cast already does, and there is no need for polymorphic_cast to duplicate existing functionality of the C++ language. The following example shows the syntactic similarity with dynamic_cast.

Downcast and Crosscast

There are two typical scenarios when using dynamic_cast or polymorphic_cast is appropriate: when downcasting from a base class to a derived class or when crosscasting, which means casting from one base class to another. The following example shows both types of casts using polymorphic_cast. There are two base classes, base1 and base2, and a class derived that inherits publicly from both of the base classes.

```
#include <iostream>
#include <string>
#include "boost/cast.hpp"

class base1 {
public:
  virtual void print() {
```

polymorphic_downcast

Header: "boost/cast.hpp"

Sometimes dynamic_cast is considered too inefficient (measured, I'm sure!). There is runtime overhead for performing dynamic_casts. To avoid that overhead, it is tempting to use static_cast, which doesn't have such performance implications. static_cast for downcasts can be dangerous and cause errors, but it is faster than dynamic_cast. If the extra speed is required, we must make sure that the downcasts are safe. Whereas dynamic_cast tests the downcasts and returns the null pointer or throws an exception on failure, static_cast just performs the necessary pointer arithmetic and leaves it up to the programmer to make sure that the conversion is valid. To be sure that static_cast is safe for downcasting, you must make sure to test every conversion that will be performed. polymorphic_downcast tests the cast with dynamic_cast, but only in debug builds; it then uses static_cast to perform the conversion. In release mode, only the static_cast is performed. The nature of the cast implies that you know it can't possibly fail, so there is no error handling, and no exception is ever thrown. So what happens if a polymorphic_downcast fails in a non-debug build? Undefined behavior. Your computer may melt. The Earth may stop spinning. You may float above the clouds. The only thing you can safely assume is that bad things will happen to your program. If a polymorphic_downcast fails in a debug build, it asserts on the null pointer result of dynamic_cast.

Before considering how to speed up a program by exchanging polymorphic_downcast for dynamic_cast, review the design. Optimizations on casts are likely indicators of a design problem. If the downcasts are indeed needed and proven to be performance bottlenecks, polymorphic_downcast is what you need. You can only find erroneous casts in testing, not production (release builds), and if you've ever had to listen to a screaming customer on the other end of the phone, you know that catching errors in testing is rather important and makes life a lot easier. Even more likely is that you've been the customer from time to time, and know firsthand how annoying it is to find and report someone else's problems. So, use polymorphic_downcast if needed, but tread carefully.

Usage

polymorphic_downcast is used in situations where you'd normally use dynamic_cast but don't because you're sure which conversions will take place, that they will all succeed, and that you need the improved performance it brings. Nota bene: Be sure to test all possible combinations of types and casts using polymorphic_downcast. If that's not possible, do not use polymorphic_downcast; use dynamic_cast instead. When you decide to go ahead and use polymorphic_downcast, include "boost/cast.hpp".

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

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

struct derived1 : public base {
  void foo() {
    std::cout << "derived1::foo()\n";
  }
};

struct derived2 : public base {
  void foo() {
    std::cout << "derived2::foo()\n";
  }
};

void older(base* p) {
  // Logic that suggests that p points to derived1 omitted
  derived1* pd=static_cast<derived1*>(p);
  pd->foo(); // <-- What will happen here?
}

void newer(base* p) {
```

numeric_cast

Header: "boost/cast.hpp"

Conversions between integral types can often produce unexpected results. For example, a long can typically hold a much greater range of values than a short, so what happens when assigning a long to a short and the long's value is outside of short's range? The answer is that the result is implementation defined (a nice term for "you can never know for sure"). Signed to unsigned conversions between same size integers are fine, so long as the signed value is positive, but what happens if the signed value is negative? It turns into a large unsigned value, which is indeed a problem if that was not the intention. numeric_cast helps ensure valid conversions by testing whether the range is preserved and by throwing an exception if it isn't.

7. The C++ Standard covers promotions and conversions for numeric types in §4.5-4.9.

Before we can fully appreciate numeric_cast, we must understand the rules that govern conversions and promotions of integral types. The rules are many and sometimes subtlethey can trap even the experienced programmer. Rather than stating all of the rules7 and then carry on, I'll give you examples of conversions that are subject to undefined or surprising behavior, and explain which rules the conversions adhere to.

When assigning to a variable from one of a different numeric type, a conversion occurs. This is perfectly safe when the destination type can hold any value that the source can, but is unsafe otherwise. For example, a char generally cannot hold the maximum value of an int, so when an assignment from int to char occurs, there is a good chance that the int value cannot be represented in the char. When the types differ in the range of values they can represent, we must make sure that the actual value to convert is in the valid range of the destination type. Otherwise, we enter the land of implementation-defined behavior; that's what happens when a value outside of the range of possible values is assigned to a numeric type.[8] Implementation-defined behavior means that the implementation is free to do whatever it wants to; different systems may well have totally different behavior. numeric_cast can ensure that the conversions are valid and legal or they will not be allowed.

[8] Unsigned arithmetic notwithstanding; it is well defined for these cases.

Usage

numeric_cast is a function template that looks like a C++ cast operator and is parameterized on both the destination and source types. The source type can be implicitly deduced from the function argument. To use numeric_cast, include the header "boost/cast.hpp". The following two conversions use numeric_cast to safely convert an int to a char, and a double to a float.

```
char c=boost::numeric_cast<char>(12);
float f=boost::numeric_cast<float>(3.001);
```

One of the most common numeric conversion problems is assigning a value from a type with a wider range than the one being assigned to. Let's see how numeric_cast can help.

Assignment from a Larger to a Smaller Type

When assigning a value from a larger type (for example, long) to a smaller type (for example, short), there is a chance that the value is too large or too small to be represented in the destination type. If this happens, the result is (yes, you've guessed it) implementation-defined. We'll talk about the potential problems with unsigned types later; let's just start with the signed types. There are four built-in signed integral types in C++:

- signed char
-

lexical_cast

Header: "boost/lexical_cast.hpp"

Lexical conversions are performed in virtually all applications. We convert strings to numeric values and vice versa. Many user-defined types can be converted to strings or created from strings. It is all too common to write the code for these conversions each time you need it, which suggests that it is very much suited for a reusable implementation. That's lexical_cast's purpose. Think of lexical_cast as using a std::stringstream as an interpreter between the string and other representation of a value. That means that it will work for any source with an appropriate output operator<< and any target with an appropriate operator<<. That's true for all of the built-in types and many user-defined types (UDTs).

Usage

lexical_cast makes a conversion between types look like any other type-converting cast. Of course, there must be a conversion function somewhere to make it work, but conceptually, it can be thought of as a cast. Rather than calling one of a number of conversion routines, or even coding the conversion locally, lexical_cast does that job for any types that meet its requirements. The source type must be OutputStreamable and the destination type must be InputStreamable. In addition, both types need to be CopyConstructible, and the target also DefaultConstructible and Assignable. OutputStreamable means that there's an operator<< defined for the type, and InputStreamable mandates an operator>>. This is true for many types, including the built-in types and the string classes from the Standard Library. To use lexical_cast, include "boost/lexical_cast.hpp".

Putting lexical_cast to Work

I won't bore you by producing conversion code manually to show how much code lexical_cast saves you, because I'm sure you've written these conversions yourself, and quite probably done so more than once. Instead, the example just uses lexical_cast for a number of common (lexical) type conversions.

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

int main() {
  // string to int
  std::string s="42";
  int i=boost::lexical_cast<int>(s);

  // float to string
  float f=3.14151;
  s=boost::lexical_cast<std::string>(f);

  // literal to double
  double d=boost::lexical_cast<double>("2.52");

  // Failed conversion
  s="Not an int";
  try {
    i=boost::lexical_cast<int>(s);
  }
  catch(boost::bad_lexical_cast& e) {
    // The lexical_cast above will fail,
    // and we'll end up here
  }
}
```

This example shows only a few of many scenarios where lexical conversion are performed, and I think you'll agree that it usually takes a few more lines of code than this to get the job done. Whenever there's uncertainty that the conversion is valid, the lexical_cast should be protected by a TRy/catch block, as you see in the preceding example.

Conversion Summary

In this chapter, you have learned about the Boost.Conversion library, starting with polymorphic_cast. The rationale for polymorphic_cast is code clarity and safetyclarity, because it gives us increased flexibility in stating our intent in code, and safety, because it's safer than its companion dynamic_cast<T*>, because tests of the resulting pointer are easily forgotten.

You then looked at safe optimizations, using polymorphic_downcast, which adds dynamic_cast-like safety in debug builds, but uses static_cast for the conversion. This makes it safer than static_cast alone.

numeric_cast helped with some of the thorny issues related to numeric conversions. Again, code clarity was improved and we stayed clear of both undefined and implementation-defined behavior.

Finally, there was lexical_cast. No more repetitive conversion functions. That's why it's been proposed for inclusion in the next revision of the C++ Standard Library. It is a tool that is very handy for converting different streamable data types.

If you were to read the implementation for these casts, you'd agree that none of them are very complicated. Still, it took insight, vision, and knowledge to recognize the need for them and to implement them correctly, portably, and efficiently. Not all people realize that there is something amiss when using dynamic_cast. Not many know the intricacies of integral type conversion and promotion. The Boost conversion "casts" include all of that knowledge and are well crafted and tested; they are excellent candidates for your use.

Library 3. Utility

PREV NEXT

How Does the Utility Library Improve Your Programs?

- Compile time assertions with BOOST_STATIC_ASSERT
- Safe destruction with checked_delete and checked_array_delete
- Prohibition of copying with noncopyable
- Retrieval of object addresses when operator& is overloaded through addressof
- Controlled participation of overloads and specializations with enable_if and disable_if

There are some utilities that just don't constitute a library in their own right, and are therefore grouped together with other entities. This is what Boost.Utility is, a collection of useful tools with no better home. They are useful enough to warrant inclusion in Boost, yet they are too small to deserve their own library. This chapter covers some of Boost.Utility's most fundamental and widely applicable tools.

We'll start with BOOST_STATIC_ASSERT, a facility for asserting integral constant expressions at compile time. Then, we'll see what happens when you delete an object through a pointer to an incomplete typethat is, when the layout of the object being destroyed is unknown. checked_delete makes that discussion more interesting. We'll also see how noncopyable prevents a class from ever being copied, which is arguably the most important topic of this chapter. Then, we'll check out addressof, which defeats the ill doings of menacing programmers[1] who overload operator&. Finally, we shall examine enable_if, which is really useful for controlling whether function overloads and template specializations are considered during name lookup or not.

[1] If you feel that I'm out of line here, please send me your most compelling use cases for overloading operator&.

BOOST_STATIC_ASSERT

Header: "boost/static_assert.hpp"

Performing assertions at runtime is something that you probably do regularly, and for good reasons. It is an excellent way of testing preconditions, postconditions, and invariants. There are many variations for performing runtime assertions, but how do you assert at compile time? Of course, the only way to do that is to have the compiler generate an error, and while that is quite trivial (I've inadvertently done it many thousand times), it's not obvious how to get meaningful information into the error message. Furthermore, even if you find a way on one compiler, it's a lot harder to do it portably. This is the rationale for BOOST_STATIC_ASSERT. It can be used at different scopes, as we shall see.

Usage

To start using static assertions, include the header "boost/static_assert.hpp". This header defines the macro[2] BOOST_STATIC_ASSERT. For the first demonstration of its usage, we'll see how it is used at class scope. Consider a parameterized class that requires that the types with which it is instantiated are of integral type. We'd rather not provide specializations for all of those types, so what we need is to assert, at compile time, that whatever type our class is being parameterized on is indeed an integral type. Now, we're going to get a little bit ahead of ourselves by using another Boost library for testing the typeBoost.Type_traits. We'll use a predicate called is_integral, which performs a compile time evaluation of its argument and, as you might guess from its name, indicates whether that type is an integral type.

[2] Yes, it's a macro. They too can be useful, you know.

```
#include <iostream>

#include "boost/type_traits.hpp"
#include "boost/static_assert.hpp"

template <typename T> class only_compatible_with_integral_types {
  BOOST_STATIC_ASSERT(boost::is_integral<T>::value);
};
```

With this assertion, trying to instantiate the class only_compatible_with_integral_types with a type that is not an integral type causes a failure at compile time. The output depends on the compiler, but it is surprisingly consistent on most compilers.

Suppose we tried to instantiate the class like this:

```
only_compatible_with_integral_types<double> test2;
```

The compiler output will look something like this:

```
Error: use of undefined type
  'boost::STATIC_ASSERTION_FAILURE<false>'
```

At class scope, you can ensure certain requirements for the class: For a template like this, the parameterizing type is an obvious example. You could also use assertions for other assumptions that the class makes, such as the size of certain types and such.

BOOST_STATIC_ASSERT at Function Scope

BOOST_STATIC_ASSERT can also be used at function scope. For example, consider a function that is parameterized on a non-type template parameterlet's assume an intand the parameter can accept values between 1

checked_delete

Header: "boost/checked_delete.hpp"

When deleting an object through a pointer, the result is typically dependent on whether the type being deleted is known at the time of the deletion. There are hardly ever compiler warnings when delete-ing a pointer to an incomplete type, but it can cause all kinds of trouble, because the destructor may not be invoked. This, in turn, means that cleanup code won't be performed. checked_delete is in effect a static assertion that the class type is known upon destruction, enforcing the constraint that the destructor will be called.

Usage

checked_delete is a template function residing in the boost namespace. It is used for deleting dynamically allocated objectsand there's a companion used for dynamically allocated arrays called checked_array_delete. The functions accept one argument; the pointer or array to be deleted. Both of these functions require that the types they delete be known at the time they are destroyed (that is, when they are passed to the functions). To use the functions, include the header "boost/checked_delete.hpp". When utilizing the functions, simply call them where you would otherwise call delete. The following program forward declares a class, some_class, that is never defined. Any compiler would allow a pointer to some_class to be deleted (more on this later), but checked_delete does not compile until a definition of some_class is available.

```
#include "boost/checked_delete.hpp"

class some_class;

some_class* create() {
  return (some_class*)0;
}

int main() {
  some_class* p=create();
  boost::checked_delete(p2);
}
```

When trying to compile this program, the instantiation of the function checked_delete<some_class> fails because some_class is an incomplete type. Your compiler will say something like this:

```
checked_delete.hpp: In function 'void
boost::checked_delete(T*) [with T = some_class]':
checked_sample.cpp:11:   instantiated from here
boost/checked_delete.hpp:34: error: invalid application of 'sizeof' to an incomplete
type
boost/checked_delete.hpp:34: error: creating array with
size zero ('-1')
boost/checked_delete.hpp:35: error: invalid application of
'sizeof' to an incomplete type
boost/checked_delete.hpp:35: error: creating array with
size zero ('-1')
boost/checked_delete.hpp:32: warning: 'x' has incomplete type
```

The first part of the preceding error message clearly spells out the problem: that checked_delete has encountered an incomplete type. But when and how are incomplete types problems in our code? The following section talks about exactly that.

What's the Problem, Anyway?

Before we really start enjoying the benefits of checked_delete, let's make sure that we understand the problem in full. If you try to delete a pointer to an incomplete type[3] with a non-trivial destructor [4] the result is undefined behavior

noncopyable

Header: "boost/utility.hpp"

The compiler is often a very good friend of the programmer, but not always. One example of its friendliness is the way that it automatically provides copy construction and assignment for our classes, should we decide not to do so ourselves. This can lead to some unpleasant surprises, if the class isn't meant to be copied (or assigned to) in the first place. When that's the case, we need to tell clients of this class explicitly that copy construction and assignment are prohibited. I'm not talking about comments in the code, but about denying access to the copy constructor and copy assignment operator. Fortunately, the compiler-generated copy constructor and copy assignment operator are not usable when the class has bases or data members that aren't copyable or assignable. boost::noncopyable works by prohibiting access to its copy constructor and assignment operator and then being used as a base class.

Usage

To make use of boost::noncopyable, have the noncopyable classes derive privately from it. Although public inheritance works, too, this is a bad practice. Public inheritance says IS-A (denoting that the derived class also IS-A base) to people reading the class declaration, but stating that a class IS-A noncopyable seems a bit far fetched. Include "boost/utility.hpp" when deriving from noncopyable.

```
#include "boost/utility.hpp"

class please_dont_make_copies : boost::noncopyable {};

int main() {
  please_dont_make_copies d1;
  please_dont_make_copies d2(d1);
  please_dont_make_copies d3;
  d3=d1;
  }
```

The preceding example does not compile. The attempted copy construction of d2 fails because the copy constructor of noncopyable is private. The attempted assignment of d1 to d3 fails because the copy assignment operator of noncopyable is private. The compiler should give you something similar to the following output:

```
noncopyable.hpp: In copy constructor
' please_dont_make_copies::please_dont_make_copies (const please_dont_make_copies&)':
boost/noncopyable.hpp:27: error: '
  boost::noncopyable::noncopyable(const boost::noncopyable&)' is
private
noncopyable.cpp:8: error: within this context
boost/noncopyable.hpp: In member function 'please_dont_make_copies&
  please_dont_make_copies::operator=(const please_dont_make_copies&)':
boost/noncopyable.hpp:28: error: 'const boost::noncopyable&
  boost::noncopyable::operator=(const boost::noncopyable&)' is private
noncopyable.cpp:10: error: within this context
```

We'll examine how this works in the following sections. It's clear that copying an assignment is prohibited when deriving from noncopyable. This can also be achieved by defining the copy constructor and copy assignment operator privatelylet's see how to do that.

Making Classes Noncopyable

Consider again the class please_dont_make_copies, which, for some reason, should never be copied.

```
class please_dont_make_copies {
public:
  void do_stuff() {
```

addressof

Header: "boost/utility.hpp"

When taking the address of a variable, we typically depend on the returned value to be, well, the address of the variable. However, it's technically possible to overload operator&, which means that evildoers may be on a mission to wreak havoc on your address-dependent code. boost::addressof is provided to get the address anyway, regardless of potential uses and misuses of operator overloading. By using some clever internal machinery, the template function addressof ensures that it gets to the actual object and its address.

Usage

To always be sure to get the real address of an object, use boost::addressof. It is defined in "boost/utility.hpp". It is used where operator& would otherwise be used, and it accepts an argument that is a reference to the type whose address should be taken.

```
#include "boost/utility.hpp"

class some_class {};

int main() {
  some_class s;
  some_class* p=boost::addressof(s);
}
```

Before seeing more details on how to use addressof, it is helpful to understand why and how operator& may not actually return the address of an object.

Quick Lesson for Evildoers

If you really, really, really need to overload operator&, or just want to experiment with the potential uses of operator overloading, it's actually quite easy. When overloading operator&, the semantics are always different from what most users (and functions!) expect, so don't do it just to be cute; do it for a very good reason or not at all. That said, here's a code-breaker for you:

```
class codebreaker {
public:
  int operator&() const {
    return 13;
  }
};
```

With this class, anyone who tries to take the address of an instance of codebreaker is handed the magical number 13.

```
template <typename T> void print_address(const T& t) {
  std::cout << "Address: " << (&t) << '\n';
}

int main() {
  codebreaker c;
  print_address(c);
}
```

It's not hard to do this, but are there good arguments for ever doing it in real code? Probably not, because it cannot be made safe except when using local classes. The reason for this is that while it is legal to take the address of an incomplete type, it is undefined behavior to do so on an incomplete class with a user-defined operator&. Because we cannot guarantee that this won't happen, we're better off not overloading operator&.

enable_if

Header: "boost/utility/enable_if.hpp"

Sometimes, we wish to control whether a certain function, or class template specialization, can take part in the set of available overloads/specializations for overload resolution. For example, consider an overloaded function where one version is an ordinary function taking an int argument, and the other is a templated version that requires that the argument of type T has a nested type called type. They might look like this:

```
void some_func(int i) {
  std::cout << "void some_func(" << i << ")\n";
}

template <typename T> void some_func(T t) {
  typename T::type variable_of_nested_type;
  std::cout <<
    "template <typename T> void some_func(" << t << ")\n";
}
```

Now, imagine what happens when you call some_func somewhere in your code. If the type of the argument is int, the first version is called. Assuming that the type is something other than int, the second (templated) version is called.

This is fine, as long as that type has a nested type named type, but if it doesn't, this code does not compile. Is this really a problem? Well, consider what happens when another integral type is used, like short, or char, or unsigned long.

```
#include <iostream>

void some_func(int i) {
  std::cout << "void some_func(" << i << ")\n";
}

template <typename T> void some_func(T t) {
  typename T::type variable_of_nested_type;
  std::cout <<
    "template <typename T> void some_func(" << t << ")\n";
}

int main() {
  int i=12;
  short s=12;

  some_func(i);
  some_func(s);
}
```

When compiling this program, you will get something like the following output from the frustrated compiler:

```
enable_if_sample1.cpp: In function 'void some_func(T)
  [with T = short int]':
enable_if_sample1.cpp:17:   instantiated from here
enable_if_sample1.cpp:8: error:
  'short int' is not a class, struct, or union type

Compilation exited abnormally with code 1 at Sat Mar 06 14:30:08
```

There it is. The template version of some_func has been chosen as the best overload, but the code in that version is not valid for the type short. How could we have avoided this? Well, we would have liked to only enable the template version of some_func for types with a nested type named type, and to ignore it for those without it. We can do that.

Utility Summary

This chapter has demonstrated some useful utility classes that can greatly simplify our daily life. BOOST_STATIC_ASSERT asserts at compile time, which is very helpful both for testing preconditions and enforcing other requirements. For generic programming, checked_delete is extremely helpful in detecting erroneous usage, which in turn can save a lot of time reading terribly verbose error messages and studying code that seems just fine. We have also covered addressof, which is a handy tool for getting to the real address of an object, regardless of what operator& says. We also saw how enable_if and disable_if can control which functions participate in overload resolution and learned what SFINAE means!

We talked about the base class noncopyable. By providing both a useful idiom and straightforward usage that catches the eye of anyone reading the code, it definitely deserves to be used regularly. The omission of a copy constructor and assignment operator in classes that need them, whether through the need for customized copying/assignment or the prohibition thereof, is all too common in code, costing lots of frustration, time, and money.

This is one of the shortest chapters in the book, and I suspect that you've read through it fairly quickly. It pays you back fast, too, if you start using these utilities right away. There are other utilities in Boost.Utility, which I haven't covered here. You might want to surf over to the Boost Web site and have a look at the online documentation to see what other handy tools there would suit you well in your current work.

Library 4. Operators

How Does the Operators Library Improve Your Programs?

- Provides a complete set of comparison operators
- Provides a complete set of arithmetic operators
- Provides a complete set of operators for iterators

Among the operators defined in C++, there are a number of related sets. When you encounter a class with one operator from one of these sets, you typically expect to find the others, too. For instance, when a class provides operator==, you expect to find operator!= and probably operator<, operator<=, operator>, and operator>=. Sometimes, a class only provides operator< in order to define an ordering so objects of that class can be used in associative containers, but that often leaves class users wanting more. Likewise, a class with value semantics that provides operator+ but not operator+= or operator- is limiting its potential uses. When you define one operator from a set for your class, you should typically provide the remaining operators from that set to avoid surprises. Unfortunately, it is cumbersome and error prone to augment a class with the many operators needed to support comparisons or arithmetic, and iterator classes must provide certain sets of operators according to the iterator category they model just to function correctly.

Besides the tedium of defining the number of operators needed, their semantics must be correct to meet users' expectations. Otherwise, the class is, for all practical purposes, unusable. We can relieve ourselves from doing it all by hand, though. As you know, some of the operators are typically implemented in terms of others, such as implementing operator+ in terms of operator+=, and that suggests that some automation of this task is possible. In fact, that is the purpose of Boost.Operators. By allowing you to define only a subset of the required comparison or arithmetic operators, and then defining the rest for you based upon those you provide, Boost.Operators enforces the correct operator semantics, and reduces your chance of making mistakes.

An additional value of the Operators library is the explicit naming of concepts that apply for different operations, such as addable for classes supporting operator+ and operator+=, shiftable for classes supporting operator<< and operator>>, and so on. This is important for two reasons: A consistent naming scheme aids understanding; and these concepts, and the classes named after them, can be part of class interfaces, clearly documenting important behaviors.

How Does Operators Fit with the Standard Library?

When using the Standard Library containers and algorithms, one typically supplies at least some relational operators (most commonly operator<) to enable sorting, and thus also storage of the type in sorted, associative containers. A common practice is to define only the bare minimum of the required operators, which has the unfortunate side effect of making the class less complete, and harder to understand. On the other hand, when defining a full set of operators, there is a risk of introducing defective semantics. In these cases, the Operators library helps to make sure that the classes behave correctly, and adhere to the requirements of both the Standard Library and the users of the type. Finally, for types that define arithmetic operators, there are a number of operators that are well suited to be implemented in terms of other operators, and Boost.Operators is of great use here, too.

Operators

Header: "boost/operators.hpp"

There are a number of base classes that comprise the Operators library. Each class contributes operators according to the concept it names. You use them by inheriting from themmultiply inheriting if you need the services of more than one. Fortunately, there are some composite concepts defined in Operators obviating the need to multiply inherit for many common cases. The following synopses describe some of the most commonly used Operator classes, the concepts they represent, and the demands they place on classes derived from them. In some cases, the requirements for the actual concepts are not the same as the requirements for the concept base classes when using Operators. For example, the concept addable requires that there be an operator T operator+(const T& lhs,const T& rhs) defined, but the Operators base class addable instead requires a member function, T operator+=(const T& other). Using this member function, the base class addable augments the derived class with operator+. THRoughout the synopses, the concepts are always stated first, followed by the type requirements for classes deriving from them. Rather than repeating all of the concepts in this library, I have selected a few important ones; you'll find the full reference at www.boost.org, of course.

less_than_comparable

The less_than_comparable concept requires the following semantics for a type T.

```
bool operator<(const T&,const T&);
bool operator>(const T&,const T&);
bool operator<=(const T&,const T&);
bool operator>=(const T&,const T&);
```

When deriving from boost::less_than_comparable, the derived class (T) must provide the equivalent of

```
bool operator<(const T&, const T&);
```

Note that the return type need not be exactly bool, but it must be implicitly convertible to bool. For the concept LessThanComparable found in the C++ Standard, operator< is required, so classes derived from less_than_comparable need to comply with that requirement. In return, less_than_comparable implements the three remaining operators in terms of operator<.

equality_comparable

The equality_comparable concept requires the following semantics for a type T.

```
bool operator==(const T&,const T&);
bool operator!=(const T&,const T&);
```

When deriving from boost::equality_comparable, the derived class (T) must provide the equivalent of

```
bool operator==(const T&,const T&);
```

Again, the return type needn't be bool, but it must be a type implicitly convertible to bool. For the concept EqualityComparable in the C++ Standard, operator== is required, so derived classes from equality_comparable need to comply with that requirement. The class equality_comparable equips T with bool operator!=(const T&,const T&).

addable

The addable concept requires the following semantics for a type T.

Usage

To start using the Operators library, implement the applicable operator(s) for your class, include "boost/operators.hpp", and derive from one or more of the Operator base classes (they have the same names as the concepts they help implement), which all reside in namespace boost. Note that the inheritance doesn't have to be public; private inheritance works just as well. In this usage section, we look at several examples of using the different concepts, and also take a good look at how arithmetic and relational operators work, both in C++ and conceptually. For the first example of usage, we'll define a class, some_class, with an operator<. We decide that the equivalence relation implied by operator< should be made available through operator==. This can be accomplished by inheriting from boost::equivalent.

```
#include <iostream>
#include "boost/operators.hpp"

class some_class : boost::equivalent<some_class> {
  int value_;
public:
  some_class(int value) : value_(value) {}

  bool less_than(const some_class& other) const {
    return value_<other.value_;
  }
};

bool operator<(const some_class& lhs, const some_class& rhs) {
  return lhs.less_than(rhs);
}

int main() {
  some_class s1(12);
  some_class s2(11);

  if (s1==s2)
    std::cout << "s1==s2\n";
  else
    std::cout << "s1!=s2\n";
}
```

The operator< is implemented in terms of the member function less_than. The requirement for the equivalent base class is that operator< be present for the class in question. When deriving from equivalent, we pass the derived classthat is, some_classas a template parameter. In main, the operator== that is graciously implemented for us by the Operators library is used. Next, we'll take a look at operator< again, and see what other relations can be expressed in terms of less than.

Supporting Comparison Operators

A relational operator that we commonly implement is less thanthat is, operator<. We do so to support storage in associative containers and sorting. However, it is exceedingly common to supply only that operator, which can be confusing to users of the class. For example, most people know that negating the result of operator< yields operator>=.[1] Less than can also be used to calculate greater than, and so on. So, clients of a class supporting the less than relation have good cause for expecting that the operators that must also (at least implicitly) be supported are also part of the class interface. Alas, if we just add the support for operator< and omit the others, the class isn't as usable as it could, and should, be. Here's a class that's been made compliant with the sorting routines of the Standard Library containers.

[1] Although too many seem to think that it yields operator>!

```
class thing {
  std::string name_;
public:
```

Operators Summary

Providing the correct set of relational and arithmetic operators for user-defined classes is vital and provides significant challenges to get right. With the use of the Operators library, this task is greatly simplified, and correctness and symmetry come almost for free. In addition to the help that the library offers in defining the full sets of operators, the naming and definitions of the concepts that a class can support is made explicit in the definition of the class (and by the Operators library!). In this chapter, we have seen several examples of how using this library improves programming with operators by simplification and ensured correctness. It is a sad fact that providing important relational and arithmetic operators for user-defined types is often overlooked, and part of the reason is that there is so much work involved to get it right. This is no longer the case, and Boost.Operators is the reason why.

An important consideration when providing relational and arithmetic operators is to make sure that they are warranted in the first place. When there is an ordering relation between types, or for numeric types, this is always the case, but for other types of classes, operators may not convey intent clearly. Operators are almost always syntactic sugar, and the importance of syntactic sugar must never be underestimated. Unfortunately, operators are also seductive. Use them wisely, for they wield vast power. When you choose to add operators to a class, the Boost.Operators library increases the quality and efficiency of your work. The conclusion is that you should augment your classes with operators only after careful thought, and use the Operators library whenever you get the chance!

The Operators library is the result of contributions from several people. It was started by David Abrahams, and has since received valuable additions from Jeremy Siek, Aleksey Gurtovoy, Beman Dawes, and Daryle Walker. As is the case for most Boost libraries, innumerable other people have been involved in making this library what it is today.

Library 5. Regex

PREV NEXT

How Does the Regex Library Improve Your Programs?

- Brings support for regular expressions to C++
- Improves the robustness of input validation

Regular expressions are very often used in text processing. For example, there are a number of validation tasks that are suitable for regular expressions. Consider an application that requires the input to consist only of numbers. Another program might require a specific format, such as three digits, followed by a character, then two more digits. You could validate ZIP Codes, credit card numbers, Social Security numbers, or just about anything else; and using regular expressions to do the validation is straightforward. Another typical area where regular expressions excel are text substitutionsthat is, replacing some text with other text. Suppose you need to change the spelling of the word colour to color throughout a number of documents. Again, regular expressions provide the best means to do thatincluding remembering to make the changes also for Colour and COLOUR, and for the plural form colours, the verb colourize, and so forth. Yet another use case for regular expressions is in formatting of text.

Many popular programming languagesPerl is a prime examplehave built-in support for regular expressions, but that's not the case with C++. Also, the C++ Standard is silent when it comes to regexes. Boost.Regex is a very complete and effective library for incorporating regular expressions in C++ programs, and it even includes several different syntaxes that are used in widespread tools such as Perl, grep, and Emacs. It is one of the most renowned C++ libraries for working with regular expressions, and is both easy to use and incredibly powerful.

How Does Regex Fit with the Standard Library?

There is currently no support for regular expressions in the C++ Standard Library. This is unfortunate, as there are numerous uses for regular expressions, and users are sometimes deterred from using C++ for writing applications that need support for regular expressions. Boost.Regex fills that void in the standard, and it has been proposed for inclusion in a future version of the C++ Standard. Boost.Regex has been accepted for the upcoming Library Technical Report.

Regex

Header: "boost/regex.hpp"

A regular expression is encapsulated in an object of type basic_regex. We will look closer at the options for how regular expressions are compiled and parsed in subsequent sections, but let's first take a cursory look at basic_regex and the three important algorithms that are the bulk of this library.

```
namespace boost {
  template <class charT,
            class traits=regex_traits<charT> >
  class basic_regex {
  public:
    explicit basic_regex(
      const charT* p,
      flag_type f=regex_constants::normal);

    bool empty() const;

    unsigned mark_count() const;

    flag_type flags() const;
  };

  typedef basic_regex<char> regex;
  typedef basic_regex<wchar_t> wregex;
}
```

Members

```
explicit basic_regex (
  const charT* p,
  flag_type f=regex_constants::normal);
```

This constructor accepts a character sequence that contains the regular expression, and an argument denoting which options to use for the regular expressionfor example, whether it should ignore case. If the regular expression in p isn't valid, an exception of type bad_expression, or regex_error, is thrown. Note that these two exceptions mean the same thing; at the time of this writing, the change from the current name bad_expression has not yet been made, but the next version of Boost.Regex will change it to regex_error.

```
bool empty() const;
```

This member is a predicate that returns true if the instance of basic_regex does not contain a valid regular expressionthat is, it has been assigned an empty character sequence.

```
unsigned mark_count() const;
```

mark_count returns the number of marked subexpressions in the regex. A marked subexpression is a part of the regular expression enclosed within parentheses. The text that matches a subexpression can be retrieved after calling one of the regular expression algorithms.

```
flag_type flags() const;
```

Returns a bitmask containing the option flags that are set for this basic_regex. Examples of flags are icase, which means that the regular expression is ignoring case, and JavaScript, indicating that the syntax for the regex is the one used in JavaScript.

Usage

To begin using Boost.Regex, you need to include the header "boost/regex.hpp". Regex is one of the two libraries (the other one is Boost.Signals) covered in this book that need to be separately compiled. You'll be glad to know that after you've built Boostthis is a one-liner from the command promptlinking is automatic (for Windows-based compilers anyway), so you're relieved from the tedium of figuring out which lib file to use.

The first thing you need to do is to declare a variable of type basic_regex. This is one of the core classes in the library, and it's the one that stores the regular expression. Creating one is simple; just pass a string to the constructor containing the regular expression you want to use.

```
boost::regex reg("(A.*)");
```

This regular expression contains three interesting features of regular expressions. The first is the enclosing of a subexpression within parenthesesthis makes it possible to refer to that subexpression later on in the same regular expression or to extract the text that matches it. We'll talk about this in detail later on, so don't worry if you don't yet see how that's useful. The second feature is the wildcard character, the dot. The wildcard has a very special meaning in regular expressions; it matches any character. Finally, the expression uses a repeat, *, called the Kleene star, which means that the preceding expression may match zero or more times. This regular expression is ready to be used in one of the algorithms, like so:

```
bool b=boost::regex_match(
  "This expression could match from A and beyond.",
  reg);
```

As you can see, you pass the regular expression and the string to be parsed to the algorithm regex_match. The result of calling the function is true if there is an exact match for the regular expression; otherwise, it is false. In this case, the result is false, because regex_match only returns true when all of the input data is successfully matched by the regular expression. Do you see why that's not the case for this code? Look again at the regular expression. The first character is a capital A, so that's obviously the first character that could ever match the expression. So, a part of the input"A and beyond."does match the expression, but it does not exhaust the input. Let's try another input string.

```
bool b=boost::regex_match(
  "As this string starts with A, does it match? ",
  reg);
```

This time, regex_match returns true. When the regular expression engine matches the A, it then goes on to see what should follow. In our regex, A is followed by the wildcard, to which we have applied the Kleene star, meaning that any character is matching any number of times. Thus, the parsing starts to consume the rest of the input string, and matches all the rest of the input.

Next, let's see how we can put regexes and regex_match to work with data validation.

Validating Input

A common scenario where regular expressions are used is in validating the format of input data. Applications often require that input adhere to a certain structure. Consider an application that accepts input that must come in the form "3 digits, a word, any character, 2 digits or the string "N/A," a space, then the first word again." Coding such validations manually is both tedious and error prone, and furthermore, these formats are typically exposed to changing requirements; before you know it, some variation of the format needs to be supported, and your carefully crafted parser suddenly needs to be changed and debugged. Let's assemble a regular expression that can validate such input correctly. First, we need an expression that matches exactly 3 digits. There's a special shortcut for digits, \d, that we'll use. To have it repeated 3 times, there's a special kind of repeat called the bounds operator, which encloses the bounds in curly braces. Putting these two together, here's the first part of our regular expression.

Regex Summary

That regular expressions are useful and important is not disputed, and this library brings terrific regex power to C++. Traditionally, users have had few choices besides using the POSIX C APIs for regular expressions. For text-processing validation tasks, regular expressions are much more scalable and reliable than handcrafted parsers. For searching and replacing, there are a number of problems that are very elegantly solved using regular expressions, but virtually impossible to solve without them.

Boost.Regex is a powerful library so it has not been possible to cover all of it in this chapter. Similarly, the great expressiveness and range of application of regular expressions necessarily means that this chapter offers little more than an introduction to them. These topics could easily fill a separate book. To learn more, study the online documentation for Boost.Regex and pick up a book on regular expressions (consult the Bibliography for suggestions). Despite the power of Boost.Regex, and the breadth and depth of regular expressions, even complete neophytes can use regular expressions effectively with this library. For programmers who have selected other programming languages due to C++'s lack of support for regular expressions, welcome home.

Boost.Regex is not the only regular expression library available for C++ programmers, but it is certainly one of the best. It's easy to use and fast as lightning when matching your regular expressions. Use it as often as you can.

The author of Boost.Regex is Dr. John Maddock.

Part II: Containers and Data Structures

This part of the book covers the libraries Boost.Any, Boost.Variant, and Boost.Tuple. They are all containers in some sense, although they have virtually nothing in common with the Standard Library container types. These are all extremely useful libraries, which many others and I use to solve programming problems most every day. The problems they solve are not really covered by either C++ or the C++ Standard Library, and they are thus very important additions to our library toolbox. It's interesting to ponder how much the availability of basic data structures affect how we program, and even how we design. Without existing structures, we craft our own, and typically do so with significant consideration for the solution domain, which limits the reusability of our work. That's a common theme for all types of programming, of course, and the tradeoff is between genericity and basically just getting the job done. The value of flexible libraries that addresses both the issues we have at hand, and most issues we are likely to encounter at a later time, is substantial. These libraries also extend our C++ vocabulary in some sense, and the more users the libraries have, the larger the community that speaks these words. I am convinced that each of the libraries in this chapter deserves a place in every C++ professional's toolbox.

Library 6. Any

PREV NEXT

How Does the Any Library Improve Your Programs?

- Typesafe storage and safe retrieval of arbitrary types
- A means to store heterogeneous types in Standard Library containers
- Types are being passed through layers that need not know anything about the types

The Any library provides a type, any, that allows for storage of any type for later retrieval without loss of type safety. It is like a variant type on steroids: It will hold any type, but you have to know the type to retrieve the value. There are times when you need to store unrelated types in the same container. There are times when certain code only cares about conveying data from one point to another without caring about the data's type. At face value, it is easy to do those things. They can be done with an indiscriminate type such as void*. They can be done using a discriminated union. There are numerous variant types available that rely on some type tag mechanism. Unfortunately, all of these suffer from a lack of type safety, and only in the most controlled situations should we ever purposely defeat the type system. The Standard Library containers are parameterized on the type they contain, which poses a seemingly impossible challenge for storing elements of heterogeneous types in them. Fortunately, the cure doesn't have to be spelled void*, because the Any library allows you to store objects of different types for later retrieval. There is no way to get to the contained value without knowing its exact type, and thus, type safety is preserved.

When designing frameworks, it isn't possible to know in advance about the types that will be used together with the framework classes. A common approach is to require the clients of the framework to adapt a certain interface, or inherit from base classes provided by the framework. This is reasonable, because the framework probably needs to communicate with various higher-level classes in order to be useful. There are, however, situations where the framework stores or otherwise accepts types that it doesn't need to (or can) know anything about. Rather than violating the type system and go with the void* approach, the framework can use any.

How Does Any Fit with the Standard Library?

One important property of Any is that it provides the capability to store objects of heterogeneous types in Standard Library containers. It is also a sort of variant data type, which is something sorely needed, and currently missing, in the C++ Standard Library.

Any

Header: "boost/any.hpp"

The class any allows typesafe storage and retrieval of arbitrary types. Unlike indiscriminate types, any preserves the type, and actually does not let you near the stored value without knowing the correct type. Of course, there are means for querying for the type, and testing alternatives for the contained value, but in the end, the caller must know the exact type of the value in an any object, or any denies access. Think of any as a locked safe. Without the proper key, you cannot get in. any requires the following of the types it stores:

- CopyConstructible It must be possible to copy the type.
- Non-throwing destructor As all destructors should be!
- Assignable For the strong exception guarantee (types that aren't assignable can still be used with any, but without the strong guarantee).

This is the public interface of any:

```
namespace boost {

  class any {
  public:
    any();
    any(const any&);

    template<typename ValueType>
     any(const ValueType&);

    ~any();

    any& swap(any &);
    any& operator=(const any&);

    template<typename ValueType>
     any& operator=(const ValueType&);

    bool empty() const;
    const std::type_info& type() const;
  };
}
```

Members

```
any();
```

The default constructor creates an empty instance of anythat is, an any that doesn't contain a value. Of course, there is no way of retrieving the value of an empty any, because no value exists.

```
any(const any& other);
```

Creates a distinct copy of an existing any object. The value that is contained in other is copied and stored in this.

```
template<typename ValueType> any(const ValueType&);
```

Usage

The Any library resides in namespace boost. You use the class any to store values, and the template function any_cast to subsequently retrieve the stored values. To use any, include the header "boost/any.hpp". The creation of an instance capable of storing any conceivable value is straightforward.

```
boost::any a;
```

To assign a value of some type is just as easy.

```
a=std::string("A string");
a=42;
a=3.1415;
```

Almost anything is acceptable to any! However, to actually do anything with the value contained in an any, we need to retrieve it, right? For that, we need to know the value's type.

```
std::string s=boost::any_cast<std::string>(a);
// throws boost::bad_any_cast.
```

This obviously doesn't work; because a currently contains a double, any_cast throws a bad_any_cast exception. The following, however, does work.

```
double d=boost::any_cast<double>(a);
```

any only allows access to the value if you know the type, which is perfectly sensible. These two elements are all you need to remember, typewise, for this library: the class any, for storing the values, and the template function any_cast, to retrieve them.

Anything Goes!

Consider three classes, A, B, and C, with no common base class, that we'd like to store in a std::vector. If there is no common base class, it would seem we would have to store them as void*, right? Well, not any more (pun intended), because the type of any does not change depending on the type of the value it contains. The following code shows how to solve the problem.

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

class A {
public:
  void some_function() { std::cout << "A::some_function()\n"; }
};

class B {
public:
  void some_function() { std::cout << "B::some_function()\n"; }
};

class C {
public:
  void some_function() { std::cout << "C::some_function()\n"; }
};

int main() {
```

Any Summary

Discriminated types can contain values of different types and are quite different from indiscriminate (read void*) types. We always depend heavily on type safety in C++, and there are few situations in which we are willing to do without it.

This is for good reasons: Type safety keeps us from making mistakes and improves the performance of our code. So, we avoid indiscriminate types. Still, it is not uncommon to find oneself in need of heterogeneous storage, or to insulate clients from the details of types, or to gain the utmost flexibility at lower levels of a hierarchy. any provides this functionality while maintaining full type safety, and that makes it an excellent addition to our toolbox!

Use the Any library when

- You need to store values of heterogeneous types in containers
- Storage for unknown types is required
- Types are being passed through layers that need not know anything about the types

The design of Any also serves as a valuable lesson on how to encapsulate a type without effect on the type of the enclosing class. This design can be used to create generic function objects, generic iterators, and much more. It is an example of the power of encapsulation and polymorphism in conjunction with templates.

In the Standard Library, there are excellent tools for storing collections of elements. When the need for storage of heterogeneous types arises, we want to avoid having to use new collection types. any offers a solution that works in many cases with existing containers. In a way, the template class any extends the capabilities of the Standard Library containers by packaging disparate types in a homogeneous wrapper that allows them to be made elements of those aforementioned containers.

Adding Boost.Any to an existing code base is straightforward. It doesn't require changes to the design, and immediately increases flexibility where it's applied. The interface is small, making it a tool that is easily understood.

The Any library was created by Kevlin Henney, and like all Boost libraries, has been reviewed, influenced, and refined by the Boost community.

Library 7. Variant

How Does the Variant Library Improve Your Programs?

- Typesafe storage and retrieval of a user-specified set of types
- A means to store heterogeneous types in Standard Library containers
- Compile-time checked visitation of variants
- Efficient, stack-based storage for variants

The Variant library focuses on typesafe storage and retrieval of a bounded set of typesthat is, on discriminated unions. The Boost.Variant library has many features in common with Boost.Any, but there are different tradeoffs as well as differences in functionality. The need for discriminated unions (variant types) is very common in everyday programming. One typical solution while retaining type safety is to use abstract base classes, but that's not always possible; even when it is, the cost of heap allocation and virtual functions[1] may be too high. One might also try using unsafe indiscriminate types such as void* (which leads to disaster), or typesafe but unbounded variant types, such as Boost.Any. The library we look at hereBoost.Variantsupports bounded variant typesthat is, variants where the elements come from a set of supported types.

[1] Although virtual functions do come with a very reasonable price with regard to performance.

Variant types are available in many other programming languages, and they have proven their worth time and again. There is very limited built-in support in C++ for variant types, only in the form of unions, that exist mainly for C compatibility. Boost.Variant remedies the situation through a class template variant, and accompanying tools for safely storing and retrieving values. A variant data type exposes an interface independent of the current value's type. If you've used some proprietary variant types before, you may have been exposed to types that only support a fixed set of types. That is not the case with this library; you define the set of types that are allowed in a variant when you use it, and a program can contain any number of disparate variant instantiations. To retrieve the value that is held in a variant, you either need to know the exact type of the current value, or use the provided typesafe visitor mechanism. The visitor mechanism makes Variant quite different from most other variant libraries, including Boost.Any (which on the other hand can hold a value of any conceivable type), and thereby enables a safe and robust environment for handling such types. C++ unions are only useful for built-in types and POD types, but this library offers discriminated union support for all types. Finally, efficiency aspects are covered, too, as the library stores its values in stack-based storage, thus avoiding more expensive heap allocations.

How Does Variant Fit with the Standard Library?

Boost.Variant permits storing heterogeneous types in the Standard Library containers. As there is no real support for variant types in C++, or in the C++ Standard Library, this makes Variant an excellent and useful extension to the Standard Library.

Variant

Header: "boost/variant.hpp"

This contains all of the Variant library through a single header file.

`"boost/variant/variant_fwd.hpp"`

contains forward declarations of the variant class templates.

`"boost/variant/variant.hpp"`

contains the definitions for the variant class templates.

`"boost/variant/apply_visitor.hpp"`

contains the functionality for applying visitors to variants.

`"boost/variant/get.hpp"`

contains the template function get.

`"boost/variant/bad_visit.hpp"`

contains the definition for the exception class bad_visit.

`"boost/variant/static_visitor.hpp"`

contains the definition for the visitor class template.

The following partial synopsis covers the most important members of the variant class template. Other functionality, such as the visitation mechanism, direct typesafe value retrieval, and advanced features such as creating the set of types through type sequences, are described in the "Usage" section.

```
namespace boost {
  template <typename T1,typename T2=unspecified, ...,
    typename TN=unspecified>
  class variant {
  public:

    variant();

    variant(const variant& other);

    template <typename T> variant(const T& operand);

    template <typename U1, typename U2, ..., typename UN>
      variant(const variant<U1, U2, ..., UN>& operand);

    ~variant();

    template <typename T> variant& operator=(const T& rhs);

    int which() const;
    bool empty() const;
    const std::type_info& type() const;
```

Usage

To start using variants in your programs, include the header "boost/variant.hpp". This header includes the entire library, so you don't need to know which individual features to use; later, you may want to reduce the dependencies by only including the relevant files for the problem at hand. When declaring a variant type, we must define the set of types that it will be capable of storing. The most common way to accomplish this is using template arguments. A variant that is capable of holding a value of type int, std::string, or double is declared like this.

```
boost::variant<int,std::string,double> my_first_variant;
```

When the variable my_first_variant is created, it ends up containing a default-constructed int, because int is first among the types that the variant can contain. We can also pass a value that is convertible to one of those types to initialize the variant.

```
boost::variant<int,std::string,double>
  my_first_variant("Hello world");
```

At any give time, we can assign a new value, and as long as the new value is unambiguously and implicitly convertible to one of the types that the variant can contain, it works perfectly.

```
my_first_variant=24;
my_first_variant=2.52;
my_first_variant="Fabulous!";
my_first_variant=0;
```

After the first assignment, the contained value is of type int; after the second, it's a double; after the third, it's a std::string; and then finally, it's back to an int. If we want to see that this is the case, we can retrieve the value using the function boost::get, like so:

```
assert(boost::get<int>(my_first_variant)==0);
```

Note that if the call to get fails (which would happen if my_first_variant didn't contain a value of type int), an exception of type boost::bad_get is thrown. To avoid getting an exception upon failure, we can pass a pointer to a variant to get, in which case get returns a pointer to the value or, if the requested type doesn't match the type of the value in the variant, it returns the null pointer. Here's how it is used:

```
int* val=boost::get<int>(&my_first_variant);
assert(val && (*val)==0);
```

The function get is a very direct way of accessing the contained valuein fact, it works just like any_cast does for boost::any. Note that the type must match exactly, including at least the same cv-qualification (const and volatile). However, a more restrictive cv-qualification will succeed. If the type doesn't match and a variant pointer is passed to get, the null pointer is returned. Otherwise, an exception of type bad_get is thrown.

```
const int& i=boost::get<const int>(my_first_variant);
```

Code that relies too heavily on get can quickly become fragile; if we don't know the type of the contained value, we might be tempted to test for all possible combinations, like the following example does.

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

template <typename V> void print(V& v) {
```

Variant Summary

The fact that discriminated unions are useful in everyday programming should come as no surprise, and the Boost.Variant library does an excellent job of providing efficient and easy-to-use variant types based upon discriminated unions. Because C++ unions aren't terribly useful for many types (they support only built-in types and POD types), the need for something else has been prevalent for a long time. Many attempts at creating discriminated unions have suffered from significant drawbacks. For example, previous attempts usually come with a fixed set of supported types, which seriously impedes maintainability and flexibility. Boost.Variant avoids this limitation through templates, which theoretically allows creating any variant type. Type-switching code has always been a problem when dealing with discriminated unions; it was necessary to test for the type of the current value before acting, creating maintenance headaches. Boost.Variant offers straightforward value extraction and typesafe visitation, which is a novel approach that elegantly solves that problem. Finally, efficiency has often been a concern with previous attempts, but this library addresses that too, by using stack-based storage rather than the heap.

Boost.Variant is a mature library, with a rich set of features that makes it easy and efficient to work with variant types. It nicely complements the Boost.Any library, and it should definitely be part of your professional C++ toolbox.

The authors of Boost.Variant are Eric Friedman and Itay Maman.

Library 8. Tuple

How Does the Tuple Library Improve Your Programs?

- Multiple return values from functions
- Grouping of related types
- Ties values together

C++, like many other programming languages, allows a function to return one value. However, that one value can be of arbitrary type, which allows grouping multiple values as the result, with a struct or class. Although possible, it is often inconvenient to group related return values in such constructs, because it means defining types for every distinct return type needed. To avoid copying large objects in a return value, and to avoid creating a special type to return multiple values from a function, we often resort to using non-const reference arguments or pointers, thereby allowing a function to set the caller's variables through those arguments. This works well in many cases, but some find the output parameters disconcerting in use. Also, output parameters don't emphasize that the return value is in fact return values. Sometimes, std::pair is sufficient, but even that proves insufficient when returning more than two values.

To provide for multiple return values, we need a tuple construct. A tuple is a fixed-size collection of values of specified types. Examples include pairs, triples, quadruples, and so on. Some languages come with such tuple types built in, but C++ doesn't. Given the power inherent in C++, this shortcoming can be amended by a library, which as you no doubt guessed, is just what Boost.Tuple does.

The Tuple library provides tuple constructs that are convenient to use for returning multiple values but also to group any types and operate on them with generic code.

How Does the Tuple Library Fit with the Standard Library?

The Standard Library provides a special case of tuple, a 2-tuple, called std::pair. This construct is used by Standard Library containers, which you have probably noted when operating on elements of std::map. You can store pairs in container classes, too. Of course, std::pair is not just a tool for container classes, it's useful on its own, and it comes with the convenience function std::make_pair, which automates type deduction, plus a set of operators for comparing pairs. A general solution for tuples, not just 2-tuples, is definitely even more useful. The offering from the Tuple library is not fully general, but it allows tuples up to 10 elements. (If more are needed, which seems unlikely but certainly not impossible, this limit can be extended.) What's more, these tuples are as efficient as a handcrafted solution using structs!

Tuple

Header: "boost/tuple/tuple.hpp"

This includes the tuple class template and the core of the library.

```
Header: "boost/tuple/tuple_io.hpp"
```

includes input and output operations for tuples.

```
Header: "boost/tuple/tuple_comparison.hpp"
```

includes relational operators for tuples.

The Tuple library resides in a nested namespace within boost called boost::tuples. To use tuples, include "boost/tuple/tuple.hpp", which contains the core library. For input and output operations, include "boost/tuple/tuple_io.hpp", and to include support for tuple comparisons, include "boost/tuple/tuple_comparison.hpp". Some Boost libraries have a convenience header that includes all of the library; Boost.Tuple doesn't. The reason for separating the library into different headers is to reduce compile times; if you won't be using relational operators, you shouldn't need to pay for them in terms of time and dependencies. For convenience, some of the names from the Tuple library are present in namespace boost: tuple, make_tuple, tie, and get. The following is a partial synopsis for Boost.Tuple, showing and briefly discussing the most important functions.

```
namespace boost {

  template <class T1,class T2,...,class TM> class tuple {
  public:
    tuple();

    template <class P1,class P2...,class PM>
      tuple(class P1,class P2,...,PN);

    template <class U1,class U2,...,class UN>
    tuple(const tuple<U1,U2,...,UN>&);

    tuple& operator=(const tuple&);
  };

  template<class T1,class T2,...,class TN> tuple<V1,V2,...,VN>
    make_tuple(const T1& t1,const T2& t2,...,const TN& tn);

  template<class T1,class T2,...,class TN> tuple<T1&,T2&,...,TN>
    tie(T1& t1,T2& t2,...,TN& tn);

  template <int I,class T1,class T2,...,class TN>
    RI get(tuple<T1,T2,...,TN>& t);

  template <int I,class T1,class T2,...,class TN>
    PI get(const tuple<T1,T2,...,TN>& t);

  template <class T1,class T2,...,class TM,
           class U1,class U2,...,class UM>
    bool operator==(const tuple<T1,T2,...,TM>& t,
                    const tuple<U1,U2,...,UM>& u);

  template <class T1,class T2,...,class TM,
           class U1,class U2,...,class UM>
    bool operator!=(const tuple<T1,T2,...,TM>& t,
                  const tuple<U1,U2,...,UM>& u);

  template <class T1,class T2,...,class TN,
```

Usage

Tuples live in namespace tuples, which in turn is inside namespace boost. Include "boost/tuple/tuple.hpp" to use the library. The relational operators are defined in the header "boost/tuple/tuple_comparison.hpp". Input and output of tuples are defined in "boost/tuple/tuple_io.hpp". A few of the key tuple components (tie and make_tuple) are also available directly in namespace boost. In this section, we'll cover how tuples are used in some typical scenarios, and how it is possible to extend the functionality of the library to best fit our purposes. We'll start with the construction of tuples, and gradually move on to topics that include the details of how tuples can be utilized.

Constructing Tuples

The construction of a tuple involves declaring the types and, optionally, providing a list of initial values of compatible types.[1]

[1] The constructor arguments do not have to be of the exact type specified for the elements when specializing the tuple so long as they are implicitly convertible to those types.

```
boost::tuple<int,double,std::string>
  triple(42,3.14,"My first tuple!");
```

The template parameters to the class template tuple specify the element types. The preceding example shows the creation of a tuple with three types: an int, a double, and a std::string. Providing three parameters to the constructor initializes the values of all three elements. It's also possible to pass fewer arguments than there are elements, which results in the remaining elements being default initialized.

```
boost::tuple<short,int,long> another;
```

In this example, another has elements of types short, int, and long, and they are all initialized to 0.[2] Regardless of the set of types for your tuples, this is how they are defined and constructed. So, if one of your tuple's element types is not default constructible, you need to initialize it yourself. Compared to defining structs, tuples are much simpler to declare, define, and use. There's also the convenience function, make_tuple, which makes creating tuples easier still. It deduces the types, relieving you from the monotony (and chance of error!) of specifying them explicitly.

[2] Within the context of a template, T() for a built-in type means initialization with zero.

```
boost::tuples::tuple<int,double> get_values() {
  return boost::make_tuple(6,12.0);
}
```

The function make_tuple is analogous to std::make_pair. By default, make_tuple sets the types of the elements to non-const, non-referencethat is, the plain, underlying types of the arguments. For example, consider the following variables:

```
int plain=42;
int& ref=plain;
const int& cref=ref;
```

These three variables are named after their cv-qualification (constness) and whether they are references. The tuples created by the following invocations of make_tuple all have one int element.

```
boost::make_tuple(plain);
boost::make_tuple(ref);
boost::make_tuple(cref);
```

Tuple Summary

The Tuple library brings the concept of tuples to C++. It is intuitive and concise, and although its primary use seems to be for multiple return value from functions, it is also very useful for creating all sorts of logical groupings such as storing sets of elements (as elements) in Standard Library containers. The alternative for achieving the same level of coherency is to create unique structs for every different return type (groupings), which is not only tedious work, it also removes the possibility of generic solutions for recurring tasks. These problems are alleviated with the use of the Boost.Tuple.

In this chapter, we've seen how to use the Tuple library and how to extend it in the form of function objects and algorithms that can work with any tuple. Accessing elements by index, and the get_head/get_tail member functions, provides consistency in working with tuples that enables many solutions that are impossible with other forms of user-defined types (UDTs).

The creator of Boost.Tuple, Jaakko Järvi, deserves credit for this great library. This creation goes a long way to prove that nearly anything lacking in C++ can be added through libraries by talented designers.

Part III: Function Objects and Higher-Order Programming

The following four libraries have the potential of changing the way you look at programming in C++ forever. Although function objects are not a novel concept, especially for people who have long been using and customizing the algorithms in the Standard Library, the libraries covered in this part of the book take function objects to a whole new level of abstraction. There are areas in C++ that are sometimes considered to be shortcomings when employing certain designs, such as the seemingly unavoidable proliferation of small function objects when using Standard Library algorithms. One must never forget that in C++, it's best to not be (too) judgmental of the language itself, for it was designed to handle its own shortcomings through libraries; and that's exactly what the libraries Boost.Bind and Boost.Lambda try to do for the aforementioned problem. Callback functions are another problematic area that is addressed here; the root of the problem is accentuated by using libraries for higher-order programming, because storing and invoking delayed function-like objects becomes an important feature. That's what Boost.Function does, and of course, it plays very nicely with the other two libraries mentioned here (and others, too). The final chapter discusses Boost.Signals, a library that reifies the Observer pattern. There is fantastic power in these librariesenabling programmers to write less code, more expressive statements, and really compact expressions that make code easier to read and maintain. With this power comes responsibility, because it's also quite possible to write virtually unparseable expressions. For many programmers, the acquaintance with these libraries has been an epiphanyI hope that it will be for you too.

Library 9. Bind

How Does the Bind Library Improve Your Programs?

- Adapts functions and function objects for use with Standard Library algorithms
- Consistent syntax for creating binders
- Powerful functional composition

When using the algorithms from the Standard Library, you often need to supply them with a function or a function object. This is an excellent way of customizing the behavior of algorithms, but you often end up writing new function objects because you don't have the tools necessary for functional composition and adaptation of argument order or arity. Although the Standard Library does offer some productive tools, such as bind1st and bind2nd, this is rarely enough. Even when the functionality suffices, that often implies suffering from awkward syntax that obfuscates the code for programmers who are not familiar with those tools. What you need, then, is a solution that both adds functionality and normalizes the syntax for creating function objects on-the-fly, and this is what Boost.Bind does.

In effect, a generalized binder is a sort of lambda expression, because through functional composition we can more or less construct local, unnamed functions at the call site. There are many cases where this is desirable, because it serves three purposesreducing the amount of code, making the code easier to understand, and localizing behavior, which in turn implies more effective maintenance. Note that there is another Boost library, Boost.Lambda, which takes these properties even further. Boost.Lambda is covered in the next chapter. Why shouldn't you just skip ahead to that library? Because most of the time, Boost.Bind does everything you need when it comes to binding, and the learning curve isn't as steep.

One of the keys to the success of Bind is the uniform syntax for creating function objects and the few requirements on types that are to be used with the library. The design takes focus away from how to write the code that works with your types, and sets it to where we are all most interestedhow the code works and what it actually does. When using adaptors from the Standard Library, such as ptr_fun and mem_fun_ref, code quickly becomes unnecessarily verbose because we have to provide these adaptors in order for the arguments to adhere to the requirements of the algorithms. This is not the case with Boost.Bind, which uses a much more sophisticated deduction system, and a straightforward syntax when the automatic deduction cannot be applied. The net effect of using Bind is that you'll write less code that is easier to understand.

How Does Bind Fit with the Standard Library?

Conceptually, Bind is a generalization of the existing Standard Library functions bind1st and bind2nd, with additional functionality that allows for more sophisticated functional composition. It also alleviates the need to use adaptors for pointers to functions and pointers to class members, which saves coding and potential errors. Boost.Bind also covers some of the popular extensions to the C++ Standard Library, such as the SGI extensions compose1 and compose2, and also the select1st and select2nd functions. So, Bind does fit with the Standard Library, and it does so very well indeed. The need for such functionality is acknowledged, and at last in part addressed by the Standard Library, and also in popular extensions to the STL. Boost.Bind has been accepted for the upcoming Library Technical Report.

Bind

Header: "boost/bind.hpp"

The Bind library creates function objects that bind to a function (free function or member function). Rather than supplying all of the arguments to the function directly, arguments can be delayed, meaning that a binder can be used to create a function object with changed arity (number of arguments) for the function it binds to, or to reorder the arguments any way you like.

The return types of the overloaded versions of the function bind are unspecifiedthat is, there is no guarantee for what the signature of a returned function object is. Sometimes, you need to store that object somewhere, rather than just passing it directly to another functionwhen this need arises, you want to use Boost.Function, which is covered in "Library 11: Function 11." The key to understanding what the bind-functions return is to grok the transformation that is taking place. Using one of the overloaded bind functionstemplate<class R, class F> unspecified-1 bind(F f)as an example, this would be (quoting from the online documentation), "A function object λ such that the expression λ(v1, v2, ..., vm) is equivalent to f(), implicitly converted to R." Thus, the function that is bound is stored inside the binder, and the result of subsequent invocations on that function object yields the return value from the function (if any)that is, the template parameter R. The implementation that we're covering here supports up to nine function arguments.

The implementation of Bind involves a number of functions and classes, but as users, we do not directly use anything other than the overloaded function bind. All binding takes place through the bind function, and we can never depend on the type of the return value. When using bind, the placeholders for arguments (called _1, _2, and so on) do not need to be introduced with a using declaration or directive, because they reside in an unnamed namespace. Thus, there is rarely a reason for writing one of the following lines when using Boost.Bind.

```
using boost::bind;
using namespace boost;
```

As was mentioned before, the current implementation of Boost.Bind supports nine placeholders (_1, _2, _3, and so forth), and therefore also up to nine arguments. It's instructive to at least browse through the synopsis for a high-level understanding of how the type deduction is performed, and when/why this does not always work. Parsing the signatures for member function pointers and free functions takes a while for the eye to get used to, but it's useful. You'll see that there are overloads for both free functions and class member functions. Also, there are overloads for each distinct number of arguments. Rather than listing the synopsis here, I encourage you to visit Boost.Bind's documentation at www.boost.org.

Usage

Boost.Bind offers a consistent syntax for both functions and function objects, and even for value semantics and pointer semantics. We'll start with some simple examples to get to grips with the usage of vanilla bindings, and then move on to functional composition through nested binds. One of the keys to understanding how to use bind is the concept of placeholders. Placeholders denote the arguments that are to be supplied to the resulting function object, and Boost.Bind supports up to nine such arguments. The placeholders are called _1, _2, _3, _4, and so on up to _9, and you use them in the places where you would ordinarily add the argument. As a first example, we shall define a function, nine_arguments, which is then called using a bind expression.

```
#include <iostream>
#include "boost/bind.hpp"

void nine_arguments(
  int i1,int i2,int i3,int i4,
    int i5,int i6,int i7,int i8, int i9) {
    std::cout << i1 << i2 << i3 << i4 << i5
      << i6 << i7 << i8 << i9 << '\n';
}

int main() {
  int i1=1,i2=2,i3=3,i4=4,i5=5,i6=6,i7=7,i8=8,i9=9;
  (boost::bind(&nine_arguments,_9,_2,_1,_6,_3,_8,_4,_5,_7))
    (i1,i2,i3,i4,i5,i6,i7,i8,i9);
}
```

In this example, you create an unnamed temporary binder and immediately invoke it by passing arguments to its function call operator. As you can see, the order of the placeholders is scrambledthis illustrates the reordering of arguments. Note also that placeholders can be used more than once in an expression. The output of this program is as follows.

```
921638457
```

This shows that the placeholders correspond to the argument with the placeholder's numberthat is, _1 is substituted with the first argument, _2 with the second argument, and so on. Next, you'll see how to call member functions of a class.

Calling a Member Function

Let's take a look at calling member functions using bind. We'll start by doing something that also can be done with the Standard Library, in order to compare and contrast that solution with the one using Boost.Bind. When storing elements of some class type in Standard Library containers, a common need is to call a member function on some or all of these elements. This can be done in a loop, and is all-too-often implemented thusly, but there are better solutions. Consider the following simple class, status, which we'll use to show that the ease of use and power of Boost.Bind is indeed tremendous.

```
class status {
  std::string name_;
  bool ok_;
public:
  status(const std::string& name):name_(name),ok_(true) {}

  void break_it() {
    ok_=false;
  }

  bool is_broken() const {
    return ok_;
  }
```

Bind Summary

Use Bind when

- You need to bind a call to a free function, and some or all of its arguments
- You need to bind a call to a member function, and some or all of its arguments
- You need to compose nested function objects

The existence of a generalized binder is a tremendously useful tool when it comes to writing terse, coherent code. It reduces the number of small function objects created for adapting functions/function objects, and combinations of functions. Although the Standard Library already offers a small part of the functionality found in Boost.Bind, there are significant improvements that make Boost.Bind the better choice in most places. In addition to the simplification of existing features, Bind also offers powerful functional composition features, which provide the programmer with great power without negative effects on maintenance. If you've taken the time to learn about bind1st, bind2nd, ptr_fun, mem_fun_ref, and so forth, you'll have little or no trouble transitioning to Boost.Bind. If you've yet to start using the current binder offerings from the C++ Standard Library, I strongly suggest that you start by using Bind, because it is both easier to learn and more powerful.

I know many programmers who have yet to experience the wonders of binders in general, and function composition in particular. If you used to be one of them, I'm hoping that this chapter has managed to convey some of the tremendous power that is brought forth by the concept as such. Moreover, think about the implications this type of function, declared and defined at the call site, will have on maintenance. It's going to be a breeze compared to the dispersion of code that can easily be caused by small, innocent-looking[8] function objects that are scattered around the classes merely to provide the correct signature and perform a trivial task.

[8] But they're not.

The Boost.Bind library is created and maintained by Peter Dimov, who has, besides making it such a complete facility for binding and function composition, also managed to make it work cleanly for most compilers.

Library 10. Lambda

How Does the Lambda Library Improve Your Programs?

- Adapts functions and function objects for use with Standard Library algorithms
- Binds arguments to function calls
- Transforms arbitrary expressions into function objects compatible with the Standard Library algorithms
- Defines unnamed functions at the call site, thereby improving readability and maintainability of the code
- Implements predicates when and where needed

When using the Standard Library, or any library employing a similar design that relies on algorithmic configuration by the means of functions and function objects, one often ends up writing lots of small function objects that perform quite trivial operations. As we saw in "Library 9: Bind 9," this can quickly become a problem, because an explosion of small classes that are scattered through the code base is not easily maintained. Also, understanding the code where the function objects are actually invoked is harder, because part of the functionality is defined elsewhere. A perfect solution to this problem is a way to define these functions or function objects directly at the call site. This typically makes the code faster to write, easier to read, and more readily maintained, as the definition of the functionality then resides in the location where it is used. This is what the Boost.Lambda library offers, unnamed functions defined at the call site. Boost.Lambda works by creating function objects that can be defined and invoked directly, or stored for later invocation. This is similar to the offerings from the Boost.Bind library, but Boost.Lambda does both argument binding and much more, by adding control structures, automatic conversions of expressions into function objects, and even support for exception handling in lambda expressions.

The term lambda expression, or lambda function, originates from functional programming and lambda calculus. A lambda abstraction defines an unnamed function. Although lambda abstractions are ubiquitous in functional programming languages, that's not the case for most imperative programming languages, such as C++. But, using advanced techniques such as expression templates, C++ makes it possible to augment the language with a form of lambda expressions.

The first and foremost motivation for creating the Lambda library is to enable unnamed functions for use with the Standard Library algorithms. Because the use of the Standard Library has virtually exploded since the first C++ Standard in 1998, our knowledge of what's good and what's missing has rapidly increasedand one of the parts that can be problematic is the definition of numerous small function objects, where a simple expression would seem to suffice. The function object issue is obviously addressed by this library, but there is still room for exploration of the uses of lambda functions. Now that lambda functions are available, we have the opportunity to apply them to problems that previously required totally different solutions. It's both fascinating and exciting that it is possible to explore new programming techniques in a language as mature as C++. What new idioms and ways of solving problems will arise from the presence of unnamed functions and expression templates? The truth is that we don't know, because we have yet to try them all out! Still, the focus here is on the practical problems that the library explicitly addressesavoiding code bloat and scattered functionality through lambda expressionsfunctions defined at the call site. We can do many wonderful things with thisand we can be really terse about it, which should satisfy both programmers, who can focus more on the problem at hand, and their managers, who can reap the benefits of a higher production rate (and, hopefully, more easily maintained code!).

How Does Lambda Fit with the Standard Library?

The library addresses a problem that is often encountered when using the Standard Library algorithmsthe need to define many simple function objects just to comply with the requirements of the algorithms. Almost all of the Standard Library algorithms also come in a version that accepts a function object, to perform operations such as ordering, equality, transformations, and so on. To a limited extent, the Standard Library supports functional composition, through the binders bind1st and bind2nd. However, these are very limited in what they can produce, and they provide only argument binding, not bindings for expressions. Given that both flexible support for binding arguments and for creating function objects directly from expressions are available in the Boost.Lambda library, it is an excellent companion to the C++ Standard Library.

Lambda

Header: "boost/lambda/lambda.hpp"

This includes the core of the library.

`"boost/lambda/bind.hpp"`

defines bind functions.

`"boost/lambda/if.hpp"`

defines the lambda equivalent of if, and the conditional operator.

`"boost/lambda/loops.hpp"`

defines looping constructs (for example, while_loop and for_loop).

`"boost/lambda/switch.hpp"`

defines the lambda equivalent of switch statements.

`"boost/lambda/construct.hpp"`

defines tools for adding construction/destruction and new/delete to lambda expressions.

`"boost/lambda/casts.hpp"`

provides cast operators for lambda expressions.

`"boost/lambda/exceptions.hpp"`

defines tools for exception handling in lambda expressions.

`"boost/lambda/algorithm.hpp" and "boost/lambda/numeric.hpp"`

defines lambda versions (essentially function objects) of C++ Standard library algorithms to be used in nested function invocations.

Usage

This library, like most other Boost libraries, is purely defined in header files, which means that you don't have to build anything to get started. However, understanding a little something about lambda expressions is definitely helpful. The following sections will walk you through this library, even including how to perform exception handling in lambda expressions! The library is quite extensive, and there's a lot of power waiting ahead. A lambda expression is often called an unnamed function. It is declared and defined when it's neededthat is, at the call site. This is very useful, because we often need to define an algorithm in another algorithm, something that isn't really supported by the language. Instead, we externalize behavior by bringing in functions and function objects from a wider scope, or use nested loop constructs with the algorithmic expressions encoded in the loops. As we shall see, this is where lambda expressions come to the rescue. This section consists of many examples, and there is often one part of the example that demonstrates how the solution would be coded using "traditional" tools. The intent is to show when and how lambda expressions help programmers write more logic with less code. There is a certain learning curve associated with lambda expressions, and the syntax may seem daunting at first glance. Like every new paradigm or tool, this one must be learnedbut trust me when I say that the profit definitely outweighs the cost.

A Little Teaser

The first program using Boost.Lambda should whet your appetite for lambda expressions. First of all, note that the lambda types are declared in the namespace boost::lambdatypically, you bring these declarations into scope with a using directive or using declarations. The core functionality of the library is available when including the file "boost/lambda/lambda.hpp", which is sufficient for our first program.

```
#include <iostream>
#include "boost/lambda/lambda.hpp"
#include "boost/function.hpp"

int main() {
  using namespace boost::lambda;

  (std::cout << _1 << " " << _3 << " " << _2 << "!\n")
    ("Hello","friend","my");

  boost::function<void(int,int,int)> f=
    std::cout << _1 << "*" << _2 << "+" << _3
      << "=" <<_1*_2+_3 << "\n";

  f(1,2,3);
  f(3,2,1);
}
```

The first expression looks peculiar at first glance, but it helps to mentally divide the expression as the parentheses do; the first part is a lambda expression that basically says, "print these arguments to std::cout, but don't do it right now, because I don't yet know the first, second, and third arguments." The second part of the expression actually invokes the function by saying, "Hey! Here are the three arguments that you need." Look at the first part of the expression again.

```
std::cout << _1 << " " << _3 << " " << _2 << "!\n"
```

You'll note that there are three placeholders, aptly named _1, _2, and _3, in the expression.[1] These placeholders denote the delayed arguments to the lambda expression. Note that unlike the syntax of many functional programming languages, there's no keyword or name for creating lambda expressions; it is the presence of the placeholders that signal that this is a lambda expression. So, this is a lambda expression that accepts three arguments of any type that support streaming through operator<<. The arguments are printed to cout in the order 1-3-2. Now, in the example, we enclose this expression in parentheses, and then invoke the resulting function object by passing three arguments to it: "Hello", "friend", and "my". This results in the following output:

Lambda Summary

Use Lambda when

- You would otherwise create a simple function object
- You need to tweak argument order or arity for function calls
- You want to create standard-conforming function objects on-the-fly
- You need flexible and readable predicates

The preceding reasons are just some of the cases where using this library makes perfect sense. Although the most common uses arise together with Standard Library algorithms, that's at least in part due to the fact that such designs still aren't very common in other libraries (the Boost libraries notwithstanding). Although the notion of algorithmic configuration through function objects needs no further proof of its usefulness, there is a long way to go before we reach conclusive insights into what domains clearly can benefit from such designs. Just by thinking about potential uses of this library is a sure way to improve your current designs.

Boost.Lambda is one of my favorite libraries, mainly because it offers so much accessible functionality that isn't otherwise provided by the language. As the STL made its way into the hearts of programmers all over the world, there was still something missing. To work efficiently with the algorithms, something more than function objects was required. Such was the impetus for Boost.Lambda, with its plethora of features that enable a truly concise programming style. There are many areas where lambda expressions are usable, but there is still much to be explored. This is to some degree functional programming in C++, which is a paradigm yet to be explored in full. This introduction to the Lambda library can empower you to continue that exploration. It's only fair to state that the syntax sometimes can be a bit clumsy compared to "real" functional programming languages, and that it does take some time for new users to get accustomed to it. But, likewise, it's fair to say that there is great value for any C++ programmer in this library! I hope it becomes one of your favorite libraries, too.

Many thanks to Jaakko Järvi and Gary Powell, the authors of this library and true pioneers of functional programming in C++!

Library 11. Function

How Does the Function Library Improve Your Programs?

- Stores function pointers and function objects for subsequent invocation

The need to store functions and function objects is common in designs with callbacks, and where functions or classes are configured with custom functionality through either function pointers or function objects. Traditionally, function pointers have been used to accommodate the need for both callbacks and delayed functions. However, using only function pointers is too limiting, and what would be better is a generalized mechanism that defines the signature of the function to be stored, and leaves it up to the caller to decide which type of function-like entity (function pointer or function object) should be provided. It would then be possible to use anything that behaves like a functionfor example, the result of using Boost.Bind and Boost.Lambda. This, in turn, means that it is possible to add state to such stored functions (because function objects are classes). This generalization is what Boost.Function offers. The library is used to store, and subsequently invoke, functions or function objects.

How Does Function Fit with the Standard Library?

The library provides functionality that does not currently exist in the Standard Library. Generalized callbacks are a natural part of virtually all frameworks decoupling the presentation layer from the business logic, and the uses are plentiful. As there is no support in the C++ Standard Library for storing function pointers and function objects for later invocation, this is an important addition to the tools offered by the Standard Library. Also, the library is compatible with the binders from the Standard Library (bind1st and bind2nd), as well as other binder libraries that extend the aforementioned binders, such as Boost.Bind and Boost.Lambda.

Function

Header: "boost/function.hpp"

The header "function.hpp" includes prototypes for functions with 0 to 10 arguments. (This is implementation defined, but 10 is the default limit for the current implementation.[1]) It is also possible to include only the header that corresponds to the number of arguments you need to usethe files are named "function/functionN.hpp", where N is in the range 0 to 10. There are two different interfaces for Boost.Function, one that is most appealing because it is syntactically close to a function declaration (and doesn't require the signature to include the number of arguments), and the other is appealing because it works with more compilers. Which to choose depends, at least in part, on the compiler that you are using. If you can, use what we refer to as the preferred syntax. Throughout this chapter, both forms will be used.

[1] Boost.Function can be configured to support up to 127 arguments.

Declarations Using the Preferred Syntax

A declaration of a function includes the signature and return type of the function or function object that the function is to be compatible with. The type of the result and the arguments are all supplied as a single argument to the template. For example, the declaration of a function that returns bool and accepts an argument of type int looks like this:

```
boost::function<bool (int)> f;
```

The argument list is supplied inside the parentheses, separated by commas, just like a function declaration. Thus, declaring a function that returns nothing (void) and takes two arguments, of type int and double, looks like this:

```
boost::function<void (int,double)> f;
```

Declarations Using the Compatible Syntax

The second way of declaring functions is to supply separate template type arguments for the return type and the argument types for the function call. Also, there's a suffix for the name of the function class, which is an integer that corresponds to the number of arguments the function will accept. For example, the declaration of a function that returns bool and accepts an argument of type int looks like this:

```
boost::function1<bool,int> f;
```

The numbering is based on the number of arguments that the function acceptsin the preceding example, there is one argument (int) and therefore function1 is needed. More arguments simply means supplying more template type parameters to the template and changing the numeric suffix. A function that returns void and accepts two arguments of type int and double looks like this:

```
boost::function2<void,int,double> f;
```

The library actually consists of a family of classes, each taking a different number of arguments. There is no need to take this into account when including the header "function.hpp", but if including the numbered versions, you must include the correct numbered header.

The preferred syntax is easier to read and is analogous to declaring a function, so you should use it when you can. Unfortunately, although the preferred syntax is perfectly legal C++ and easier to read, not all compilers support it as yet. If your compiler is among those that cannot handle the preferred syntax, you need to use the alternative form. If you need to write your code with maximum portability, you might also choose to use the alternative form. Let's take a look at the most important parts of a function's interface.

Usage

To start using Boost.Function, include "boost/function.hpp", or any of the numbered versions, ranging from "boost/function/function0.hpp" to "boost/function/function10.hpp". If you know the arity of the functions you want to store in functions, it taxes the compiler less to include the exact headers that are needed. When including "boost/function.hpp", the other headers are all included, too.

The best way to think of a stored function is a normal function object that is responsible for wrapping another function (or function object). It then makes perfect sense that this stored function can be invoked several times, and not necessarily at the time when the function is created. When declaring functions, the most important part of the declaration is the function signature. This is where you tell the function about the signature and return type of the functions and/or function objects it will store. As we've seen, there are two ways to perform such declarations. Here's a complete program that declares a boost::function that is capable of storing function-like entities that return bool (or a type that is implicitly convertible to bool) and accept two arguments, the first convertible to int, and the second convertible to double.

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

bool some_func(int i,double d) {
  return i>d;
}

int main() {
  boost::function<bool (int,double)> f;
  f=&some_func;
  f(10,1.1);
}
```

When the function f is first created, it doesn't store any function. It is empty, which can be tested in a Boolean context or with 0. If you try to invoke a function that doesn't store a function or function object, it throws an exception of the type bad_function_call. To avoid that problem, we assign a pointer to some_func to f using normal assignment syntax. That causes f to store the pointer to some_func. Finally, we invoke f (using the function call operator) with the arguments 10 (an int) and 1.1 (a double). When invoking a function, one must supply exactly the number of arguments the stored function or function object expects.

The Basics of Callbacks

Let's look at how we would have implemented a simple callback before we knew about Boost.Function, and then convert the code to make use of function, and examine which advantages that brings forth. We will start with a class that supports a simple form of callbackit can report changes to a value by calling whoever is interested in the new value. The callback will be a traditional C-style callbackthat is, a free function. This callback could be used, for example, for a GUI control that needs to inform observers that the user changed its value, without having any special knowledge about the clients listening for that information.

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

void print_new_value(int i) {
  std::cout <<
    "The value has been updated and is now " << i << '\n';
}

void interested_in_the_change(int i) {
  std::cout << "Ah, the value has changed.\n";
}
```

Function Summary

Use Function when

- You need to store a callback function, or function object
- You want to decouple function calls from the implementation, for example between the GUI and the implementation
- You want to store function objects created by binder libraries to be invoked at a later time, or multiple times

Boost.Function is an important addition to the offerings from the Standard Library. The well-known technique of using function pointers as a callback mechanism is extended to include anything that behaves like a function, including function objects created by binder libraries. Through the use of Boost.Function, it is easy to add state to the callbacks, and to adapt existing classes and member functions to be used as callback functions.

There are several advantages to using Boost.Function rather than function pointers: relaxed requirements on the signature through compatible function objects rather than exact signatures; the possibility to use binders, such as Boost.Bind and Boost.Lambda; the ability to test whether functions are emptythat is, that there is no targetbefore attempting to invoke them; and the notion of stateful objects rather than just stateless functions. Each of these advantages favor using Boost.Function over the C-style callbacks that have been prevalent in solving this type of problem. Only when the small additional cost of using Boost.Function compared to function pointers is prohibitive should the function pointer technique be considered.

Boost.Function was created by Douglas Gregor. It is a library with many powerful features, and is expertly designed and implemented to provide exceptional user value.

Library 12. Signals

How Does the Signals Library Improve Your Programs?

- Flexible multicast callbacks for functions and function objects
- A robust mechanism for triggering and handling events
- Compatibility with function object factories, such as Boost.Bind and Boost.Lambda

The Boost.Signals library reifies signals and slots, where a signal is something that can be "emitted," and slots are connections that receive such signals. This is a well-known design pattern that goes under a few different namesObserver, signals/slots, publisher/subscriber, events (and event targets)but these names all refer to the same thing, which is a one-to-many relation between some source of information and instances that are interested in knowing when that information changes. There are many cases where this design pattern is used; one of the most obvious is in GUI code, where certain actions (for example, the user clicks a button) are loosely connected to some kind of action (the button changes its appearance, and some business logic is performed). There are many more cases where signals and slots are useful to decouple the trigger of an action (signal) from the code that handles it (one or more slots). This can be used to dynamically alter the behavior of the handling code, to allow multiple handlers of the same signal, or to reduce type dependencies through an abstract connection between types via signals and slots. With Boost.Signals, it is possible to create signals that accept slots with any given function signaturethat is, slots that accept arguments of arbitrary types. This approach makes the library very flexible; it accommodates the signaling needs of virtually any domain. By decoupling the source of the signal and the handlers thereof, systems become more robust in terms of both physical and logical dependencies. It's possible to let the signaling types be totally ignorant of the slot types, and vice versa. This is imperative to achieve a higher level of reusability, and it can help break cyclic dependencies. So, a signals and slots library isn't only about object-oriented callbacks, it's also about the robustness of the whole system to which it is applied.

How Does Signals Fit with the Standard Library?

There is nothing in the C++ Standard Library that addresses callbacks, yet there is an obvious need for such facilities. Boost.Signals is designed in the same spirit as the Standard Library, and it is a great addition to the Standard Library toolbox.

Signals

Header: "boost/signals.hpp"

This includes all of the library through a single header.

```
"boost/signals/signal.hpp"
```

contains the definition of signals.

```
"boost/signals/slot.hpp"
```

contains the definition of the slot class.

```
"boost/signals/connection.hpp"
```

contains definitions of the classes connection and scoped_connection.

To use this library, either include the header "boost/signals.hpp", which ensures that the entire library is available, or include the separate headers containing the functionality that you need. The core of the Boost.Signals library exists in namespace boost, and advanced features reside in boost::signals.

The following is a partial synopsis for signal, followed by a brief discussion of the most important members. For a full reference, see the online documentation for Signals.

```
namespace boost {

  template<typename Signature,
  // Function type R(T1, T2, ..., TN)
    typename Combiner = last_value<R>,
    typename Group = int,
    typename GroupCompare = std::less<Group>,
    typename SlotFunction = function<Signature> >
  class signal : public signals::trackable,
                 private noncopyable {
  public:
    signal(const Combiner&=Combiner(),
           const GroupCompare&=GroupCompare());

    ~signal();

    signals::connection connect(const slot_type&);
    signals::connection connect(
      const Group&,
      const slot_type&);

    void disconnect(const Group&);

    std::size_t num_slots() const;

    result_type operator()
      (T1, T2, ..., TN);
  };
}
```

Types

Let's have a look first at the template parameters for signal. There are reasonable defaults for all but the first

Usage

When faced with needing more than one piece of code in a program to handle a given event, typical solutions involve callbacks through function pointers, or directly coded dependencies between the subsystem that fires the event and the subsystems that need to handle it. Circular dependencies are a common result of such designs. Using Boost.Signals, you gain flexibility and decoupling. To start using the library, include the header "boost/signals.hpp".[2]

The following example demonstrates the basic properties of signals and slots, including how to connect them and how to emit the signal. Note that a slot is something that you provide, either a function or a function object that is compatible with the function signature of the signal. In the following code, we create both a free function, my_first_slot, and a function object, my_second_slot; both are then connected to the signal that we create.

```
#include <iostream>
#include "boost/signals.hpp"

void my_first_slot() {
  std::cout << "void my_first_slot()\n";
}

class my_second_slot {
public:
  void operator()() const {
    std::cout <<
      "void my_second_slot::operator()() const\n";
  }
};

int main() {
  boost::signal<void ()> sig;

  sig.connect(&my_first_slot);
  sig.connect(my_second_slot());

  std::cout << "Emitting a signal...\n";
  sig();
}
```

We start by declaring a signal, which expects slots that return void and take no arguments. We then connect two compatible slot types to that signal. For one, we call connect with the address of the free function, my_first_slot. For the other, we default-construct an instance of the function object my_second_slot and pass it to connect. These connections mean that when we emit a signal (by invoking sig), the two slots will be called immediately.

```
sig();
```

When running the program, the output will look something like this:

```
Emitting a signal...
void my_first_slot()
void my_second_slot::operator()() const
```

However, the order of the last two lines is unspecified because slots belonging to the same group are invoked in an unspecified order. There is no way of telling which of our slots will be called first. Whenever the calling order of slots matters, you must put them into groups.

Grouping Slots

It is sometimes important to know that some slots are called before others, such as when the slots have side effects that other slots might depend upon. Groups is the name of the concept that supports such requirements. It is a signal

Signals Summary

Use Signals when

- You need robust callbacks
- There can be multiple handlers of events
- The connection between the signal and the connected slots should be configurable at runtime

That Boost.Signals supersedes old-style callbacks should be blatantly clear by now, and this library is one of the best signals and slots implementations available. The design pattern that the library captures is well known and has been studied for a long time, so the domain is mature. Some programming languages already have such mechanisms available directly in the languagefor example, delegates and events in .NET. In C++, the problem is elegantly solved with libraries. Signals and slots are used to separate the trigger mechanism of an event from the code that handles it. This separation decouples subsystems and makes them more comprehensible. It also solves the problem of updating multiple interested parties when important events take place. There are numerous places in a typical program or library where signals and slots are useful. Whether you are writing a GUI framework or an intrusion detection system for a power plant, Signals is ready to take care of all your signaling needs. It is easy to learn how to use, yet it also offers the advanced functionality that is required for complex tasks. For example, custom Combiners make it possible to write event mechanisms that are tailor-made for a certain domain.

Boost.Signals was written by Douglas Gregor (who incidentally also wrote Boost.Function). This is a great library; thank you Doug!

Endnotes

2. The Boost.Signals library and the Boost.Regex library are the only libraries covered in this book that actually require compiling and linking for use. The process is simple, and it's described in great detail in the online documentation, so I won't cover it here.

3. binary_search has the attractive complexity O(logN).

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index