EXTENDED STL

Collections and Iterators

Volume 1

MATTHEW WILSON

Praise for *Extended STL, Volume 1*

"Wilson's menu of STL treatments will no doubt be good eating for generic programming adherents, ardent C programmers just now taking on STL and C++, Java programmers taking a second look at C++, and authors of libraries targeting multiple platforms and languages. Bon appetit!"

—George Frazier, Cadence Design Systems, Inc.

"A thorough treatment of the details and caveats of STL extension."

—Pablo Aguilar, C++ Software Engineer

"This book is not just about extending STL, it's also about extending my thinking in C++."

—Serge Krynine, C++ Software Engineer, RailCorp Australia

"You might not agree 100% with everything Wilson has to say, but as a whole his book is the most valuable, in-depth study of practical STL-like programming."

—Thorsten Ottosen, M.C.S., Boost Contributor

"Wilson is a master lion tamer, persuading multifarious third-party library beasts to jump through STL hoops. He carefully guides the reader through the design considerations, pointing out the pitfalls and making sure you don't get your head bitten off."

—Adi Shavit, Chief Software Architect, EyeTech Co. Ltd

"Wilson's book provides more than enough information to change the angst/uncertainty level of extending STL from 'daunting' to 'doable.' "

—Garth Lancaster, EDI/Automation Manager, Business Systems Group, MBF Australia

"This book will open up your eyes and uncover just how powerful STL's abstractions really are."

—Nevin ":-)" Liber, 19-year veteran of C++

"In the canon of C++ there are very few books that extend the craft. Wilson's work consistently pushes the limits, showing what can and cannot be done, and the tradeoffs involved."

—John O'Halloran, Head of Software Development, Mediaproxy

"Essential concepts and practices to take the working programmer beyond the standard library."

—Greg Peet

"*Extended STL* is not just a book about adapting the STL to fit in with your everyday work, it's also an odyssey through software design and concepts, C++ power techniques, and the perils of real-world software development—in other words, it's a Matthew Wilson book. If you're serious about C++, I think you should read it."

—Björn Karlsson, Principle Architect, ReadSoft; author of *Beyond the C++ Standard Library: An Introduction to Boost*

Extended STL, Volume 1

This page intentionally left blank

Extended STL, Volume 1

Collections and Iterators

Matthew Wilson

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Wilson, Matthew.
Extended STL, Volume 1 : collections and iterators / Matthew Wilson.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-30550-7 (pbk. : alk. paper)
1. C++ (Computer program language) 2. Standard template library. I. Title.
QA76.73.C153W55235 2007
005.13'3--dc22 2007016507

ISBN 10: 0-321-30550-7
ISBN 13: 978-0-321-30550-3
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, June 2007

This one's dedicated to

Uncle John, who wasn't kidding about the second time round,

and to

Ben and Harry, whose "I want you to play with me, Daddy" *saved me from many a hard day's work (and meeting my first two deadlines),*

but mostly to

My beautiful wife, Sarah, without whom little would be achievable and less would be worth the effort, whose support in a whole manner of ways exceeds even my own optimistic fancies.

This page intentionally left blank

Contents

Preface

My Uncle John is what my parents' generation would call "a man's man." He's tough, rugged, a bit scary, with more than a little of the cowboy in him, and he would admit to fear about as readily as I could render modest defeat. So when he described to me that the challenge in doing your second parachute jump is overcoming the fear of the known, I took note. Having now written two books, I can certainly attest to this same fear. Starting a second when you know how much suffering awaits is not something done lightly. So the question arises, why have I done so?

The reason, elucidated in the Prologue, amounts to an attempt to answer the following seemingly simple dichotomy.

- C++ is too complex.
- C++ is the only language sufficiently powerful for my needs.

One area in which this dichotomy is most pronounced is in using and, particularly, in extending the Standard Template Library (STL). This book (and its sibling, Volume 2), distills the knowledge and experience I have accumulated in tackling this challenging subject over the last decade or so.

Aims

This book describes one good way to use and extend the STL. It defines the following:

- The *collection* concept and how it differs from the *container* concept
- The *element reference category* concept, including why it's important, how it's defined, how it's detected, and the compromises it imposes on the design of STL extension collections and iterators
- The phenomenon of *external iterator invalidation* and the implications of its surprising behavior on the design of STL-compatible collections
- A mechanism for detecting features of arbitrary collections that may or may not provide mutating operations

It explains several issues:

- Why a transforming iterator adaptor must return elements by value
- Why a filtering iterator must always be given a pair of iterators to manipulate
- What to do if the underlying collection changes during iteration
- Why you should proscribe meaningless syntax for your output iterator classes and how to do so using the ***Dereference Proxy*** pattern

It demonstrates how to:

- Adapt *elements-en-bloc APIs* to the STL collection concept
- Adapt *element-at-a-time APIs* to the STL collection concept
- Share enumeration state in order to properly fulfill the requirements of the *input iterator* concept
- Enumerate potentially infinite collections
- Specialize standard algorithms for specific iterator types to optimize performance
- Define a safe, platform-independent STL extension for the system environment, implemented in terms of a global variable
- Adapt a collection whose iterator instances' copyability is determined at runtime
- Provide access to a reversible collection that is not repeatable
- Write into a character buffer using an iterator

Extended STL addresses these issues and more. It also looks at how general-purpose, STL-compliant libraries may be built without sacrificing robustness, flexibility, and, especially, performance. *Extended STL* teaches you how to have your abstraction cake, with efficiency cream, and eat it.

You should read this book if you want to:

- Learn specific principles and techniques for STL extension
- Learn more about the STL, by looking *inside* the implementation of STL extensions
- Learn general techniques for implementing wrappers over operating system APIs and technology-specific libraries
- Learn how to write iterator adaptors and understand the reasons behind the restrictions on their implementations and use
- Pick up techniques for optimizing the performance of general-purpose libraries
- Use proven software components for STL extension

Subject Matter

I believe that you must write about what you know. Because the main purpose of this book is to impart understanding of the process and issues of STL extension, much of the material discussed derives from my work with my own (open-source) **STLSoft** libraries. Since I've implemented just about everything in **STLSoft** from scratch, it's therefore the best material to enable me to speak authoritatively. This is especially important when discussing design errors; publicly documenting other people's design errors in detail is unlikely to achieve many positive outcomes.

But this does not mean that reading *Extended STL* obliges you to use **STLSoft** or that followers of other libraries cannot learn anything appropriate to their practice here. Indeed, rather than proselytizing the use of any particular library, the material presented gives you an inside-out look at STL extension, focusing on STL principles and extension practices, rather than relying on extant knowledge of **STLSoft** or any other library. If, when you've read this book, you don't use the **STLSoft** libraries, I won't be troubled, so long as you've taken away useful knowledge on how to implement and use other STL extensions.

I make no pretension that the methods of STL extension I shall demonstrate in any way represent the *only* way. C++ is a very powerful language that, sometimes to its detriment, supports a variety of styles and techniques. For example, many, though not all, collections are best implemented as STL collections, while others are better represented as stand-alone iterators. There's a fair amount of overlap, about which much equivocation abides.

For most STL extensions discussed, I take the reader (and the author, in some cases!) on a journey from the raw APIs being wrapped up through intermediate, and often flawed, implementations before reaching an optimum, or at least optimal, version. I do not shy away from the implementation and/or conceptual complexities. Indeed, some of the techniques required to mold external APIs into STL form necessarily involve a degree of technical cunning. I'm not, for the sake of a simple tale, going to pretend that these things don't exist or leave them unexplained in the implementation. I will cover these things, and in so doing I hope to be able to (1) debunk some of their complexity and (2) explain why they are necessary.

One of the best ways to understand STL is to learn how STL components are implemented, and the best way to do that is by implementing them. If you don't have the time (or the inclination) to do that, I recommend that you avail yourself of the *second* best way, which is to read this book.

Structure

This book is divided into three main parts.

Part One: Foundations

This collection of small chapters provides grounding for the material discussed in Parts II and III. It begins with a brief recap of the main features of the STL, followed by a discussion of concepts and principles pertinent to STL extension, including the introduction of a new concept, the *element reference category*. The next few chapters consider foundational concepts, mechanisms, paradigms, and principles: conformance, constraints, contracts, *DRY SPOT*, RAII, and shims. The remaining chapters cover template tools and techniques, including traits and *inferred interface adaptation*, and several essential components used in the implementations described in Parts II and III.

Part Two: Collections

This represents the bulk of the book. Each chapter covers one or more related real-world collection and its adaptation into an STL extension collection component along with suitable iterator types. The subject matter tracks adaptations of subjects as diverse as file system enumeration, COM enumerators, non-STL containers, Scatter/Gather I/O, and even collections whose elements are subject to external change. The issues covered include concepts of iterator category selection and element reference categories, state sharing, mutability, and external iterator invalidation.

Part Three: Iterators

While the material in Part II includes the definition of iterator types associated with collections, Part III is devoted to stand-alone iterator types. The subjects covered range from custom output iterator types, including a discussion of simple extension of the functionality of `std::ostream_iterator`, to sophisticated iterator adaptors that can filter and transform the types and/or values of the underlying ranges to which they're applied.

Volume 2

Volume 2 is not yet complete and its structure not finalized, but it will contain material on *functions*, *algorithms*, *adaptors*, *allocators*, and the STL extension concepts *range* and *view*.

Supplementary Material

CD-ROM

The accompanying CD-ROM contains various free software libraries (including all those covered in the text), test programs, tools, and other useful software. Also included are three full but unedited chapters that didn't make it into print—either to save space or to avoid too much compiler specificity—and numerous notes and subsections from other chapters.

Online Resources

Supplementary material will also be available online at http://extendedstl.com/.

Acknowledgments

As is customary, I owe a huge debt to my wife, Sarah. With this book, as with the last, she has supported me through missed deadline after missed deadline with minimal complaint. She hopes this is the last book I will write. Marriage being all about compromise, I've promised her this is the last one that will take me years (not months) to write. Wish me luck.

I also have to thank my mum and Robert for their unwavering support and, in particular, tolerance of receiving questions on grammar at all hours of the day and night (they're GMT, I'm GMT+10).

I've ridden thousands of kilometers during the writing of this book, many of them with my (younger and fitter) friend Dave Treacy. On those days when I've not been stopping every few kilometers to write down my latest bunch of inspirations, I've ridden with Dave, whose conversations have served as very useful distractions from my STL obsessions and whose encouragement has been most welcome. Thanks, Dave, but beware: The Doctor will show you a clean pair of wheels some day!

Garth Lancaster has served many roles in our recent history: advisor, client, friend, fellow gourmand, reviewer. Garth, thanks for the frequent, but always considered, input into my libraries, books, business life, and dinner itinerary. See you at the launch party at Nino's!

Simon Smith is one of my oldest friends in my adopted country and a super-smart cookie. I know this to be so, partly because he is constantly being poached by bigger and better companies who recognize his singular techno-managerial abilities, but mainly because he keeps hiring me to conduct analyses of his charges, to identify where he can apply his unique talents in making them run better, faster, cheaper, and more harmoniously. (Either that or he's just being kind.)

Much of the writing process has been conducted at high volume, so I must thank the usual crew of groovy funkers: 808 State, Barry White, Level 42, MOS, New Order, Stevie Wonder, and, to a frighteningly addictive extent, Fatboy Slim; come on Norman, time for another! And I would like to offer very special thanks to two artists whose beautiful music has measured the course of my transition from overenthusiastic boy to overenthusiastic uncle, husband, and dad. First, to George Michael, for his affecting evocation and funky beats, and for proof that the slow-baked pie tastes best (something I've been assuring my editor for the last 18 months since the original deadline expired). And, notwithstanding my being one of *those* boys with "four distinguished A-level passes" (though I never did work at Jodrell Bank), to Paddy MacAloon, assuredly the greatest lyricist of the last three decades.

On the subject of my editor, I must thank Peter Gordon for skillfully and steadfastly steering me through another marathon effort and helping cajole me into "writing fewer words." Peter's assistant, Kim Boedigheimer, also deserves myriad thanks for helping marshal the relevant parties and for the tolerance she displays with respect to my frequent requests for input, advance fees, and free books. Thanks also to Elizabeth Ryan, John Fuller, Marie McKinley, and especially my patient and forgiving copyeditor with the mellifluously redolent moniker, Chrysta Meadowbrooke.

Now I must thank my reviewers, to whom I owe an enormous debt: Adi Shavit, Garth Lancaster, George Frazier, Greg Peet, Nevin :-) Liber, Pablo Aguilar, Scott Meyers, Sean Kelly, Serge Krynine, and Thorsten Ottosen. No hyperbole of mine can adequately express the debt I owe, so, as is the custom, I thank them for where they have steered me true and take sole responsibility for where I have obdurately stuck to my possibly erroneous hypotheses.

I also want to thank several people who have influenced the progress of the book in other ways: Bjarne Stroustrup, Björn Karlsson, Garth Lancaster, Greg Comeau, Kevlin Henney, and Scott Meyers. Each has offered advice and kind words that, though subtle and modest, have tremendously affected the final form of this book (and Volume 2). Many thanks.

Finally, I want to thank the users of the **Pantheios** and **STLSoft** libraries, many of whom have proffered encouragement, sage technical criticism, requests and suggestions, and even demands to get hold of prepublication copies of this book! I hope they're not disappointed by the outcome.

Parachutes: Coda

By the way, Uncle John says the third parachute jump was a breeze, so I'll take heart from that for the preparation of my next two books, which I'm going to get on with as soon as I send this one off to the publisher. See you next year!

About the Author

Matthew Wilson is a software development consultant specializing in project remediation for Synesis Software and creator of the **STLSoft** and **Pantheios** libraries. He is the author of *Imperfect C++* (Addison-Wesley, 2004), a former columnist for *C/C++ Users Journal*, and a contributor to several leading publications. He has more than 15 years of experience using C++. Based in Australia, Wilson holds a Ph.D. from Manchester Metropolitan University (UK).

Prologue

Is every language fated to push at the complexity barrier until it falls over in a heap?

—Adam Connor

If something is hard to use, I just don't use it as much.

—Melanie Krug

A Dichotomy of Character

Late in the preparation of my first book, *Imperfect C++*, I proposed this book to my editor and confidently pronounced that it would be easy, would consist entirely of readily digestible content, would take me less than six months, and would be so thin it would slide easily between two layers of abstraction. As I write this, it has been twenty months since the original deadline for a final manuscript delivery, and what I had expected to be a slim volume of 16–20 chapters has morphed into two volumes, the first of which consists of 43 chapters and intermezzos (plus another 3 included on the CD). The one promise I've largely been able to keep is that the material is eminently digestible to anyone with a reasonable level of experience in C++.

So why did I get my estimates so badly wrong? Well, it's not simply because I'm a software engineer, and our estimates should always be multiplied by at least three grains of salt. Rather, I believe it comes down to four important factors.

1. STL is not intuitive and requires a significant mental investment in order to acquire a position of comfort in using it.
2. For all its technical sophistication and brilliant cohesiveness, STL is limited in outlook and does not adequately address abstractions outside its sometimes narrow conceptual definitions.
3. The C++ language is *Imperfect*.
4. C++ is hard, but its payoff is efficiency without sacrifice of design sophistication.

In recent years, C++ has gone modern, which means it has become very powerful but also, alas, somewhat esoteric and undiscoverable. If you've written a nontrivial template library involving any kind of metaprogramming, you've likely learned a lot and gotten yourself a very powerful tool. However, it's also likely that you've created something impenetrable to all but the most ardent and cunning code explorers.

The C++ language is intended for use via extension. Apart from the small subset of applications where C++ is used as "a better C," the use of C++ revolves around the definition of types—classes, enumerations, structures, and unions—that, to a significant degree, are made to look like built-in types. For this reason, many of the built-in operators can be overloaded in C++. Thus, a vector can overload the subscript operator—`operator []()`—to look (and act) like a built-in array; any classes that can be copied (usually) define the assignment operator; and so on. But because C++ is imperfect, powerful, and highly extensible, it is particularly vulnerable to what Joel Spolsky has called the *Law of Leaky Abstractions*, which states: "All nontrivial abstractions, to some degree, are leaky." Simply, this means that successful use of nontrivial abstractions requires some knowledge of what's *under* the abstraction.

This is but one of the reasons why many C++ developers roll their own libraries. It's not just due to *Not Invented Here* syndrome; it's because you can, and often do, find yourself using a component written by a third party and find that, while you know and can use and understand, say, 80% of its functionality, the remaining 20% remains hidden in a dark pit of obfuscation. This obfuscation may be a result of complexity, nonconformity to established concepts or idioms, inefficiency, efficiency, limitation of scope, inelegance of design or implementation, poor coding style, and so on. And it can be exacerbated dramatically by practical problems with the current state of compiler technology, particularly when expressing error messages in nontrivial template instantiations.

One reason I'm able to write this book is that I've spent a *lot* of time investigating and implementing STL-related libraries, rather than accepting what was given by the C++ standardization (1998) or adopting the work of others. One reason I've chosen to write this book is to be able to pass along what I've learned in the process, not only for those of you who wish to write STL extensions but also, necessitated by the *Law of Leaky Abstractions*, for those of you who wish only to *use* STL extensions written by others—because you'll probably have to peek under the covers every now and then.

Principles of UNIX Programming

In *The Art of UNIX Programming* (Addison-Wesley, 2004), Eric Raymond formalizes the best practices of the UNIX community, derived through extensive and diverse experience, in the form of a set of rules. These will guide us in our STL adaptation enterprise and are characterized here as the following principles.

- *Principle of Clarity*: Clarity is better than cleverness.
- *Principle of Composition*: Design components to be connected to each other.
- *Principle of Diversity*: Distrust all claims of a "one true way."
- *Principle of Economy*: Programmer time is expensive; conserve it in preference to machine time.
- *Principle of Extensibility*: Design for the future because it will be here sooner than you think.
- *Principle of Generation*: Avoid hand hacking; write programs to write programs when practicable.
- *Principle of Least Surprise*: In interface design, always do the least surprising thing.
- *Principle of Modularity*: Write simple parts connected by clean interfaces.
- *Principle of Most Surprise*: When you must fail, fail noisily and as soon as possible.
- *Principle of Optimization*: Get it working before you optimize it.
- *Principle of Parsimony*: Write large components only when it is clear by demonstration that nothing else will do.
- *Principle of Robustness*: Robustness is the child of transparency and simplicity.
- *Principle of Separation*: Separate policy from mechanism; separate interfaces from engines.

- *Principle of Simplicity*: Design for simplicity; add or reveal complexity only where you must.
- *Principle of Transparency*: Design for visibility to make inspection and debugging easier.

Seven Signs of Successful C++ Software Libraries

Along with these principles, the material in this book (and in Volume 2) will be guided by the following seven signs of successful C++ software libraries: efficiency, discoverability and transparency, expressiveness, robustness, flexibility, modularity, and portability.

Efficiency

When I came out of university, having spent four years writing in C, Modula-2, Prolog, and SQL as an undergraduate and then three more years writing optical network simulators in C++ as a postgraduate, I really thought I was the cat's meow. And worse, I thought that anyone who used languages other than C or C++ was ignorant, unintelligent, or unsophisticated. The blunder of that perception is *not*, as some might assume, that I realized that I still had at least twelve years (and counting) to go until I really knew something about C++. Rather, it was that I perceived no virtue in other languages.

These days, I am a little wiser, and I recognize that the software engineering landscape is a broad one with many differing requirements. Execution time is not always an important factor, let alone the overriding one (*Principle of Economy*). When writing a system script, using Python or (perplexingly) Perl or (preferably) Ruby is a far better idea, if it takes thirty minutes, than spending three days writing the same functionality in C++ just to get 10% better performance. This is so in many cases where same-order performance differences are irrelevant. Far from the "C++ or die" attitude that I had those long years ago, these days I elect to use Ruby for many tasks; it doesn't make your cheeks laugh or your hair fall out. C++, however, remains my primary language when it comes to writing robust, high-performance software. Indeed:

> If you don't need or want to write efficient code, don't program in C++, and don't read this book.

(You should still buy it, though!)

Many would argue that there are other reasons for choosing C++, particularly `const`-correctness, strong compile-time type safety, superior facilities for implementing user-defined types, generic programming, and so on. I would agree that these are significant and much-missed aspects of some other languages, but when balancing all the issues involved in general software engineering—particularly the principles of *Composition*, *Economy*, *Least Surprise*, *Transparency*, and *Optimization*—it's clear (at least to me) that efficiency is the factor sine qua non. A few diehard opponents of the language still argue that C++ is not efficient. To those poor deluded souls, I will simply observe that if C++ is slowing you down, you're not using it correctly. A larger group argues that one can write highly efficient software in other languages. Although this holds true in several specific application areas, the notion that any other language can currently compete with C++ in efficiency *and* scope of application is pure fantasy.

Why should we be efficient? Well, the word around town is that, unless the materials scientists pull one out of the bag, we're about to run out of orders of magnitude in performance in electronic substrates. Since I never went to the School of Humble Prognostications, I shall demur from subscribing to this notion wholeheartedly ahead of incontrovertible evidence. However, even if we go on eking out further advances from our nonquantum substrates, it's a fair bet that operating systems will go on getting fatter and slower and that software will continue to accrue greater levels of complexity (for all those nontechnical reasons) and will be used in a greater spectrum of physical devices. Efficiency is important, and that is not likely to change.

You might question whether such a strong focus on efficiency runs counter to the *Principle of Optimization*. Applied willy-nilly it can certainly be so. However, libraries usually have the potential for a wide user base and long life, implying that at least some of their potential applications will have performance constraints. Hence, in addition to ensuring correctness, the authors of successful libraries must usually address themselves to efficiency.

STL was designed to be very efficient, and if used correctly it can be, as we will see in many instances throughout this book. Part of the reason for this is that STL is open to extension. But it's also quite easy to use (or extend) it inefficiently. Consequently, a central aim of this book is to promote efficiency-friendly practices in C++ library development, and I make no apology for it.

Discoverability and Transparency

While efficiency is a strong motivating factor for the use of C++, it is tempered by deficiencies in two necessary attributes of any libraries you may consider using: discoverability and transparency. Extensive definitions of these two aspects of software in general are given in Raymond's *Art of UNIX Programming*. Throughout this book, I shall use my own related definitions, as they pertain to C++ (and C) software libraries:

Definition: Discoverability is how easy it is to understand a component in order to be able to use it.

Definition: Transparency is how easy it is to understand a component in order to be able to modify it.

Discoverability is primarily the obviousness of a component's interface—form, consistency, orthogonality, naming conventions, parameter relativity, method naming, use of idiom, and so on—but also comprises the documentation, tutorials, samples, and anything else that helps the potential user to get to the "Aha" moment. A discoverable interface is easy to use correctly and hard to use incorrectly. Transparency is more closely focused on the code—file layout, bracing, local variable names, (useful) comments, and so on—although it also comprises implementation documentation, if any. The two attributes are related through different levels of abstraction: If a component has poor discoverability, code that uses it will suffer reduced transparency.

Permit me to share another personal observation: In my professional life, I've made significant commercial use of very few nonproprietary C++ libraries (and even those tend to require significant enhancement to their flexibility). For anything else for which I've needed C++, I've written my own libraries. At face value, this casts me as someone with a severe case of *Not Invented Here* syndrome: natural empiricism run amok. However, contrast this with my use of libraries in other languages. I've used literally dozens of C libraries (or libraries with C APIs) and been perfectly happy to do so. With other languages—D, .NET, Java, Perl, Python, Ruby—I use third-party libraries with few qualms. Why should this be so?

The answer has something to do with efficiency but typically a whole lot more to do with discoverability and transparency. (And that's not even touching on the failure of runtime polymorphism-based object orientation to deliver on its promises. Discussions of *that*, however, are outside the scope of this book, somewhat outside of my area of expertise, and quite outside my interest.)

Good software engineers are imbued with all manner of survival-of-the-most-fit instincts, among which is a finely honed sense of costs and benefits when it comes to assessing components for potential use. If a component promises much in terms of performance and/or features but is very poor on discoverability, that sense kicks in and you get a "bad feeling" about adopting it. Here's when the roll-your-own sentiments start to assert themselves.

Further, in cases where discoverability is adequate (or even very good), the component might still be a bad bet if it has poor transparency. Transparency is important for several reasons. First, having a simple and well-presented interface is all well and good, but you're not going to be able to effect fixes or enhancements with any degree of confidence if the implementation is a wild woman's knitting. (Or a crazy chap's crochet, if you prefer. I'd hate to needle anyone with perceived gender bias in my sewing metaphors.) Second, if the overall balance of characteristics is such that you are inclined to use the component despite its low discoverability, you may be forced to inspect the internals in order to properly understand how to use it. Third, you may be interested in learning how to implement similar kinds of libraries, which is an important characteristic of libraries in the open-source era. Finally, there's the plain commonsense aspect that if the implementation looks bad, it probably is bad, and you might rightly be skeptical about the other attributes of the software and its author(s).

For me, both discoverability and transparency are very important characteristics of C++ libraries, reflected by a strong focus on both in this book and in my code. If you check out my libraries, you'll see a clear and precise (some might say pedantic) common code structure and file layout. That's not to say all the code itself is transparent, but I'm trying, Ringo, I'm trying *real* hard.

Expressiveness

Another reason people use C++ is that it is extremely expressive (powerful).

Definition: Expressiveness is how much of a given task can be achieved clearly in as few statements as possible.

There are three primary advantages of highly expressive code. First, it affords increased productivity, since less code needs to be written and the code that is being written is at a higher level of abstraction. Second, it promotes code reuse, which engenders greater robustness by dint of the reused component implementations receiving more coverage throughout their use contexts. Third, it tends to produce less buggy software. Bug incidence tends to be largely dependent on the number of code lines, in part a function of the reduced incidence of control flow issues and explicit resource management when operating at higher levels of abstraction.

Consider the following fragment of C code to delete all the files in the current directory.

```
DIR*  dir = opendir(".");
if(NULL != dir)
{
  struct dirent*  de;
  for(; NULL != (de = readdir(dir)); )
  {
    struct stat st;
    if( 0 == stat(de->d_name, &st) &&
        S_IFREG == (st.st_mode & S_IFMT))
    {
      remove(de->d_name);
    }
  }
  closedir(dir);
}
```

I'd suggest that most competent C programmers could tell you what this code does just by looking at it. Let's say you're an experienced VMS/Windows programmer, and you've been shown this as your first piece of UNIX code. I believe you'd understand all of it straight off the bat, with the possible exception of the file attribute constants `S_IFREG` and `S_IFMT`. This code is highly transparent and suggests that the **opendir**/**readdir** API is highly discoverable, as indeed it is. However, the code is not very expressive. It is verbose and contains lots of explicit control flow, so having to repeat it, with slight modifications, each time you need to do such a thing is going to lead to copy-and-paste purgatory.

Now let's look at a C++/STL form, using the class `unixstl::readdir_sequence` (Chapter 19).

```
readdir_sequence entries(".", readdir_sequence::files);

std::for_each(entries.begin(), entries.end(), ::remove);
```

In contrast to the former example, almost every part of this code directly relates to the task being undertaken. For anyone familiar with the STL idioms of iterator pairs and algorithms, this is highly discoverable. The second line reads as follows: "`for_each` element in the range [`entries.begin()`, `entries.end()`), `remove()` it." (This uses the half-open range notation—a leading square bracket and a trailing parenthesis—to describe all the elements in the range starting from `entries.begin()` up to, but not including, `entries.end()`.) Even absent any prior knowledge or documentation of the `readdir_sequence` class, the code is transparent (implying good discoverability of `readdir_sequence`) as long as the reader knows or can guess what "readdir" might be.

Nonetheless, there are disadvantages to high levels of expressiveness. First, too much abstraction is the enemy of transparency (and, potentially, discoverability). This is one reason why abstractions should be kept at a relatively low level: too much and the transparency of the system as a whole will suffer, even when that of a given level is good. Second, because a potentially large amount of work is being done in a small number of statements, the performance costs can mount up: In general, it is also important that the underlying code is efficient. Third, users of the component are limited to anticipated features of its abstraction. In this case, the directory sequence provides flags for filtering files and/or directories but does not do so based on file attributes or file size. In higher-level abstractions, the seemingly arbitrary provision of some functionality and not others can be troublesome. (And everybody always wants something different!)

For STL components, there are two further problems. First, many STL libraries, including several standard library implementations, are poorly written for readability—adversely affecting transparency. Runtime bugs can be extremely hard to eke out. And the situation with compile-time bugs is as bad or worse. The parlous state of template instantiation error messages on even the latest C++ compilers means that both discoverability and transparency suffer greatly when things go wrong. Error messages for even relatively simple cases tend to be impenetrable. One important aspect of writing STL extension libraries, therefore, is anticipating these and adding nonfunctioning code to reduce the user's consternation in such cases. We'll see several examples of this throughout the component implementations described in the book.

Second, the clear gains in expressiveness with STL as shown in the example above are not available in every case. Often, a suitable function is not available, requiring either the writing of a custom function class, which reduces expressiveness (since you have to code separate out-of-scope function classes), or the use of function adaptors, which reduces discoverability and transparency. We'll see in Volume 2 more advanced techniques for dealing with these cases, but none represent a free ride in terms of efficiency, discoverability and transparency, flexibility, and portability.

Robustness

If something doesn't work, it's not likely to be successful. C++ is sometimes accused of being a language that too easily facilitates bad practice. Naturally, I don't agree, and I contrarily assert that, with appropriate discipline, C++ programs can be extremely robust. (My favorite real-world paying-job activities are writing network servers and having them run for years without experiencing errors. The downside of this, of course, is that I miss out on those fat remediation contracts. Here "Down Under," my software has carried billions of dollars' worth of transactions across the continent without a flicker. And I've never had the opportunity to charge cash money for fixing them, which is, you must admit, as it should be.)

The measures required for robust software are compounded by the use of templates, since the compiler completes its checking only when instantiating a template. Thus, bugs in template libraries may reside undetected by the compiler for extended periods of time, only precipitated by certain specializations. To ameliorate this situation, C++ libraries, and especially template libraries, should make liberal use of contract programming enforcements (Chapter 7) to detect invalid states and of constraints (Chapter 8) to prevent instantiation of proscribed specializations.

The principles of *Clarity*, *Composition*, *Modularity*, *Separation*, and *Simplicity* all apply to robustness; it is featured strongly in this book.

Flexibility

The principles of *Least Surprise* and *Composition* suggest that components should be written to work together in a way that matches the expectations of users. Templates offer a lot of promise in this regard, since we can define functionality that will apply to instantiations of arbitrary types. A classic example of this is the `std::max()` function template.

```
template <typename T>
T max(T const& t1, T const& t2);
```

Such genericity as is afforded by this function template is easily achieved and easily understood. However, it is not so hard to confound the intent of the template.

```
int   i1 = 1;
long  l1 = 11;
max(i1, l1);    // Compile error!
```

Some inflexibilities can be more profound. Consider the case where you wish to load a dynamic library, using a class (in this case, the notional class `dynamic_library`). You hold the path name in a C-style string.

```
char const*     pathName = . . .
dynamic_library dl(pathName);
```

If you now find yourself wanting to instantiate an instance of `dynamic_library` with an instance of `std::string`, you're going to have to change two lines, even though the logical action has not changed.

```
std::string const&  pathName = . . .
dynamic_library     dl(pathName.c_str());
```

This violates the *Principle of Composition*. The `dynamic_library` class should instead be coded to work seamlessly with string objects as readily as with C-style strings. Not doing so causes users unnecessary inconvenience and leads to cluttered code that is brittle with respect to change.

Modularity

Failure to achieve modularity leads to bloated and fragile monolithic frameworks, with unhappy users, poor performance and robustness, and limited power and flexibility. (And we won't even mention the compilation times!) Because C++ uses static type checking and inherits C's declaration/inclusion model, it is relatively easy to find oneself in situations of undue coupling. C and C++ support forward declaration of types, but this is viable only when types are used by a pointer or by a reference, rather than by a value. Since C++ is a language that promotes the use of value semantics, this can be something of a contradiction.

Modularity is an area where template libraries can shine, if managed correctly. Because compilers use *Structural Conformance* (Section 10.1) to establish whether a given instantiation is valid, it is possible to write code to work with a set of types conforming to a given concept without any a priori inclusion visibility of the types. The STL is a leading example of this, and other successful libraries follow suit.

Portability

Except where convinced of the long-term viability of the current context—architecture, operating system, compiler, compiler settings, standard/third-party libraries, and so on—in which a library is used, authors of successful libraries should concern themselves with portability. Exceptions are extremely rare, which means that in practice almost all authors should expend effort to avoid obsolescence of their libraries.

You need look no further than the system headers of your favorite operating system to see the evidence of a failure to account for portability. But it's not as easy as you might think; otherwise a great many smart engineers would not have been wrong-footed. Writing portable code relies on having a constant awareness of the assumptions on which you are working. These range from the obvious, such as architecture and operating system, to the subtle and challenging, such as the versions of libraries, and even what bugs they might have and the established workarounds for them.

Another area of portability is in the dialect of C++ being used. Most compilers offer options to turn various C++ language features off or on, effectively providing dialects, or subsets, of the language. For example, very small components are sometimes built without exception-handling support. With care (and effort!), portability can meaningfully be extended to cover such cases, as we'll see with several components throughout this book.

STL extensions by nature need a high degree of portability, from handling different operating systems right through to handling compiler bugs and language dialects, so this characteristic also receives a strong focus throughout the book.

Balancing the Signs: Satisfiction, Dialecticism, and Idioms Old and New

It should be no great surprise to learn that very few libraries can be said to score highly on all seven signs of success. Only very small libraries with very tightly defined functionality are likely to do so. For all the rest, the challenge in writing such libraries lies in finding the appropriate balance. In this section, I will attempt to outline my strategy for achieving successful balances.

As with most things to those whose pragmatism manages to overcome their dogmatism, there is no single clear answer. The power of STL cannot be denied, but neither can its ability to obfuscate, nor its tendency to draw the unwary into unmaintainable, inefficient, unportable, or simply incomprehensible coding. There are several mechanisms at work. If you're writing C++ template libraries, it's very easy to fall into your own style and techniques, creating your own private idioms. This is **dialecticism**, and it is a hindrance to communication between engineers.

From the perspective of a library user, it's not uncommon to find yourself overcoming initial discoverability hurdles of such dialecticism, or even plain bad design, and find yourself in a position of relative comfort. Now you're lodged in one of a potentially infinite number of local maxima of effectiveness. It's possible that you're using the best library for the job, and/or using that library most effectively, but it's also possible that you could be far more effective using another library or using the given library in a different way. This situation is known as satisfice or satisficing—you're **satis**fied with what (currently) suf**fice**s. I like to use the pleasingly nuanced term **satisfiction**. Satisfiction can lead into dialecticism, ignorance of good idiom, or both. But we engineers do not have infinite time in which to research the best mechanisms for doing our work, and the software tools we use are increasingly large and complex, so it is practically impossible to avoid satisfiction.

Each of us scratches out a little patch on the complexity plane, within which we, as smart human beings, become quickly comfortable. Once we're comfortable, our work is no longer complex and off-putting to us and may become adopted by others attracted by power, efficiency, and flexibility and repulsed by coupling, lack of portability, and lack of discoverability. The degree to which that work then propagates is a mix of marketing and technical merit. Once it gets far enough, it becomes accepted and then treated as normal and simple, even when it isn't—probably the best example we have of this is the STL itself.

Whether as library writers, users, or both, we need mechanisms to survive dialecticism and satisfiction. Such mechanisms include idioms, anti-idioms, and peeks inside the black box. Experienced practitioners often forget that idioms are not innate actions amenable to intuition. The spellings of many words in the English lexicon are far from intuitive. The use of a mouse to click buttons, menu items, and scroll bars is *not* intuitive. And no part of the STL, and not much of the rest of C++, is innately intuitive.

For example: C++ makes all classes copyable (*CopyConstructible* and *Assignable*) by default. In my opinion, and that of others, this is a mistake. And the protection required of programmers for any or all classes that need to be *noncopyable* is not obvious by any means.

```
class NonCopyable
{
public: // Member Types
  typedef NonCopyable   class_type;
  . . .
private: // Not to be implemented
  NonCopyable(class_type const&);
  class_type& operator =(class_type const&);
};
```

This is a widely recognized practice, so much so that it is part of the common lore of C++. It is an idiom. Similarly, STL is one gigantic idiom. Once you are experienced in STL, using the basic STL provided by the standard library becomes idiomatic. Where STL extension introduces new concepts and practices, it requires new idioms to tame them.

Just as there are valid idioms, old and new, so can there be anti-idioms. STL practitioners must be alert to avoid these. For example, treating an iterator as a pointer is an anti-idiom, one that is likely to hurt you down the track.

Identifying and reinforcing established idioms, describing useful new idioms, warning against faux/anti-idioms, and looking inside the black box are the tactical measures of this book (and Volume 2), with which we navigate the balance of the seven signs of successful C++ software libraries.

Example Libraries

Since I'm a practical kind of chap, I like to read books that impart knowledge to me from real experience. (This is doublespeak for "My brain turns to fudge when asked to inhabit the abstract plane.") Therefore, most of the examples in the book are drawn from my own work, in particular from a small set of open-source projects. Cynics among you may think this is just a weak ploy to popularize my libraries. That may be a small part of it, I suppose, but there are serious reasons as well. First, in addition to making sure that I know what I'm talking about—something of passing importance in writing a book—using my own work allows me to discuss the mistakes involved in all their manifest dreadfulness without offending anyone or incurring any lawsuits. And there is no impediment to you, gentle reader, in getting your hands on the libraries to see if I've been full and honest in my prognostications herein.

STLSoft

STLSoft is my big baby, brought kicking and screaming into the C++ tropopause over the last half decade or so. It is free, purports to be portable (between compilers and, where appropriate, between operating systems), easy to use, and, most important, efficient. Like all my open-source libraries, it uses the modified BSD license.

Two features have ensured that **STLSoft** is easy to use and, in particular, easy to extend. First, it's 100% header-only. All you have to do is make the appropriate include and ensure that the **STLSoft** include directory is in your include path. The second feature is that the level of abstraction is kept deliberately low (*Principle of Parsimony*), and mixing of technologies and operating system features is avoided or kept to an absolute minimum (*Principle of Simplicity*). Rather, the libraries are divided into a number of subprojects that address specific technology areas.

Although it offers many useful features for STL and non-STL programming alike, the main purpose of **STLSoft** is to provide general-purpose components and facilities, at a moderately low level of abstraction, to support commercial projects and other open-source libraries. It is high on efficiency, flexibility, and portability and low on coupling. Where compromises must be made, expressiveness and abstraction richness are sacrificed to preserve these characteristics.

STLSoft Subprojects

The main subproject is, somewhat confusingly, called the **STLSoft** subproject. Most platform- and technology-independent code lives here. There are allocators and allocator adaptors (see Volume 2), algorithms (see Volume 2), iterators and iterator adaptors (discussed in Part III), memory utilities, string manipulation functions and classes (Chapter 27), classes for defining (time/space-efficient) C++ properties (see Chapter 35 of *Imperfect C++*), metaprogramming components (Chapters 12, 13, and 41), shims (Chapter 9), compiler and standard library feature discrimination (and substitution), and lots more. **STLSoft** components live within the `stlsoft` namespace.

The three biggest subprojects are **COMSTL**, **UNIXSTL**, and **WinSTL**, whose components reside in the `comstl`, `unixstl`, and `winstl` namespaces, respectively. **COMSTL** provides a host of utility components for working with the **Component Object Model** (**COM**) and also provides STL-compatible sequence adaptors over the *COM enumerator* and *COM collection* concepts; these are described in Chapters 28 and 30, respectively. **COMSTL** also supports one of my new libraries, **VOLE** (also 100% header-only; included on the CD), which provides a robust, succinct, and compiler-independent way to drive **COM** automation servers from C++.

UNIXSTL and **WinSTL** provide operating-system- and technology-specific components for UNIX and Windows operating systems, several of which we will see in Parts I and II. They have a number of structurally conformant components, such as `environment_variable`, file_path_buffer (Section 16.4), filesystem_traits (Section 16.3), memory_mapped_file, module, path, performance_counter, process_mutex, and thread_mutex. These are drawn together, along with some brand-new components, such as environment_map (Chapter 25), into the platformstl namespace, comprising the **PlatformSTL** subproject. Note that this approach is starkly different from one that abstracts away operating system differences: Only those components that are sufficiently structurally conformant to facilitate platform-agnostic coding without substantial intrusion of the preprocessor are allowed into **PlatformSTL**.

The other subprojects address additional technology-specific areas. **ACESTL** applies STL concepts to some components from the popular **Adaptive Communications Environment** (**ACE**) library (Chapter 31). **MFCSTL** attempts to make the aged **Microsoft Foundation Classes** (**MFC**) look more STL-like, as we see in Chapter 24 with the std::vector-like CArray adaptors. **RangeLib** is the **STLSoft** implementation of the range concept; ranges are covered in Volume 2. **ATLSTL**, **InetSTL**, and **WTLSTL** are all smallish projects that enhance **ATL**, Internet programming, and **WTL** in an STL way.

Although each **STLSoft** subproject (apart from the **STLSoft** main project, which resides in stlsoft) has its own apparently distinct top-level namespace, these namespaces are actually aliases for nested namespaces within the stlsoft namespace. For example, the comstl namespace is actually defined in the **COMSTL** root header *<comstl/comstl.h>*, as shown in the following code.

```
// comstl/comstl.h
namespace stlsoft
{
  namespace comstl_project
  {
    . . . // COMSTL components
  } // namespace comstl_project
} // namespace stlsoft

namespace comstl = ::stlsoft::comstl_project;
```

All other **COMSTL** components, defined in other **COMSTL** header files (all of which include *<comstl/comstl.h>*), define their components within the stlsoft::comstl_project namespace but are seen by all client code as residing within the comstl namespace. The payoff here is that all components within the stlsoft namespace are automatically visible to components within the putative comstl namespace, which saves a lot of typing and distracting namespace qualifications. The same technique is used for all other subprojects.

Tip: Use namespace aliasing to allow namespace hierarchies to be used with minimal syntactic intrusion on client code.

Boost

Boost is an open-source organization whose focus is the development of libraries that integrate with the standard library and may be proposed as future contributions to the standard. It has a large contributor base, including several members of the C++ standards committee.

I'm not a **Boost** user or contributor, so **Boost** components aren't covered in detail in Volume 1; only the `boost::tokenizer` component is discussed (Section 27.5.4). If you want to learn how to *use* **Boost**, you should check out the book *Beyond the C++ Standard Library: An Introduction to Boost* (Addison-Wesley, 2005), written by my friend Björn Karlsson.

Open-RJ

Open-RJ is a structured file reader for the Record-JAR format. It includes mappings to several languages and technologies, including COM, D, .NET, Python, and Ruby. In Chapter 32, I describe a general mechanism for emulating in C++ the flexibility in the semantics of the subscript operator in Python and Ruby, using the **Open-RJ/C++** `Record` class.

Pantheios

Pantheios is a logging library for C++ that is 100% type-safe, generic, extensible, thread-safe, atomic, and *exceedingly* efficient: You pay only for what you use, and you pay only once. **Pantheios** has a four-part architecture, comprising core, front end, back end, and an application layer. The application layer uses **STLSoft**'s string access shims (Section 9.3.1) to provide infinite genericity and extensibility. The core aggregates all component parts of a logging statement into a single string and dispatches to the back end, which may be one of a set of stock back ends, or a custom back end of the user's own. The front end arbitrates which message severities are processed and which skipped; a custom front end may be used. The implementation of the **Pantheios** core is discussed in Chapters 38 and 39, which demonstrate how iterator adaptors can be used to apply algorithms to custom types.

recls

The **recls** library (**rec**ursive **ls**) is a multiplatform recursive file system searching library, written in C++ and presenting a C API. Like **Open-RJ**, **recls** includes mappings to several languages and technologies, including COM, D, Java, .NET, Python, Ruby, and STL. A **recls** search involves specifying a search root, a search pattern, and flags that moderate the search behavior, as shown in Sections 20.10, 28.2, 30.2, 34.1, and 36.5. Like **Pantheios**, it uses many **STLSoft** components, including `file_path_buffer`, `glob_sequence` (Chapter 17), and `findfile_sequence` (Chapter 20).

Presentation Conventions

What anyone thinks of me is none of my business.
—Peter Brock

Me fail English? That's unpossible.
—Ralph Wiggum, *The Simpsons*

Most presentation conventions in the book are self-evident, so I'll just touch on those that warrant a little explanation.

Fonts

The fonts and capitalization scheme discriminates between the following types of things in the body text: **API** (e.g., the **glob** API), `code`, *Concept* (e.g., the *shim* concept), *<headername.hpp>*, **Library** (e.g., the **ACE** library), *literal* or *path* (e.g., *"my string"*, *NULL*, *123*, */usr/include*), ***Pattern*** (e.g., the ***Façade*** pattern), *Principle* (e.g., the *Principle of Least Surprise*), **shim** (e.g., the **get_ptr** shim). Code listings use the following conventions:

```
// In namespace namespace_name
class class_name
{
  . . . // something that's a given, or has been shown already
public: // Class Section Name, e.g., "Construction"
  class_name()
  {
    this->something_emphasized();
    something_new_or_changed_from_previous_listing();
  }
  . . .
```

. . . versus ...

. . . denotes previously seen code, boilerplate code, or previously seen or yet-to-be-specified template parameter lists. This is not to be confused with ... , which denotes an ellipsis, as used in variadic function signatures and catchall clauses.

End Iterator Precomputation

To keep the code snippets digestible, I've shown handwritten iterator loops without using end iterator precomputation. In other words, the book shows listings such as:

```
typedef unixstl::readdir_sequence   rds_t;
rds_t   files(".", rds_t::files);

for(rds_t::const_iterator b = files.begin(); b != files.end(); ++b)
{
  std::cout << *b << std::endl;
}
```

rather than what I would normally write:

```
. . .
for(rds_t::const_iterator b = files.begin(), e = files.end(); b != e;
++b)
{
  std::cout << *b << std::endl;
}
```

This is likely to be more efficient in many cases (and will certainly not be less in any). The same can be said of evaluating the `size()` of a collection once. So, despite the fact that you won't see it anywhere else in this book, I suggest the following:

Tip: Prefer to precompute the end iterator when enumerating iterator ranges. Prefer to precompute the size of a collection when using indexing.

Of course, the best way to precompute the endpoint iterator is to use an algorithm, where possible, as in the following:

```
std::copy(files.begin(), files.end()
        , std::ostream_iterator<rds_t::value_type>(std::cout, "\n"));
```

Nested Class Type Qualification

I also take a few shortcuts in qualification of types where the context is unambiguous. For example, rather than writing out the full return type of `Fibonacci_sequence::const_iterator::class_type&` for the `Fibonacci_sequence::const_iterator::operator ++()` method, I'll just write it as follows:

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  . . .
```

Should you wish to compile code directly from the book, you will need to bear these expediencies in mind. Thankfully, I'm including all the test files on the accompanying CD, so you shouldn't need to go to the effort of transcription from the text.

NULL

I use *`NULL`* for pointers, for two reasons. First, despite anything you might have heard to the contrary, there *is* a strongly typed *`NULL`* available for C++ (as described in Section 15.1 of *Imperfect C++*). Second, I believe that being all modern and using *`0`*, merely because *`NULL`* has no more meaning to the compiler, is just plain nuts. Code is for people first, compilers second.

Template Parameter Names

Template parameters are one- or two-letter capitals, for example, `T` (type/value-type), `S` (sequence/string), `C` (container/character), `VP` (value policy type), and so on. A few are shown in full, for example, `CHOOSE_FIRST_TYPE`, but in code they are one or two letters. There are two reasons for this. First, many all-uppercase words are `#define`d in third-party libraries and application code. Experience such a conflict once and you'll take big steps not to do so again, I promise you! Conversely, we may assume that instances of one (and maybe two) character letters are not `#define`d. If you find yourself using any library that does `#define` symbols with one or two letters, throw it away immediately in all good conscience.

The second reason is that it is not valid to create a member type with the same name as a template parameter. In other words, the following is illegal.

```
template <typename iterator>
struct thing
{
  typedef iterator    iterator; // Compile error
};
```

I strongly believe, therefore, that:

```
template <typename I>
struct thing
{
  typedef I           iterator;
};
```

is much clearer than:

```
template <typename Iterator>
struct thing
{
  typedef Iterator    iterator;
};
```

Almost without exception, I immediately use the short template parameter names to define member types, and they are *not* used further throughout the remainder of the template class definition. This is especially significant because it makes obvious the absence of member types, many of whose absence can lie unseen for a long (and, thereby, ultimately vexing) time. By having just a single letter, the class template definition must immediately busy itself defining all these member types, which is a *good thing*. If you don't agree, that's your prerogative. But I assure you that this is the most survivable scheme I've come across.

Member and Namespace-Scope Type Names

Member types are named `xyz_type`, except for established/standard names, such as `iterator`, `pointer`, and so on; local/namespace-scope types are named `xyz_t`.

Calling Conventions

Where they are mentioned, the three common Windows calling conventions are referred to as **cdecl** (equivalent to the calling convention used on UNIX), **fastcall**, and **stdcall**. Unless specified otherwise, all COM methods and all Windows API functions are **stdcall**, and everything else is **cdecl**, though I will always mention the calling convention when it pertains to the discussion. Where a calling convention is indicated, it is using the Microsoft compiler extension keywords `_cdecl`, `_fastcall`, and `_stdcall`; other compilers may use different keywords.

Endpoint Iterators

The C++ standard refers to the iterators returned from `end()` and `rend()`, and their equivalents, as being "past-the-end values." I use the term **endpoint iterators** for clarity and consistency. After all, these functions are not named `past_the_end()` and `rpast_the_end()`. When discussing an iterator reaching the end of its traversal, I will refer to it as having reached the `end()` point, that is, the state at which it will compare equal with the iterator returned by the collection's `end()` method.

Namespace for Standard C Names

I do not place types, function names, and other reserved names from the C standard library in the `std` namespace. This saves space in the code examples, but it also corresponds to my own practice. It's simply never going to happen that `size_t` or `strlen` will be removed from the global namespace, so to write `std::size_t` and `std::strlen` in their stead is gratuitous and distracting. (To me, anyway; other authors may have different views.)

Class Adaptors and Instance Adaptors

Some of the patterns literature uses the terms ***Adaptor*** and ***Object Adaptor*** when referring to, respectively, a compile-time adaptation of a type and a runtime adaptation of an instance of a type. Neither term pleases me much; the former is too general, the latter uses the word *object*, which is *way* overused in software development literature. I prefer the terms ***Class Adaptor*** and ***Instance Adaptor***, and that's what I'll be using throughout this book.

Header File Names

Header files are always mentioned with surrounding angle braces. This is to avoid ambiguity when mentioning the (stupidly named) extensionless standard header files. In other words, *`<algorithm>`*, rather than the more ambiguous *`algorithm`*, refers to the header file.

This page intentionally left blank

PART ONE

Foundations

The 16 mini-chapters in this part contain essential foundation material that is used throughout Parts II and III (and in Volume 2). The material includes the following:

- STL (and extended STL) concepts
- Useful concepts, laws, practices, and principles that apply to the practice of STL extension
- Specific template tools and techniques on which this book draws
- General-purpose components used throughout the implementations of the components described in Parts II and III

Chapter 1 offers a brief description of the Standard Template Library and in particular the *container* (Section 1.2) and *iterator* (Section 1.3) concepts. As demonstrated throughout this book, these concepts are insufficient in scope to encompass the world of STL extensions; Chapter 2 describes an expanded set of STL concepts, in particular that of *STL collection* (Section 2.2). Chapter 3 describes the important new concept of *element reference category*, which is required to suitably describe (and robustly manipulate) those collections that do not own their own elements. Chapter 4 describes a little-known feature of C++ that impacts, both positively and negatively, the use of STL collections whose element reference category is *by-value temporary* (Section 3.3.5).

The following seven chapters cover issues of C++ programming in general and STL extension in particular: the *DRY SPOT* principle (Chapter 5); the *Law of Leaky Abstractions* (Chapter 6); contract programming (Chapter 7); constraints (Chapter 8); the *shim* concept (Chapter 9); the Duck Rule, the Goose Rule, and conformance (Chapter 10); and C++'s principle potent resource management mechanism, *Resource Acquisition Is Initialization* or RAII (Chapter 11). Skipping ahead for a moment, Chapter 14 describes the problem of the decreasing returns in discoverability for the (supposed) increasing power in additional template arguments. This problem will crop up in several extensions throughout Parts II and III.

Chapter 12 describes the traits, mini-traits, and other small template tools used throughout the rest of the book. Chapter 13 discusses a much more significant, general-purpose mechanism for type inference and management, which is used to “fix” incomplete or functionally limited types or to define appropriately limited functionality in adapted collection and iterator types. Chapter 15 describes a simple technique for avoiding compiler-specific behavior—compile errors—by implementing nonmember operators in terms of public member comparison functions.

We finish Part I with Chapter 16, which describes several components (from the **STLSoft** libraries) used in the implementations of the collections and iterator adaptors described in Parts II and III.

This page intentionally left blank

CHAPTER 1

The Standard Template Library

So far, C++ is the best language I've discovered to say what I want to say.
—Alexander Stepanov

Every revolution is unthinkable beforehand, and obvious afterwards.
—Kevin Bealer

Writing an introduction to the Standard Template Library (STL) would be a book in itself, and that's not the book I'm writing. There are several such books (some listed in the Bibliography), and I recommend that you have at least one of them in your head, or at hand, before you start reading this one.

1.1 Core Concepts

The STL is based around six core concepts: containers, iterators, algorithms, function objects, adaptors, and allocators. *Containers* store objects. An *iterator* is a data-structure-independent abstraction used to access elements and traverse element ranges. *Algorithms* use iterators to operate on element ranges generically, independent of the types of elements and data structures within which they reside. *Function objects* define generic operations that may be applied to the elements manipulated by algorithms. *Adaptors* allow disparate types to conform to existing concepts by changing their interfaces, including ***Class Adaptors*** (those that adapt types) and ***Instance Adaptors*** (those that adapt instances of types). *Allocators* provide an abstraction of the memory allocation and object construction operations carried out by containers.

The use of the term *STL* is somewhat muddled in the C++ world. Strictly speaking, the STL refers to the original six-concept generic template library developed by Stepanov and his collaborators in the 1980s and 1990s. The use of the term *STL* as we know it today is actually merely a colloquialism for the subset of the C++ standard library that deals with the six concepts. The significance of this is twofold. First, there are plenty of C++ purists who will relish every opportunity to jump on you and correct your (mis)use of the term *STL*, pointing out that you should refer to the standard library. Second, there are other STL-compatible things in the standard library that do not appear in the original STL, most notably the **IOStreams**. I make use of the **IOStreams**' STL-ish characteristics for pedagogical purposes throughout the book and will cover a particular case of STL extension that is **IOStreams**-focused (Chapter 34). However, I will overwhelmingly use the term *STL* throughout because of its concision, because we're not going to pay much attention to standard library components that do not intersect with the original STL's contents, and because I wish to reserve the term *standard* for when I refer to components that are in the standard library or that conform to the requirements of concepts as spelled out in the C++ standard (C++-03).

1.2 Containers

The standard provides four sequence containers and four associative containers. It also defines three sequence container adaptors.

1.2.1 Sequence Containers

The sequence containers (C++-03: 23.2) maintain a strict linear arrangement of their elements. The three common sequence container class templates are `deque`, `list`, and `vector`. The standard (C++-03: 23.1.1;2) recommends that `list` should be preferred when frequent insertions and deletions take place in the middle of the sequence, `deque` should be preferred when frequent insertions and deletions take place at the beginning or end of the sequence, and `vector` should be preferred otherwise. The fourth container class, `basic_string`, is primarily for representing character strings, but it is a bona fide sequence container, so you can, should you be so perverse, use it to store types other than `char` and `wchar_t`.

The three sequence container adaptor class templates, `queue`, `priority_queue`, and `stack`, are ***Class Adaptors***, in that they may be used to adapt any (sequence container) type that is *CopyConstructible* (C++-03: 20.1.3; after copy construction, the copy is equivalent to the original) and provides the required small number of operations. (Instances of a type `T` that meets the *CopyConstructible* requirement can be copy constructed from [`const` and non-`const`] instances of `T`.)

For example, the `stack` adaptor requires only the `empty()`, `size()`, `back()`, `push_back()`, and `pop_back()` methods of the adapted type in order to implement its `empty()`, `size()`, `top()`, `push()`, and `pop()` methods.

The standard does not provide any container ***Instance Adaptors***; we'll meet one of these in Chapter 24.

1.2.2 Associative Containers

The associative containers (C++-03: 23.3) provide lookup of elements based on a key. The four associative containers are `map`, `multimap`, `set`, and `multiset`. `map` and `multimap` provide an association between a key and a mapped type. For example, the specialization `std::map<std::string, int>` provides a mapping between `std::string` and `int`; the specialization of `std::set<std::wstring>` represents a collection of unique values of type `std::wstring`.

1.2.3 Contiguity of Storage

Unlike every other container in the standard library, `vector` is the only one guaranteed to offer contiguous storage, so that it is compatible with C APIs. In other words, for any nonempty instance, this is well formed:

```
extern "C" void sort_ints(int* p, size_t n);

std::vector<int>  v = . . .
assert(!v.empty());
sort_ints(&v[0], v.size());
```

But this is not:

```
std::deque<int>   d = . . .
assert(!d.empty());
sort_ints(&d[0], d.size()); // This is a crash waiting to happen
```

And, for that matter, neither is the following:

```
std::string       s1("abc");
char              s2[4];
assert(!s1.empty());
::strcpy(&s2[0], &s1[0], s1.size()); // Bad day . . . eventually
```

This can be a surprise to some users of the `std::string` (or `std::wstring`), particularly since there are no known noncontiguously storing standard library string implementations. Nonetheless, it is so, and you must not forget.

1.2.4 swap

All standard containers provide a constant-time `swap()` method, which enables two instances of the same type to exchange their internal state by a simple exchange of member variables.

```
std::vector<int>  vec1(100);
std::vector<int>  vec2;

vec1.swap(vec2);

assert(0 == vec1.size());
assert(100 == vec2.size());
```

The `swap()` methods are also guaranteed to not throw exceptions, which facilitates writing exception-safe components composed in terms of the standard containers. Many other classes (standard and otherwise) also provide `swap()` for this reason.

1.3 Iterators

The iterator concept is modeled on C/C++ pointers. Generally speaking, an iterator may be incremented, dereferenced, and compared. Certain iterators may also be decremented, and some are even subject to pointer arithmetic. The set of meaningful pointer-like operations to which an iterator instance may be meaningfully subjected defines its refinement. The standard library currently defines five refinements (also known as categories) of the concept: input iterator, output iterator, forward iterator, bidirectional iterator, and random access iterator. These are also sometimes referred to individually as concepts, that is, you might refer to the bidirectional iterator concept.

1.3.1 Input Iterators

The most basic nonmutating iterator refinement is the *input iterator*, whose types support a limited set of pointer operations (C++-03: 24.1.1). Consider the instance `ii` of a type `II` that models the input iterator category. We may default construct it:

```
II  ii;
```

We may copy construct it:

```
II  ii2(ii);
```

We may copy assign it:

```
II  ii3;

ii3 = ii2;
```

We may increment it:

```
++ii2; // Preincrement, or
ii3++; // Postincrement
```

We may compare it to another instance of the same type (obtained from the same container):

```
if(ii == ii2)
{}
if(ii3 != ii)
{}
```

As long as it does not refer to the endpoint, we may dereference it to obtain an instance of the iterator type's value type, `V`:

```
II  end = . . . // Obtain the endpoint iterator value
if(end != ii)
{
  V v1  = *ii; // Dereference ii to yield type convertible to V
}
```

Equivalent instances, if dereferenceable, should yield identical values:

```
II  ii5 = ii;
if(end != ii)
{
  assert(*ii5 == *ii);
}
```

One very important feature of input iterators is that "algorithms on [them] should never attempt to pass through the same iterator twice; they should be *single pass* algorithms" (C++-03: 24.1.1;3). This means that the second of the following equivalence invariants *cannot* be asserted:

```
II  ii6 = ii;

assert(ii == ii6); // Valid
++ii;
++ii6;
assert(ii == ii6); // Not valid!
```

The means by which specific types support this requirement differ. Some may independently elicit the values from the underlying collection, whereas others may share state, as we'll see in Part II.

It's important to realize that the single-pass nature applies to all copies obtained from the source iterator. In the following code, in which `II` is an input iterator type, the sizes of `n1` and `n2` or `n1` and `n3` can never be equivalent (except in the case that `n1` is *0*) because `n2` and `n3` will *always* be *0*. This is because the three iterator instances `ii1`, `ii7`, and `ii8` share state.

```
II  ii7 = ii1;
II  ii8 = ii1;
int n1  = std::distance(ii7, end);  // Iterate range [ii7, end)
int n2  = std::distance(ii8, end);  // ii8 already == end
int n3  = std::distance(ii1, end);  // ii1 already == end

assert(0 == n2);
assert(0 == n3);
```

1.3.2 Output Iterators

The most basic mutating iterator refinement is the *output iterator*, whose types support a limited set of pointer operations (C++-03: 24.1.2). Consider the instance `oi` of a type `OI` that models the output iterator category. As with input iterators, we may default construct it, copy construct it, copy assign it, pre- and postincrement it, and compare it. Unlike input iterators, however, its dereference does not yield a value; rather, it evaluates to an *lvalue*, that is, something that may serve as the left-hand side of an assignment statement. Thus, an output iterator dereference is the mechanism by which the value is written out, as in:

```
OI  oi  = . . . // Obtain start of writable range
OI  end = . . . // Obtain endpoint iterator value

for(; end != oi; ++oi)
{
  *oi = V(); // Assigns result of V() to *oi
}
```

1.3.3 Forward Iterators

The *forward iterator* refinement combines the input and output iterator refinements, with the important additional requirement that types modeling this refinement must be able to take part in multipass algorithms. This means that a copied iterator must be able to provide an identical range as the instance copied. In the following code, in which `FI` is a forward iterator type, the sizes of `n1`, `n2`, and `n3` will be equivalent. This is because the three iterator instances `fi1`, `fi2`, and `fi3` have independent state.

```
FI  fi1 = . . .
FI  fi2 = fi1;
FI  fi3 = fi1;
FI  end = . . .
int n1  = std::distance(fi1, end); // Iterate range [fi1, end)
int n2  = std::distance(fi2, end); // Iterate range [fi2, end)
int n3  = std::distance(fi3, end); // Iterate range [fi3, end)
assert(n1 == n2);
assert(n1 == n3);
```

This has important ramifications when you are implementing STL extensions. To be able to support forward iterator semantics, a type must be able to access the same elements from the underlying range from separate (related) instances. As we'll see, when adapting some classes of APIs to the STL, providing forward iterator semantics can involve significant effort, sometimes requiring copying a nontrivial amount of state information between copied instances (see Chapter 28). We'll also see cases where iterator copies have independent state but are not able to guarantee to enumerate an identical range of values (see Chapter 26).

1.3.4 Bidirectional Iterators

The *bidirectional iterator* refinement incorporates all the requirements of the forward iterator refinement and also provides decrement operations for reversing its traversal of its underlying range.

```
BI  bi    = . . .
BI  end   = . . .
BI  begin = . . .
```

```
for(bi = begin; bi != end; ++bi) // Forwards
{}
// At this point bi == end. Now we traverse the range backwards.
for(; ; —bi)
{
  if(bi == begin)
  {
    break;
  }
}
```

In order for a container to be reversible, its iterator(s) must support at least the bidirectional refinement. It is important to note that, as shown in the example, an endpoint instance of a bidirectional iterator type must be decrementable and therefore must maintain information as to how to "get back on" the underlying range. Consequently, unlike input and forward iterator types, a default-constructed instance of a bidirectional iterator type cannot be equal to the endpoint instance. We'll look at the ramifications of this in Chapters 23, 26, and 36.

1.3.5 Random Access Iterators

The *random access iterator* refinement combines the requirements of the bidirectional iterator refinement but also supports pointer arithmetic. This means that you can subscript or offset (via addition and subtraction) a random access iterator instance.

```
RI  ri  = . . .
RI  ri2 = ri + 1;
RI  ri3 = ri - -1;
assert(ri2 == ri3);
V   v1  = ri[1];
V   v2  = *ri2;
assert(v1 == v2);
```

It is important to realize that even though pointers satisfy the requirements of the random access refinement, the reverse does not hold: Random access iterators are not pointers, even though they share most syntactic constructs. The crucial thing to remember is that the following relation holds true for pointers but not necessarily for random access iterators:

```
&*(p + 1) == &*p + 1;
```

1.3.6 Member Selection Operator

The standard library requires that all iterator types whose value types are aggregates shall also support the pointer member selection operator, as in the following:

```
struct X
{
  int x;
};

some_iterator<X>  si  = . . .
some_iterator<X>  si2 = . . .
some_iterator<X>  end = . . .

if( end != si &&
    end != si2)
{
  si->x = si2->x;
}
```

The standard (C++-03: 24.1.1;1) requires that an expression in which the pointer member selection operator is applied to an iterator is semantically identical to that in which the dot member selection operator is applied to the result of applying the dereference operator on the iterator. In other words, `it->m` is identical to `(*it).m`.

Unfortunately, there are some situations with STL extensions (Section 3.5, Chapter 36) where this operator cannot be supported because the iterator instance does not have access to an instance of the value type whose address it could return from `operator ->()`. Furthermore, you can get into trouble when using this operator on iterators to ranges of so-called smart pointers.

Consider that we have a container type (`C`) whose instances hold instances of a smart pointer type (`P`) that manages object lifetimes, and that this smart pointer can effect early release of its managed object by a `release()` method. Now suppose that the managed type (`T`) also has a `release()` method, and we wish to call that on an instance of the container's iterator type (`I`). We might write the following:

```
C   cont = . . .
I   it   = cont.begin();
it->release();
```

Unfortunately, this does not call `T::release()`, it calls `P::release()`, thereby (in all likelihood) destroying the `T` instance. When we come to use `cont` again, we will have an unpleasant surprise. We really would have wanted to write the following:

```
C   cont = . . .
I   it   = cont.begin();
it->->release();
```

The C++ language does not support this kind of construction, for good reason. (Imagine how obfuscated C++ competitions would be swamped by entries vying for the maximum number of concatenated pointer member selection operator invocations in a single expression.)

To make C++ do what we want, we're forced to eschew the member selection operator and instead use the (parenthesized) dereference operator, as follows:

```
C   cont = . . .
I   it   = cont.begin();
(*it)->release();
```

This is an annoying inconsistent blot on the syntax of iterators. When we add in the restrictions surrounding iterators that have *by-value temporary* as their element reference category (described in Section 3.3.5), use of the pointer member selection operator is seen to be fraught with frustration and disappointment. Supporting it in iterators is nothing more than syntactic sugar with no intrinsic value whatsoever, and I think it was a grave mistake for the standard to prescribe that it feature in the iterator concepts. In my work, I avoid its use completely, and with only a couple of exceptions, you will always see the use of iterator dereference and the dot member selection operator in this book and in my code. I advise you to do the same.

Tip: Prefer iterator dereference and the dot member selection operator (`(*it).m`) over the pointer member selection operator (`it->m`).

1.3.7 Predefined Iterator Adaptors

As well as defining the iterator and its five refinements, the standard also provides several iterator adaptors.

`std::reverse_iterator` is a ***Class Adaptor*** that adapts a bidirectional or random access iterator type to define the equivalent reverse iterator type. For example, the reverse iterator types for the `std::vector` container class template are defined as shown in Listing 1.1.

Listing 1.1 Reverse Iterator Member Types for `std::vector`

```
// In namespace std
template< typename T // The container value type
        , typename A = std::allocator<T>
        >
class vector
{
public: // Member Types
  typedef  ????                          iterator;
  typedef  ????                          const_iterator;
  std::reverse_iterator<iterator>        reverse_iterator;
  std::reverse_iterator<const_iterator> const_reverse_iterator;
  . . .
```

In almost all cases of defining the reverse iterator types for containers and collections, `std::reverse_iterator` is the appropriate tool; Chapter 26 shows one of the rare exceptions to this rule.

The standard also defines several iterator ***Instance Adaptors***. `std::back_insert_iterator`, `std::front_insert_iterator`, and `std::insert_iterator` adapt container instances by presenting an output iterator by which values will be inserted via the methods `push_back()`, `push_front()`, and `insert()`, respectively. The iterator instances are returned by the creator functions `std::back_inserter()`, `std::front_inserter()`, and `std::inserter()`, respectively. Creator functions are used to dramatically simplify the use of templates. In particular, the automatic type-deduction mechanism, necessary for resolving function overloads, means that template parameters need not be explicitly specified by the user. In Part III, we'll look at just how much hardship this can avoid.

For instance, the following example copies the elements from a vector into a list, in reverse order.

```
std::vector<int>  vec;

. . . // Code that inserts elements into vec

std::list<int>    lst;
std::copy(vec.begin(), vec.end(), std::front_inserter(lst));
```

The other standard ***Instance Adaptors*** are the stream iterators, which provide adaptation of input and output streams and stream buffers in the form of input and output iterators: `std::istream_iterator`, `std::ostream_iterator`, `std::istreambuf_iterator`, and `std::ostreambuf_iterator`.

The following example shows how `std::istream_iterator` may be used to read in a set of values from the standard input stream, `std::cin`, and how `std::ostream_iterator` may be used to write that set, after sorting, to a string stream.

```
std::vector<int>    values;
std::stringstream   sstm;

std::copy(std::istream_iterator<int>(std::cin)    // Read in values
        , std::istream_iterator<int>()
        , std::back_inserter(values));
std::sort(values.begin(), values.end());          // Sort values
std::copy(values.begin(), values.end()
        , std::ostream_iterator<int>(sstm, " ")); // Write out
```

The whole of Part III of this book is dedicated to iterator adaptors.

1.4 Algorithms

STL algorithms are generic function templates that are applied to iterator ranges and operate on the ranges and/or the elements within them. For example, the `std::distance()` algorithm does not dereference the iterators passed to it but merely calculates the number of elements represented by the given range. Conversely, the `std::transform()` algorithm assigns to the elements within one range the transformed values of the elements within another range.

```
std::vector<int>  v1 = . . . ;
std::vector<int>  v2(v1.size());
std::transform(v1.begin(), v1.end(), v2.begin(), ::abs);
```

Although there are numerous uses of algorithms throughout this volume, the algorithm concept is discussed in detail in Volume 2, along with customizations and extensions.

1.5 Function Objects

A function object, or functor, is a callable object. That means it is either a genuine function or an instance of a class type defining the function call operator. Function objects are primarily used in combination with algorithms. They occur either as predicates or functions. Predicates take one or more arguments and return a Boolean value that may be used to control selection of range elements. Functions take zero or more arguments and return a result value that may be used in the transformation or assignment of elements. Function objects are discussed in detail in Volume 2, along with customizations and extensions.

1.6 Allocators

The allocator concept (C++-03: 20.1.5) describes the requirements for types that provide memory-handling services. These services include definition of certain member types, along with methods for the allocation and deallocation of raw memory and the (in-place) construction and destruction of objects of the allocator's parameterizing type. Fundamentally, allocators abstract the details of memory handling from containers (and other components that manipulate memory) into a single well-defined interface, allowing such components to be specialized with different memory services as required. For example, the following code shows two specializations of the `std::vector` class template, based on implicit selection of the default allocation scheme and explicit selection of a custom allocator (in this case, the **WinSTL** `processheap_allocator`, a ***Façade*** over the Windows **Heap** API).

```
std::vector<int>                                       vec1;
std::vector<int, winstl::processheap_allocator<int> >   vec2;
```

`vec1` will allocate its memory, via `std::allocator`, from the C++ free store. `vec2` will allocate its memory, via `winstl::processheap_allocator`, from the Windows process heap.

The details of the allocator concept will be covered in Volume 2, along with examples of and techniques for defining allocator ***Façades*** and allocator ***Adaptors*** and a surprising examination into the issue of allocators bearing state.

CHAPTER 2

Extended STL Concepts, or When STL Meets the Real World

Good law is clear and simple, and gives broad discretionary powers to [practitioners] to deal with particular cases.

—Professor Ron McCallum

Learn what you should be doing, and do it.

—Billy Connolly

The previous chapter covered the essentials of the STL, including the core concepts of containers, iterators, algorithms, function objects, allocators, and adaptors. Unfortunately, extending STL shows that some of these concepts are either too prescriptive or too coarse grained. In this chapter, we'll examine these concepts with respect to the material covered in this volume about collections and iterator adaptors

2.1 Terminology

Although the standard library incorporates much from the original STL, it does not (yet) incorporate all. For example, the C++-03 standard currently defines only tree-based associative containers and not the hash-table-based associative containers from the original STL. Proposals exist for such containers, however, and the next release of the C++ standard will contain them. But the fact remains that the standard library is not a superset of the STL.

Conversely, the standard library contains STL-compatible components that are not part of the STL, that is, the **IOStreams**. Whatever you might think of the **IOStreams** as a serious I/O library, the addition of STL-conformant interfaces to it is a net benefit for C++.

STL and the standard library have a lot of overlap, yet they are distinct. In this volume, I do not cover any STL components that are not also part of the standard library and vice versa. Therefore, when I refer to a standard component, I mean one that is both part of the STL and part of the standard library and whose definition is in accordance with the standard library.

But since this is a book primarily about extending the STL, we also need a nomenclature for discussing extensions. Such extensions do not merely add new containers. Indeed, most of the extensions that provide access to ranges of elements do not even qualify as containers. I refer to these as STL collections. The next section discusses the collection concept and how it relates to the container concept.

2.2 Collections

The STL concerns itself only with containers. The standard defines containers as "objects that store other objects [and that] control allocation and deallocation of these objects through constructors, destructors, insert and erase operations" (C++-03: 23.1;1). However, there is a *far* greater set of types pertaining to collections of objects than would fit that definition, as shown in Figure 2.1. Thus, I will use the taxonomy defined in this chapter throughout the rest of the book and in Part II in particular.

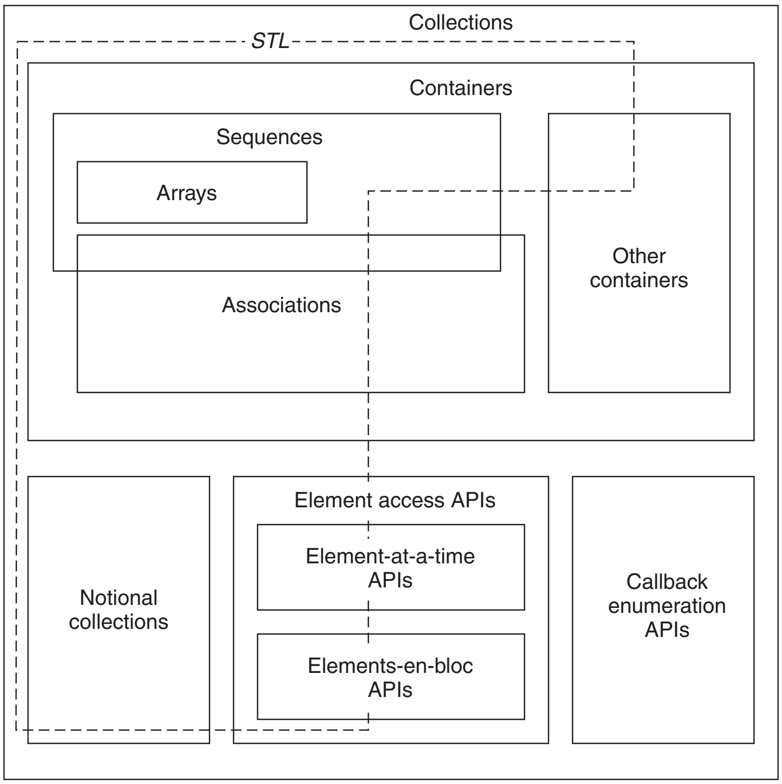
First, we define a collection.

Definition: A *collection* is a grouping of zero or more elements and an interface by which those elements may be accessed in mutating and/or nonmutating fashion.

Collections may be categorized into several distinct groups. *Notional collections*, such as mathematical series, have no physical existence. The Fibonacci sequence (discussed in Chapter 23) is one of these.

Element access APIs provide access to their elements via a programming interface whose types may not necessarily be the actual types, if any, that the underlying ranges contain. Furthermore, the manner in which the elements are returned to the caller can be significantly different

Figure 2.1 A taxonomy of collection concepts, showing overlap with STL



from the manner in which they are stored. For example, the UNIX **glob** API returns to its caller a callee-allocated block of strings containing the paths of all matching elements retrieved. This is an *elements-en-bloc API*. Conversely, the UNIX **opendir**/**readdir** API provides information about each successive file system entry retrieved in a given search in the form of null-terminated strings representing the entry's file name only. This is an *element-at-a-time API*. (The structures and functions of these APIs, and their adaptation in STL collection components, are discussed in Chapters 17 and 19, respectively.)

Taking our lead from the C++ standard gives us a definition for a container.

Definition: A *container* is a collection that owns its objects and provides operations by which those objects may be accessed and modified and, optionally, added, removed, and rearranged.

Prosaically, you may think of containers as what you already know them to be and collections as containers plus everything else.

Since we are interested in STL in this book, I define two additional concepts: STL collection and STL container.

Definition: An *STL collection* is a collection that provides mutating and/or nonmutating iterator ranges via `begin()` and `end()` methods.

Definition: An *STL container* is a container that provides mutating and/or nonmutating iterator ranges via `begin()` and `end()` methods.

Note that an STL container does not necessarily fulfill all of the requirements of the standard's container concept; an entity that does meet them may be referred to as a standard-conformant container.

Definition: A *standard-conformant container* is an STL container that meets the requirements of the standard library's container concept.

Definition: A *standard container* is a standard-conformant container currently defined by the standard library.

It is usually possible to fulfill the requirements of the container concept by the addition of the methods `size()` (and `max_size()`), `empty()`, `swap()`, and several member typedefs. However, this is not always the case.

Finally, we define two more terms, one for containers that, like `std::vector`, store their elements in a contiguous block and one for collections that provide access to elements that are stored thusly, and whose iterators are contiguous (Section 2.3.6).

Definition: A *contiguous container* is an STL container that stores its elements in a contiguous block of memory and exhibits contiguous iterators.

Definition: A *contiguous collection* is an STL collection whose elements are stored in a contiguous block of memory and that exhibits contiguous iterators.

The dotted STL box in Figure 2.1 denotes the extent to which we can apply STL concepts and provide STL compatibility. By definition, the standard containers are STL containers. The overlap with other containers, notional collections, and (both types of) element access APIs identifies the main business of the material in Part II, whereby collections of all these kinds are wrapped in STL-compatible forms.

Note that *callback enumeration APIs* are not compatible with STL and cannot be made so. The reasons for this will be discussed in Volume 2, when we will look at the *range* concept. The range concept is concerned with the manipulation of groups of elements en bloc and is thereby able to encapsulate *all* collection types. It is thus complementary to the STL *iterator* concept, which facilitates the manipulation of individual elements.

2.2.1 Mutability

Most containers allow changes to the objects they contain and to the arrangement of those elements within the container. For example, `std::list` allows the elements to be modified via the references returned by `front()`, `back()`, and dereference of a mutating (non-`const`) iterator, and it allows the arrangement of elements to be modified by operations such as `insert()`, `erase()`, and `sort()`. Such containers are, obviously, mutable. What may not be so obvious, however, is that they support two forms of mutability, which are severally important to the understanding and classification of collections.

Definition: An *element-mutable collection* permits modification to its elements. An *element-immutable collection* does not.

Definition: An *arrangement-mutable collection* permits modification to the arrangement of its elements. An *arrangement-immutable collection* does not.

All standard-conformant containers are arrangement-mutable and element-mutable.

A few containers, such as **STLSoft**'s multidimensional array family (described in Chapter 33 of *Imperfect C++*), allow modification of elements but have a fixed structure, such that all objects are contained in the container's constructor and persist until they are destroyed in the container's destructor. These containers are element-immutable. Although virtually unheard of in practice, a container that is element-mutable but arrangement-immutable is possible; however, the standard library does not cater to this possibility. (Such containers are seen in other languages, where immutable element types [usually strings] are common, including Python, Java, and .NET.)

If a collection is both element-immutable and arrangement-immutable, it is immutable.

Definition: An *immutable collection* does not permit modification to its elements or to the arrangement of its elements.

Immutable containers are something of a contradiction in terms since the whole purpose of a container is to be able to hold groups of things whose composition are not known until runtime. However, in the broader world of collections, many, if not most, collections are either element-mutable-only or arrangement-mutable-only; indeed, some are immutable. This may be because the elements they are providing access to are taken en bloc from the underlying API or because there is no means by which the elements may be modified. We'll see examples of collections with differing forms of mutability throughout Part II and in Volume 2.

2.3 Iterators

There are other ways in which STL extensions do not conform to established STL concepts. In the previous chapter, we discussed the characteristics of the different iterator refinements. The problem with the iterator (Section 1.3) concept and the refinements defined in the standard is that a number of characteristics are conflated in the five existing iterator categories.

When defining/using iterators in respect of container types, the issues are clear and obvious. Indeed, some of them would be deemed moot. But when writing STL extensions and having to define iterator types for STL collections, it can be far less clear, as we'll see throughout this book.

2.3.1 Mutability

In the world supported by the standard, *input iterators* support only nonmutating access to their elements, *output iterators* support only mutating access to their elements, and all other iterator categories support both nonmutating and mutating access. Of course, in the real world, we will want to be able to define STL collections that may provide different permutations of these characteristics, for example, only nonmutating operations while supporting bidirectional traversal.

2.3.2 Traversal

An instance of a *forward iterator* can be copied to yield an instance that will enumerate an identical range of elements. The *bidirectional iterator* refinement of forward iterator defines operations for reverse traversal of a range. If an iterator is reversible but cannot guarantee to yield identical ranges between copies, there is no suitable category within the standard to represent its behavioral characteristics. We'll see an example of this in Chapter 26.

2.3.3 Compile-Time Characteristic Determination

For all cases covered by the STL, the supported iterator refinement is known at compile time. This is also true for the majority of STL extensions. However, there are a few cases where the required behavior of an underlying collection may vary at runtime. There is no mechanism within the STL to represent this. We'll see an example of this and how it's handled in Chapter 28.

2.3.4 Element Reference Category

Variation in the nature of iterators' element references—what I call the *element reference category* (Chapter 3)—is not addressed at all: Iterators are assumed to be able to provide a genuine C++ reference to the current element at all times. As we'll see, there are occasions when this is not desirable for logical and/or efficiency reasons and rare cases when it is not possible (in typetransformative iterator adaptors). This issue is discussed in detail in Chapter 3 and impacts on the majority of collections (in Part II) and iterator adaptors (in Part III).

2.3.5 State Sharing versus Independence

A type that models the input iterator (Section 1.3.1) refinement does so because it cannot fulfill the requirements of the forward iterator (Section 1.3.3) refinement. Usually, this is because the underlying range is single-pass. The only examples of this provided in the standard library are `istream_iterator` and `istreambuf_iterator`, both of which implement the single-pass nature by invoking the requisite retrieval operation of their underlying stream or buffer. The stream or buffer is shared by any and all copies of a given instance, and therefore their shared state and single-pass behavior is implicit.

However, implementing STL extensions on element-at-a-time APIs is not always so simple. There may be state associated with the retrieval, and that state must be shared among any and all copies of a given instance, to properly and safely implement the full semantic requirements of input iterator. As we'll see, in Chapters 19 and 20 and Section 31.4, it is ironically the case that implementing the lesser iterator refinements usually requires the most effort.

2.3.6 Revising the Iterator Refinement Taxonomy?

In an ideal world, we might wish for a much broader definition of iterator categorization. Indeed, in the next chapter I will talk about a new separate characteristic of iterators, the element reference category. Given the variation around the basic theme that is to be found in the wider world of STL extension, we might imagine three or more orthogonal characteristics. Arguments could certainly be put forth for this. Notwithstanding any technical merits to such delineation—we'll see later how element reference category may be inferred using existing concept characteristics (Chapter 41)—it would certainly enhance understanding. For example, rather than referring to a *constant forward iterator*, we could refer to a *nonmutating cloneable unidirectional iterator*. However, it's likely that the geometric expansion of such characteristics would be counterproductive, even were a consensus achievable between practitioners. But this is all moot. The standard is what it is, and we cannot hope to so radically redefine a concept as fundamental as iterator.

Nonetheless, for clarity I will use two new terms in this book. When discussing input iterators I will refer to the nonmutating single-pass semantics, to stress that such an iterator type is restricted to a single active instance over a given range at any one time. The other new term refers to a refinement of the *random access iterator* concept, giving the *contiguous iterator* concept. It has all the characteristics of random access iterator, with the additional requirement that the following relation holds for any of its instances:

```
&*(it + 1) == &*it + 1;
```

This is the same relation I mentioned in Section 1.3.5 that distinguished random access iterators from pointers. However, it's still important to note that not all contiguous iterators are pointers, since a class type that overloads the address-of operator (`operator &()`) can also support this relation. (I argue strongly against overloading this operator in Chapter 26 of *Imperfect C++*. Notwithstanding the worthy objections to this practice, it is still done on occasion. In the extra chapter, Beware Nonpointer Contiguous Iterators, included on the CD for this book, we'll look at an example of its use in one compiler's standard library, the problems that this causes for STL extenders, and a solution.)

You might wonder why this additional refinement is significant. In the standard library, the only container that could be said to provide contiguous iterators is `std::vector`. (Note that this does not include what one reviewer refers to as the "red-headed stepchild of standard library containers," `std::valarray`. I am not a numerical programmer, so maybe I miss some essential nuance, but to me `valarray` seems an abomination so bad that it is second only to `std::vector<bool>` for its profound seemed-like-a-good-idea-at-the-time-ness. In case you didn't know, `operator ==()` on a `valarray` is not a nonmutating comparison operator, returning a Boolean reflecting the equality of its two comparands, but is a function that returns an instance of `valarray<bool>` whose elements indicate the equality of the elements of the two comparands! Whatever the rights and wrongs, `valarray` is not discussed in this book.)

Outside the standard library, in the world of STL extension, there can be many more contiguous iterators associated with extension collections. Talking in terms of this extra refinement helps connote this characteristic of such collections and also helps emphasize the differences between random access iterators and pointers (and their equivalent class types that overload `operator &()`). Furthermore, a range represented by contiguous iterators will be compatible with a function that takes pointer parameters. Consider the following two ranges of random access iterator (`RI`) and contiguous iterator (`CI`) types, each of which has the value type `V`:

```
RI  r_begin = . . .
RI  r_end   = . . .
CI  c_begin = . . .
CI  c_end   = . . .
```

Given the function `use_V()`:

```
void use_V(V* pv, size_t n);
```

and assuming that the ranges [`r_begin`, `r_end`) and [`c_begin`, `c_end`) are not empty, the following expression is well defined:

```
use_V(&*c_begin, c_end – c_begin);
```

but this one is not:

```
use_V(&*r_begin, r_end – r_begin);
```

We'll see several examples of the importance of the difference between these two refinements in Parts II and III.

CHAPTER 3

Element Reference Categories

In any revolutionary area, most of our current thinking is wrong.
—David Suzuki

If we had some ham, we could have some ham and eggs, if we had some eggs.
—L. Hunter Lovin

3.1 Introduction

Q: *When is an element reference not a reference?*
A: *When it's elicited from a collection or an iterator that does not own its element(s).*

That seems like a relatively uncontentious statement. As we will see, however, it has important and far-reaching implications for STL extensions. In this chapter, I will introduce the concept of an *element reference category*, which is a peer to the familiar concept of *iterator category* and just as important to STL extensions.

The reason you may not have heard of element reference categorization in your travels thus far with STL is quite simple: Almost none of the components in STL proper exhibit the more exotic and restrictive element reference categories. So permit me to take you on a necessary, though occasionally arcane, journey into the world of element references.

3.2 C++ References

A C++ reference is a name for an existing object. Two aspects of this are important when we are concerned with extending STL. First, a reference is just a name for another instance, known as the referent: It's an alias of the referent. Whatever you do to the reference, you are actually doing to the referent, as shown here:

```
int   referent  = 0;
int&  reference = referent;
++reference;
assert(1 == referent);
```

Second, a reference is a name for an *existing* referent. If the program renders the referent invalid, further use of the reference is undefined. One of the classic pitfalls of C++ is returning a reference to a local variable, as in the following:

```
std::string const& fn(char const* s)
{
  std::string str(s);
  return str;
}

std::string const& rstr = fn("Something nasty is about to transpire");

std::cout << rstr << std::endl; // Boom!
```

C++'s scoping rules require that the instance `str` is destroyed before `fn()` returns, so its use in the output statement yields undefined behavior (probably a crash).

3.2.1 STL Container Element References

The standard requires that containers store their contained elements by value. However, for reasons of usability and performance, containers expose their contained elements by reference. For example, in order to modify the first element in a `std::vector<std::string>` container, we need only obtain a reference to it.

```
std::vector<std::string>  cont = . . . // Assume nonempty
std::string&              str  = cont.front();

str.append(" and a little more"); // Changes element inside cont
```

If STL containers could not provide mutating references to their elements, their contents would have to be changed by replacement, which would be inefficient at best and would have unacceptable semantics in certain cases. For example, to append to the zeroth element, we would have to append to a copy and then replace the original with the result.

```
cont.replace(0, cont[0] + " and a little more");
```

Similarly, if we could obtain the contained elements only by value rather than by nonmutating (`const`) reference, we would incur performance losses for many contained element types.

```
std::string str = cont[0]; // Instance copied: Wasteful ...
```

It's much better to be able to take a reference. Further, a nonmutating (`const`) reference to an element of an STL container remains valid as long as no mutating operation is carried out on the container. (It's also the case that some mutating operations of some containers, such as `std::list`'s `push_back()`, may still leave references valid.) Given that, you can make strong assumptions, as in the following:

```
void fn(std::vector<int> const& cont)
{
  if(!cont.empty())
```

```
   {
     int const& ri = *cont.begin(); // ri is valid from here ...
     . . .
   } // ... to here, no matter what happens in between
 }
```

Nonetheless, this expeditious mix of references with container value semantics means that the issue of referent validity is not completely straightforward with STL and much less so with STL extensions. Permit me now to sketch my taxonomy.

3.3 A Taxonomy of Element Reference Categories

There are five categories (and one variant) of element references, shown in Figure 3.1 in descending order of importance. The following subsections discuss each category.

3.3.1 Permanent

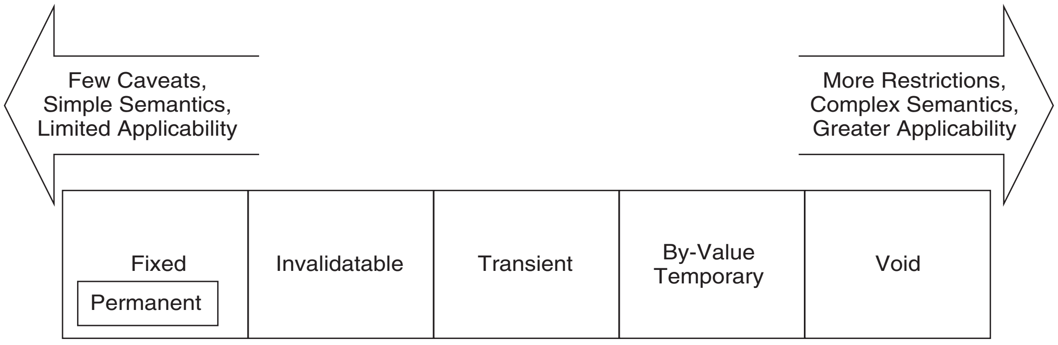
Definition: A *permanent* element reference references an element of a nonlocal arrangement-immutable collection (Section 2.2).

Consider the following permanent element reference:

```
namespace something // Also applies in global namespace
{
  int         ai[10];
  int const&  ri = ai[4];
} // namespace something
```

Aside from pathological global-object-ordering dependencies (described in Chapter 11 of *Imperfect C++*), such references are guaranteed to be valid for their entire lifetime. Permanent references are a special case of fixed references, described next, and are very rare in practice. They are not discussed further in this book.

Figure 3.1 A taxonomy of element reference categories



3.3.2 Fixed

Definition: A *fixed* element reference references an element of an arrangement-immutable collection.

Consider the following code:

```
stlsoft::fixed_array_2d<int>  ar2d(10, 20);
int&                          i_2_3 = ar2d[2][3];
```

STLSoft provides a suite of class templates collectively known as the fixed array classes. They are arrangement-immutable because all elements are created, and their dimensions fixed, at construction time. No operations cause elements to be destroyed: The reference to the element at index `[2][3]` is valid throughout the lifetime of `ar2d` and is only destroyed, along with the other 199 elements, in the array's destructor.

Unlike permanent references, it is not *that* difficult to get hold of an invalid reference, although it requires deliberate action (or very sloppy programming).

```
int& bad_wolf()
{
  stlsoft::fixed_array_2d<int>  ar2d(10, 20);
  return ar2d[2][3];
}

int& i_2_3 = bad_wolf(); // Use of i_2_3 is undefined. Boom!
```

Though rare to see, it is possible for an *arrangement-mutable* collection (Section 2.2) to support the fixed reference category, if used in such a way that its mutating operations never result in invalidation of extant references. An example of this—a class wrapping a `std::list<>` and exposing only additive operations—is shown in Section 25.6.1.

3.3.3 Invalidatable

Definition: An *invalidatable* element reference references an element of an arrangement-mutable collection.

An invalidatable reference is valid only as long as no mutating operations on the collection destroy the referent element. In other words, after a call to `insert()` and/or `erase()` (or their associated `push_front()`, `push_back()`, `clear()`, or other methods), the referent will be invalid. Invalidatable element references are the ones you're most likely familiar with from standard containers.

The precise relationship between specific mutating methods and reference invalidation is collection-dependent. In fact, it's a fundamental property of the structure of a particular collection type. For example, adding an element to a `std::vector` with no spare spaces (i.e., `capacity() == size()`) will cause a reallocation of the memory block within which the elements are stored, thereby invalidating any iterators and/or references to the elements elicited before the addition.

```
std::vector<int>  vi(10);
int&              ri = vi[0];

vi.resize(vi.capacity() + 1);

int i = ri; // Undefined!
```

Conversely, adding an element to a `std::list` never invalidates extant iterators/references. Further, removing one or more elements from a `std::list` invalidates only extant iterators/references to elements in the range of elements removed. Consider the following code:

```
std::list<int>  vi;

vi.push_back(10);
vi.push_back(20);

std::list<int>::iterator  b0  = vi.begin();
std::list<int>::iterator  b1  = b0;
int&                      ri0 = *b0;
int&                      ri1 = *++b1;

vi.erase(b0);

std::cout << "ri0=" << ri0; // Here there be dragons!
std::cout << "; ri1=" << ri1 << std::endl;
```

Use of the reference `ri1` in the last statement is entirely proper. Only the use of `ri0` is invalid. On my test machine, this statement prints *`"ri0=-17891602; ri1=20"`*.

By contrast, removing an element from a `std::vector` invalidates all the elements from the removal point to the end of the element sequence. However, this is less obvious than the case with `std::list` and often may not manifest any apparent problem. Consider the following code:

```
std::vector<int>  vi;

vi.push_back(10);
vi.push_back(20);

std::vector<int>::iterator  b0  = vi.begin();
std::vector<int>::iterator  b1  = b0;
int&                        ri0 = *b0;
int&                        ri1 = *++b1;
```

```
vi.erase(b0);

std::cout << "ri0=" << ri0; // Here be more cunning dragons!
std::cout << "; ri1=" << ri1 << std::endl;
```

In this case, the output on my machine is `"ri0=20; ri1=20"`. The slightly surprising thing about this is that the referent of `ri0` appears not to be invalidated, *in this case*, because there is more than one element. Those elements after the erasure point are logically moved down, but in fact this is effected by copy assignment—one of the reasons why `std::vector` requires its elements to be *Assignable* (C++-03: 23.1). After copy assignment, the copy is equivalent to the original. Hence, the referent of `ri0`, which formerly held the value *10*, has the value *20* copied into it.

Naturally, the fact that the space formerly occupied by the second element still contains the value *20* is entirely dependent on undefined features of the given implementation; it could equally have been overwritten by a random value.

As we'll see later in the book, some STL adaptations of contiguous-allocation containers (such as the `CArray` adaptors in Chapter 24) have different movement semantics, in which case the `ri0` reference would indeed be invalidated (even though, perplexingly, it would still work as expected). Regardless of the way collection mutation is implemented, invalidatable element references are invalidated only in response to arrangement-mutating operations on their host collections.

3.3.4 Transient

Definition: *Transient* element references can be invalidated by nonmutating use of their host objects.

Transient element references are commonly seen with iterators of class type that contain an element of the value type of their collection, in the case where the collection itself does not store its elements. An example of this is `std::istream_iterator`. Upon construction, and with each invocation of its increment operator(s), the latest byte(s) for its value type are read from the underlying stream and maintained in an internal variable ready for access by client code via `operator *()`.

The standard does not prescribe that the same instance is used to store each value read from the stream; it merely stipulates that the return type of `operator *()` is `T const&`. Herein lies the transience of such references, both logical and physical. Logically, and irrespective of the state of the elements of the underlying collection, incrementing (or decrementing) an iterator invalidates the reference(s) derived from it.

Furthermore, you *must* assume that such references are also physically transient: In other words, the memory containing the referent is destroyed when the iterator is incremented. In reality, all standard library implementations with which I've executed the following code indicate that the iterator instance uses a single internal member variable:

```
std::istream_iterator<char> b(std::cin);

for(; b != std::istream_iterator<char>(); ++b)
{
  ::printf("*(%p) == %c\n", &*b, *b);
}
```

With GCC 3.4 and an input of `"abcd"` this gives the following output:

```
*(0022FED4) == a
*(0022FED4) == b
*(0022FED4) == c
*(0022FED4) == d
```

Thus references derived from an iterator at different iteration points continue to refer to the same physical variable. When reading two or more characters, the following code assertion holds true:

```
std::istream_iterator<char> b(std::cin);

char const& r0 = *b++;
char const& r1 = *b++;

assert(&r0 == &r1); // Holds for common impls, but not required
```

It is a relatively simple matter to implement a standard-conforming iterator that does not re-use the same member variable. Naturally, I've implemented such a class for your delectation: `transient_istream_iterator`. (It's included on the CD.) It reallocates an instance of the value type from the heap on each read operation. Running with the same input noted earlier, the output is as follows:

```
*(003F4F80) == a
*(003F4EA0) == b
*(003F4F80) == c
*(003F4EA0) == d
```

For those of you who may have reservations about the seemingly pick-and-mix adherence to, or flouting of, STL rules when writing STL extensions, it's worth noting that `std::istream_iterator` does the same thing. The standard defines that "the result of `operator ->` on an end of stream is not defined" (C++-03: 24.5.1;1). We'll see more such exceptions to the STL rules throughout the book.

3.3.5 By-Value Temporary

Definition: A *by-value temporary* element reference is actually not a reference at all but is in fact a temporary instance returned from the callee.

Usually the callee is an iterator, either an instance of an iterator adaptor (see Part III) or the iterator of an STL extension collection (see Part II). For example, the `transform_iterator` (Chapter 36) iterator adaptor applies a unary function to the elements of its base iterator type to transform the value and/or type. As such, it's not in a position to give out a reference to anything, so it exhibits the by-value temporary element reference category: The value returned from its dereference operator is an instance of, rather than a reference to, the transformed type. Similarly, the iterator class of the `string_tokeniser` (Section 27.6) does not maintain a copy of the element at the current enumeration point but rather creates an instance of the string fragment corresponding to its enumeration point on demand.

Iterators that exhibit the by-value temporary element reference category must define their `pointer` member type as `void` and their `reference` member type as the same type as their value type. The former is so that by-value temporary can be detected at compile time (see Chapter 41): It's like a mini-concept decoration. The latter is so that adaptations, especially via `std::reverse_iterator`, have a viable non-`void` type with which to implement their dereference operator. For example, Listing 3.1 shows the specialization of the `std::iterator` type generator template (Section 12.2) for the `string_tokeniser::const_iterator` class (Section 27.6.1).

Listing 3.1 Declaration of By-Value Temporary Iterator Category

```
template <. . .>
class string_tokeniser
{
  . . .
  typedef ????                  value_type;
  . . .
  class const_iterator
    : public std::iterator< std::forward_iterator_tag
                          , value_type, ptrdiff_t
                          , void, value_type // BVT
                          >
  {
    . . .
```

In some cases, it may be that an STL extension collection itself has an element reference category of by-value temporary (which I'll indicate with `BVT` in the code listings). For example, the `environment_map` (Chapter 25) cannot maintain internal references to its elements. Its subscript operator must return instances of its string type.

Although it may sound like a corner case, the by-value temporary element reference category is actually one of the more common, if not the most common, of all element reference categories in the world (or, at least, *my* world) of STL extension. As we'll see throughout Volume 1, understanding this category, as well as expressing it and using it correctly in STL extension implementations, is vital.

3.3.6 Void

Definition: The *void* element reference category denotes that no value can be obtained from an iterator instance.

Void is the most restrictive and featureless element reference category. Iterators and containers with this category cannot create values. We see an example of this in the standard library with the `ostream_iterator`, which specializes the `std::iterator` type generator template with `void` for all members other than the iterator category.

```
template <. . .>
class ostream_iterator
  : public iterator<output_iterator_tag, void, void, void, void>
{
  . . .
```

All output iterators (see Chapters 34, 39, and 40) must do the same.

3.4 Using Element Reference Categories

"This is all very well," you may say, "but how does it affect my implementation and use of STL extension collections?" Well, if we can identify important differences in element reference category between types that we might wish to use generically, we can tailor our code to the capabilities of these types, for purposes of efficiency and/or avoidance of undefined behavior.

3.4.1 Detecting Category Flavors at Compile Time

Although it took me some time to realize it, there is scope within the existing framework provided by STL for the most important discrimination of element reference categories for iterators, a vital part of the techniques for iterator adaptation described in Part III.

By determining which, if any, of the `pointer`, `reference`, and `value_type` member types are defined as `void`, we can distinguish the element reference categories of by-value temporary and void, as shown in Table 3.1. (The definition of `difference_type` as `void` for void iterators is not necessary for identification of element reference category, but it is very useful in proscribing pointer arithmetic for output iterators.)

Table 3.1 Iterator Member Type Characteristics Indicating Element Reference Category

Reference Category	Defining Characteristic
Permanent, fixed, invalidatable, or transient	Member type `pointer` is not `void`. Member type `reference` is not `void`. Member type `value_type` is not `void`. Member type `difference_type` is not `void`.
By-value temporary	Member type `pointer` is `void`. Member type `reference` is not `void` but is not a C++ reference. Member type `value_type` is not `void`. Member type `difference_type` is not `void`.
Void	Member type `pointer` is `void`. Member type `reference` is `void`. Member type `value_type` is `void`. Member type `difference_type` is `void`.

There is no provision for separating out the other iterator element reference categories based on their member types, but that's actually of virtually no practical significance, simply because every trained STL user knows not to use a reference retrieved from a subsequently incremented (or decremented) iterator, without doing a double take on his or her assumptions. That's quite a different matter from thinking it's acceptable to use a reference retrieved from an iterator instance only used and not modified in any way.

3.4.2 Using the Compiler to Avoid Undefined Iterator Behavior

The important feature of temporary references is that they are, well, *really temporary*. Consider the following seemingly valid function template:

```
template <typename I>
typename std::iterator_traits<I>::value_type& get_iter_val(I it)
{
  return *it;
}
```

For all standard iterators, and for many STL extension iterators, the use of such a function is without issue (as long as the iterator is dereferenceable—not the endpoint iterator and not an output/void iterator). The function elicits from `it` a reference to its associated element and returns that reference to its caller. No copies of the element are made, no memory allocated, likely no significant action of any kind taken at the machine code level.

However, for those iterators whose element reference category is by-value temporary, such a function has two problems because the dereference of the iterator yields an actual instance rather than a reference. It is illegal in C++ to attempt to assign a temporary to a non-`const` reference (C++-03: 8.5.3). Furthermore, even if this were allowed, the reference that the client receives would be to an instance that was destroyed before `get_iter_val()` returned, according to C++'s scoping rules (Section 3.2).

The appropriate way to define this function is to use the `std::iterator_traits::reference` member type.

```
template <typename I>
typename std::iterator_traits<I>::reference get_iter_val(I it)
{
  return *it;
}
```

This form works as intended for all dereferenceable iterators. In the case of by-value temporary iterators, it is correct because the `reference` member of the iterator is the same as its `value_type`; thus, it returns by value. For all other iterator types, it is identical to the first definition of `get_iter_val()`.

3.5 Defining `operator ->()`

A clear consequence of the by-value temporary and void reference categories is that iterators exhibiting those references cannot implement the member selection operator, `operator ->()`. This is because they have nothing with which to provide a pointable/referenceable return value, as required by the language for any overload of this operator.

Rule: Iterators having element reference categories by-value temporary or void cannot define the member selection operator.

This rule compounds the problems described in Section 1.3.6. In truth, in most cases an iterator supporting by-value temporary may be made to support transient by storing an instance of the currently enumerated value, but this may have deleterious performance effects. In any case, there are situations in iterator adaptation where this is not so (Chapter 36). Thankfully, there are techniques for handling the definition/omission of the operator in respect to the element reference category, as we'll see in Chapter 41.

3.6 Element Reference Categories: Coda

If all this seems a bit obscure and/or pointless right now, worry not. The significance will become quite clear as we travel through Parts II and III, as the different characteristics impact on the design and implementation of collections, their iterators, and iterator adaptors.

CHAPTER 4

The Curious Untemporary Reference

Be yourself; everybody else is already taken.
—Sarah Leunde

A little-known feature of C++ impacts the semantics of code that makes use of *by-value temporary* element references (Section 3.3). Consider the following code:

```
std::string const& rs = std::string("Is it safe? Zzzz");

std::cout << rs << std::endl;
```

If your first instinct is to declare the code ill defined and the use of the reference `rs` in the output statement likely to crash the process, consider yourself reasonable but wrong. This is not so. C++ requires that references to instances of class type be valid until the end of their scope of declaration (C++-03: 8.5.3;5, 12.2;3), which may be achieved by creating a hidden temporary referent (Section 3.2). I call this the *curious untemporary reference*.

The implication of this is that by-value temporary element references (Section 3.3.5) may, in some circumstances, behave as if they're of a more persistent category. For example, the following code, using the `platformstl::environment_map` component (Chapter 25), is well defined:

```
environment_map::value_type const& v = environment_map()["PATH"];

std::cout << v << std::endl;
```

But there are two reasons you shouldn't get comfortable with this. First, the temporary instance is required to last only for the duration of its scope. If you pass it off to another reference, perhaps by attempting to return the reference from a function (as we saw in Section 3.2), all bets are off.

Second, this technique will not work for collections whose value types are not of class type or do not have value semantics. The same construction is undefined for both `unixstl::glob_sequence` (Section 17.3):

```
char const*& v = *glob_sequence(".", "*stl*.h*").begin();

std::cout << v << std::endl; // Not going to be pretty!
```

and `comstl::enumerator_sequence` (Section 28.5):

```
typedef enumerator_sequence<IEnumVARIANT, VARIANT, . . .>   enseq_t;
IEnumVARIANT*   p  = . . .
VARIANT const*& v = *enseq_t(p, false).begin();
::VariantCopy(. . . , v); // Boom for sure!
```

As we'll see, even when used safely, the curious untemporary reference can be used for good (Section 9.2.2) or ill (Section 25.9.5). Either way, it's something you need to be aware of.

CHAPTER 5

The *DRY SPOT* Principle

I love being a writer. What I can't stand is the paperwork.
—Peter de Vries

The *DRY SPOT* principle is what I call the business of avoiding multiple definitions of things. DRY stands for Don't Repeat Yourself, as eruditely espoused by those masters of the game, The Pragmatic Programmers, in their eponymous book. SPOT stands for Single Point of Truth, the equivalent principle well known under that name in the UNIX world. In respect to both these oracular sources, I refer to this principle as *DRY SPOT*, in no small part because, being an Englishman, I can no more resist a bad pun than one dog can resist sniffing another.

The *DRY SPOT*, by whatever name, is something that's basic fare for all good software engineers. It dictates that you store a piece of information in only one place. For example, rather than using the number 32 (which may not be correct on all architectures) for the number of bits in an `int` dotted around your code, you should define a constant such as the following:

```
const BITS_IN_INT = 8 * sizeof(int);
```

5.1 *DRY SPOTs* in C++

In C++ we dilute the principle a little by recognizing that *0* is a valid exception, in part because it can be converted to any numeric or pointer type as a special case. (Notwithstanding this, I always refer to a null pointer as the literal *NULL*, rather than the literal *0*.)

5.1.1 Constants

C++ programmers sometimes like to grant special (literal) status to other constants: *1* and *-1* are two obvious cases. For example, we might want to know when a reference count is *1* (Section 25.9). Making a special constant for that, for example, *REF_COUNT_1*, would be just silly. By the same token, defining a constant for the invalid file handle value, *-1*, returned by the UNIX `open()` and `socket()` system calls also impacts negatively on transparency. We also want to avoid pedanticism when doing multiplication. If you're writing code to double the size of a graphic, using a constant—`NUMBER_2`—for the multiplicand is plain silly.

Beyond such commonsense examples, though, the use of so-called magic numbers is a creeping ill, and you must be always on your guard. Write a script to grep through your source code base to find any that are not *1*, *0*, or *-1*. You might have an unpleasant surprise.

5.1.2 `dimensionof()`

In Chapter 14 of *Imperfect C++*, I bemoaned the lack of a `dimensionof()` operator for providing, at compile time, the (outermost) dimension of arrays and set about providing an implementation of it. As I discussed there, the use of `sizeof(x) / sizeof(x[0])`, well known in C, has the problem that `x` may be a pointer or a user-defined type that defines the subscript operator. In both cases, the result of the division is very likely to be quite wrong.

The solution I demonstrated there—which is used in the **STLSoft** libraries—employs a template-based technique, shown in Listing 5.1. The compiler translates a macro, `STLSOFT_NUM_ELEMENTS()`, taking an array name into a `sizeof()` operation on a template function that returns an instance of a struct template, whose size is equivalent to the dimension of the array. (For older compilers that get confused by the function template, it just defaults to `(sizeof(ar) / sizeof(0[ar]))`.)

Listing 5.1 Definition of STLSoft's `STLSOFT_NUM_ELEMENTS()`

```
// In namespace stlsoft
template <int N>
struct ss_array_size_struct
{
  unsigned char c[N];
};
template <class T, int N>
ss_array_size_struct<N> const& ss_static_array_size(T (&)[N]);

#define STLSOFT_NUM_ELEMENTS(ar) \
            sizeof(stlsoft::ss_static_array_size(ar).c)
```

I'm not going to go into the full rationale here—my editor has enough trouble with the volume of my output without my including previously published work! If you want to get hold of *Imperfect C++* and read up on this solution further, be my guest. Or just take my word that whenever you see `STLSOFT_NUM_ELEMENTS(ar)` in this book, you're seeing compile-time evaluation of the (outermost) dimension of the array `ar`, and the compiler will reject any attempt to apply it to pointers or instances of user-defined types that define the subscript operator.

Rule: Always use a `dimensionof()` analog that can discriminate and reject (at compile time) pointers and user-defined types when determining the dimension of arrays.

5.1.3 Creator Functions

Some templates are sufficiently complex that it can be a real bear to remember the specifics of their specializations. Thankfully, C++ supports implicit instantiation of function templates, so you don't have to stipulate all the types involved in your expressions when using them. This has led to the use of creator functions, which we'll see often in the chapters on iterators in Part III. For example, rather than writing the following code (from Chapter 36):

```
avg_mean<int>(stlsoft::transform_iterator<I
                            , std::pointer_to_unary_function<int, int>
                            >(from, std::ptr_fun(::abs))
            , stlsoft::transform_iterator<I
                            , std::pointer_to_unary_function<int, int>
                            >(to, std::ptr_fun(::abs)));
```

we can instead write:

```
avg_mean<int>( stlsoft::transformer(from, std::ptr_fun(::abs))
             , stlsoft::transformer(to, std::ptr_fun(::abs)));
```

which is something of a relief, to say the least. Stipulation of the same transform function object to both invocations of `transformer()` appears to be a *DRY SPOT* violation, but it's not (except when it occasionally, subtly, is). This will all be discussed in detail in Chapter 36.

5.2 Not Quite *DRY SPOTs* in C++

Sometimes, alas, the language's otherwise creditable support for *DRY SPOT* lets us down.

5.2.1 Parent Classes

When defining a class in a hierarchy, it is good form, and my consistent practice, to define a `parent_class_type` member type, with which you may refer to the parent class. This is especially useful when implementing hierarchies that might be subject to change, with parent classes either being renamed or exchanged. If everything in the child class is specified in terms of `parent_class_type`, rather than the name of the (current) parent class, changes are made much more seamlessly, in terms of the ease and correctness of the change itself and in terms of apparent differences when viewed via a version control system.

Consider the **UNIXSTL** exception class `missing_entry_point_exception` (shown in Listing 5.2), which is derived from `unix_exception`. This is used in `dl_call()` (Section 16.6).

Listing 5.2 Exception Class Using `parent_class_type` Member Type

```
// In namespace rangelib
class missing_entry_point_exception
  : public unix_exception
{
public: // Member Types
  typedef unix_exception                  parent_class_type;
  typedef missing_entry_point_exception   class_type;
public: // Accessors
  explicit char const* what() const throw()
  {
    if(m_name.empty())
    {
      return parent_class_type::what();
    }
    else
      . . .
```

To change missing_entry_point_exception to derive from a different class, all that's required is to change the parent class and the definition of parent_class_type. Obviously, this is a two-step process, hence it is *not quite DRY SPOT*. However, it is as close as it's possible to get without being so.

Tip: Always define a parent_class_type for each derived class in a (single-inheritance) hierarchy.

This technique is especially important when the hierarchy may be subject to changes that insert a class between the previous parent and child. Consider the case where the exception hierarchy needs to be altered such that the notional class dl_exception is inserted between missing_entry_point_exception and (its now grandparent) unix_exception. If missing_entry_point_exception had not used parent_class_type and instead just referred to unix_exception throughout, it would be all too easy to leave the implementation of its what() method as it stands, with negative consequences when an instance is caught and interrogated.

5.2.2 Function Return Types

Another instance of not-quite-ism occurs with function template return values. Consider the fragment of the **c_str_ptr** *string access shim* (Section 9.3.1) function overload for the Windows SYSTEMTIME type, shown in Listing 5.3.

Listing 5.3 Definition of c_str_ptr Overload for Windows SYSTEMTIME Type
```
basic_shim_string<TCHAR> c_str_ptr(SYSTEMTIME const& t)
{
  typedef basic_shim_string<TCHAR>  string_t;
  . . . // Various calls to calculate size, into 'numChars' variable
  string_t    s(numChars);
  . . . // Various calls to fill buffer with string-ified version of t
  return s;
}
```

In this case, the specialization of basic_shim_string has had to be made twice, in the function return type and within the function body. (Obviously, once inside the function you should, as shown, create a typedef. Otherwise, you might find yourself writing it several more times.) You might think there are no dangers here because the compiler will warn us if there's any mismatch, but that is not always so. Note the use of the ambient character type TCHAR, which evaluates to either char or wchar_t (depending on the absence or presence of the UNICODE preprocessor symbol). It's possible to have a mismatch between TCHAR and char, or TCHAR and wchar_t, that works in one compilation mode, only to fail in another. Such things can crop up long after you've worked on the code and can cause many a skipped heartbeat of consternation.

One of my reviewers commented that he was disappointed that this item doesn't come with a definitive solution. Unfortunately, there isn't one—you always have to specify *something* twice. However, when the *something* is sufficiently complex, you can significantly ameliorate the copy/paste angst by using a generator template (Section 12.2), as illustrated in the definition of the `member_selector()` iterator creator functions discussed in Section 38.4.5.

Tip: Consider using generator templates to reduce copy/paste risks when defining a function whose nontrivial return type must also be explicitly specified within the function body.

5.3 Closed Namespaces

Another aspect to the *DRY SPOT* principle is easy to miss among the myriad nuances of C++ lore. As I hope you know, a namespace provided by the C++ `namespace` keyword is *open*: You may extend its contents at any time by reopening the namespace via another namespace declaration under the same name. This can be subverted in ways unforeseen by the original author of the code. For example, you might write the following code, with the intent that your namespace will remain sacrosanct.

```
// OstensiblyDefinitiveVersion.hpp
namespace covenant
{
  int func(std::string const& s); // Specific version for std::string
  template <typename S>
  int func(S const& s);           // General version for other types
} // namespace covenant
```

However, there's nothing preventing any user from making potentially breaking changes to the namespace.

```
// CavalierManipulations.hpp
namespace covenant
{
  int func(std::exception const& x);
} // namespace covenant
```

Now the scope of the original function template is reduced, and the behavior of code previously dependent on only the original namespace may have been modified. This is generally not good, which is why the standard proscribes any additions to the `std` namespace that are not full specializations of existing templates. (See Section 31.5.3 for an interesting twist on this subject.) Of course, this kind of thing is not bad in all cases: The powerful and infinitely extensible *shim* concept (Chapter 9) relies on this very facility.

To get a truly *closed* namespace, you need to use a `class/union/struct`, as in:

```
// EnforceablyDefinitiveVersion.hpp
struct covenant
{
private:
  covenant(); // Prevent construction: it is a "namespace" after all
public:
  static int func(std::string const& s);
  template <typename S>
  static int func(S const& s);
}; // "namespace" covenant
```

Such a “namespace” is subject to restrictions that a real namespace is not. For one thing, it cannot have member constants that are not of integral type. Nor can it declare member namespaces, though this can be faked with a nested `class`/`struct`/`union`. It has other advantages over namespaces apart from being closed. For example, it can declare private members and their friends, thereby restricting its members to a set of known contexts.

Tip: Synthesize closed namespaces with structs (with private constructors).

CHAPTER 6

The Law of Leaky Abstractions

The stronger the faith, the closer the devil.
—Unattributed

In November 2002, in a particularly important and informative installment of his popular online column, “Joel on Software,” Joel Spolsky coined a new software law, which he calls the *Law of Leaky Abstractions*.

Law: All nontrivial abstractions, to some degree, are leaky.

Essentially, Joel’s thesis is that all (nontrivial) abstractions are imperfect, and there is always a point at which a user of an abstraction will need to know something about what’s underneath the abstraction in order to survive.

STL is a marvelous abstraction. Literally marvelous; I mean it. Take a moment to consider just how marvelous, and if you don’t agree now, be sure to take another moment once you’ve finished this book, and I am confident you will then agree.

But, and it’s a really *humongous* but, STL is a particularly leaky abstraction. To be sure, most abstractions in C++ are necessarily leaky, partly because of the freedom accorded to its practitioners (who are assumed to be both adept and diligent), and partly because it has all that nasty ol’ C compatibility to deal with. But STL is extreme, even for C++. Take the case of iterators. Iterators in C++ are based on the notion of a pointer. And the fun begins. . . .

A pointer `p` is the address of a place in memory. Memory is contiguous. The element `e` referenced by a pointer may be accessed—dereferenced—by using the dereference operator, `operator *()`, on the pointer, `*p`. If the element is of class type—a `struct`, `class`, or `union`—one of its constituent elements, `m`, may be accessed via the member selection operator, `operator ->()`, on the pointer, `p->m`. If the pointer points into an array of elements, pointer arithmetic may be used. `++p` and `p++` increment the pointer to refer to the element immediately after `e` in memory; the former evaluates to the value of the pointer after its increment, the latter to the value prior to its increment. `--p` and `p--` perform the same duties but move the pointer to refer to the element immediately preceding `e` in memory. `++p` yields the same result as `p + 1`, which is the same as `&p[1]`, which is the same as `&(*(p + 1))`, which is the same as `&(*(1 + p))`, which is the same as `&1[p]`. Pointer instances may be compared, for equality or inequality, with the special value *0* (sometimes codified as the macro *NULL*). Pointer instances may be arithmetically compared (by using `==`, `!=`, `<`, `<=`, `>`, and `>=`) with other pointer instances of the same type, as long as they're part of the same array, and the type of `*p` is not `void`.

Not one of the five categories of iterator provided by the standard guarantees to support all those pointer semantics. Even the extended category I introduce in this book, *contiguous iterator* (Section 2.3.6), does not require absolute semantic equivalence with pointers (although it gets close). All nonpointer STL iterators cannot, by definition, support the semantics of the abstraction they're modeled on. And we wonder why people get confused!

This book touches on many abstractions (as does Volume 2), and they all leak. That's part of life, part of software engineering, part of C++, and particularly part of STL. There's no escaping the abstractions, but we can prepare ourselves for them with knowledge, experience, idiom, good practice, and applicable concrete technology. This book aims to provide you with a good dose of all but the experience, for which you'll have to dip in a toe or ten.

CHAPTER 7

Contract Programming

Notions of responsibility are importantly preventive.
—Professor Martin Seligman

Emergency! Emergency! There's an emergency going on!
—Holly, *Red Dwarf*

Contract programming is the business of expressing the expected and allowed behaviors of software components as contracts and of including constructs to verify adherence to these contracts in your code. This chapter briefly covers the aspects of contract programming that pertain to the content of this book and Volume 2; I've included on the CD some articles I've written on the subject, within which my perspective on, and rationale for, the use of contract programming in C++ is clearly spelled out, in case you want to jump in with both feet.

It's important to realize that a software contract need never be enforced, or even expressed, in code. Sometimes a contract cannot be practicably expressed in code. Other times the author may deem it not worthwhile. My approach is always strongly focused on expressing and enforcing contracts in code, but that does not mean that I want to mislead you to believe that every contract condition is expressible in code or that absence of an enforcement implies that all behavior not proscribed is acceptable.

7.1 Enforcement Types

Enforcements are actions that the code takes to detect and deal with contract violations. Three kinds of enforcements are commonly used: preconditions, postconditions, and class invariants.

Preconditions state what conditions must be true in order for a function/method to perform according to its design. Satisfying preconditions is the responsibility of the caller. *Postconditions* say what conditions will exist after the function/method has performed according to its design. Satisfying postconditions is the responsibility of the callee. *Class invariants* state what conditions hold true for the class to be in a state in which it can perform according to its design; an invariant is a consistency condition that every class instance must satisfy whenever it's observable from the outside. Class invariants should be verified after construction, before destruction, and before and after the external call of every public member function; importantly, a class invariant does not have to hold *during* the call of a method. (Otherwise, how would we ever change the state of classes with more than one member variable?)

Precondition enforcement in C++ commonly takes the form of a test on the arguments to a method and/or the state of the method's object. These enforcements are easy to create, as shown in the following examples. Listing 7.1 shows the precondition test for one of the constructors of `stlsoft::scoped_handle` (Section 16.5), to ensure that it is not passed a *NULL* pointer for the cleanup function.

Listing 7.1 Example Precondition Test in the `scoped_handle` Constructor

```
// In namespace stlsoft
template<typename H>
class scoped_handle
{
  . . .
public: // Construction
  scoped_handle(H h, void (STLSOFT_CDECL* fn)(H), H hNull = 0)
      : . . . // Initialize members
  {
    STLSOFT_MESSAGE_ASSERT("Precondition violation: NULL fn ptr"
                          , NULL != fn);
  }
  . . .
```

It is a lot harder to provide meaningful implementations of postconditions in C++, and there will not be any presented in the material in this book (and probably not in Volume 2 either).

My mechanism for implementing class invariants involves each class defining a nonpublic nonvirtual method `is_valid()`, which takes no arguments and returns a Boolean value. This is then called at the start (after any precondition tests) and the end of each public method; the invariant method may also have assertions internally, without waiting to return to its caller, as appropriate to the implementation. Listing 7.2 shows how the invariant is checked for the `iterator` of the `enumerator_sequence` class template (Section 28.6) before and after the implementation (in the form of the private `increment_()` method) of the increment operator.

Listing 7.2 Preincrement Operator Using Invariant Enforcements

```
template <. . .>
class_type& enumerator_sequence::iterator::operator ++()
{
  COMSTL_ASSERT(is_valid());
  increment_();
  COMSTL_ASSERT(is_valid());
  return *this;
}
```

7.2 Enforcement Mechanisms

Enforcements have three aspects: detection, reporting, and response. *Detection* involves testing arguments and/or state to determine whether a contract violation has occurred. *Reporting* involves informing someone/something that the violation has occurred. *Response* involves the actions taken to terminate the process.

In C/C++, the best-known enforcement is the `assert()` macro, which combines the three aspects as an `if` (or the tertiary operator), a call to `print()`, and a call to `exit()`. The **STLSoft** libraries (and other libraries of mine discussed in the book) use their own assert macros, which can be (re)defined by the user in order to implement detection, reporting, and response in whatever manner they see fit.

CHAPTER 8

Constraints

I see, and I do not understand. I hear, and I forget. I do, and I remember.
—Proverb

The average man doesn't want to be free. He wants to be safe.
—Henry Louis Mencken

A constraint is a condition that one software element places on another, at compile time, in order for them to successfully interoperate. Constraints may be thought of as compile-time contract enforcements. I bang on at great length about constraints in Section 1.2 of *Imperfect C++*. My editor would be quite distraught if I were to repeat that material here, so I will just describe the two main types of constraints supported by the language and give a couple of examples of each.

8.1 Type System Leverage

The classic constraint given by Bjarne Stroustrup (in a post on comp.lang.c++.moderated) constrains a derived type `D` to be derived from a base class `B`. My version of it, from the **STLSoft** libraries, appears in Listing 8.1.

Listing 8.1 Definition of the `must_have_base` Constraints Class

```
template< typename D // Derived class
        , typename B // Base class
        >
struct must_have_base
{
public:
  ~must_have_base()
  {
    void (*p)(D*, B*) = constraints;
  }
private:
  static void constraints(D* pd, B* pb)
  {
    pb = pd;
  }
};
```

This constraint works by requiring that a pointer to the template parameter `D` can be assigned to a pointer to the template parameter `B`. Hence, `D` must be the same type as `B` or must be (publicly) derived from `B`.

We might use such a constraint if we were writing a function template and needed to ensure that the type was of a specific hierarchical relationship, as in the **MFCSTL** `c_str_ptr_null_CWnd_proxy` class shown in Listing 8.2. (This is an internal worker class. Don't worry too much about the hideous name; it is never uttered in client code.)

Listing 8.2 Use of a Constraint to Enforce a Parameter to a Constructor Template

```
// In namespace mfcstl
class c_str_ptr_null_CWnd_proxy
{
  . . .
public: // Construction
  template<typename W>
  c_str_ptr_null_CWnd_proxy(W const& w)
  {
    must_have_base<W, CWnd>() // Ensure W is CWnd or derived from it
    . . .
```

This type of constraint works by using the C++ type system to enforce the assumptions made by the designer of the code.

8.2 Static Assertions

The other mechanism for effecting constraints is to use *static assertions*. A static assertion is a macro that, when expanded, creates a piece of benign code that is compilable when the assertion condition is true but causes a compilation error when the condition is false. There are several possible mechanisms: switch statements, bit fields, and array declarations. I use the latter.

Consider the static assertion (shown in Listing 8.3) from the `dl_call()` library (Section 16.6) that enforces that the three function arguments `a0`, `a1`, and `a2`, which are to be passed to the invocation of the dynamically loaded function, are compatible with the invocation. To be compatible, they must be of fundamental type, of pointer type, or of function pointer type, or the user must explicitly state that they are compatible by defining a specialization of the `is_valid_dl_call_arg` traits class. (The first three are stock **STLSoft** traits classes; the latter is part of the `dl_call()` library.)

Listing 8.3 Use of Constraints to Restrict Types in a Generic Function Template

```
// In namespaces unixstl / winstl
template< typename R, typename L, typename FD
        , typename A0, typename A1, typename A2
        >
R dl_call(L const& library, FD const& fd, A0 a0, A1 a1, A2 a2)
```

```
{
  STLSOFT_STATIC_ASSERT(is_fundamental_type<A0>::value ||
                        is_pointer_type<A0>::value ||
                        is_function_pointer_type<A0>::value ||
                        is_valid_dl_call_arg<A0>::value);
  . . . // Same for A1 and A2
  return dl_call_MOD<R>(. . . , a0, a1, a2);
}
```

If the function is used with types that do not conform to those constraints, the `STLSOFT_STATIC_ASSERT()` will resolve to an attempt to declare an array of length *0* (or *-1*, depending on the compiler), which will cause compilation to fail, thereby rejecting the attempt to circumvent the restrictions placed on the use of `dl_call()` by its designer. Who would be me. Bad kitty!

CHAPTER 9

Shims

It is change, continuing change, inevitable change, that is the dominant factor in society today. No sensible decision can be made any longer without taking into account not only the world as it is, but the world as it will be.

—Isaac Asimov

Nothing is more admirable than the fortitude with which millionaires tolerate the disadvantages of their wealth.

—Nero Wolfe

9.1 Introduction

Shims are simple but powerful tools for generalized programming that afford moderate to high cohesion with minimal—often zero—coupling. The *shim* concept describes an unbounded suite of function overloads that generalize between (aspects of) conceptually related but physically unrelated types and exhibit the following characteristics.

- *Name*: The shim name constitutes the function name *and* the namespace within which they reside. For example, the full name for the **c_str_ptr** shim is `stlsoft::c_str_ptr`.
- *Intent*: The intent indicates the common purpose of all functions constituting the shim. For example, the intent of the **c_str_ptr** shim is to translate the instances of its supported types into non-*NULL* pointers to nul-terminated character strings representing their string form.
- *Category*: The shim category describes the naming convention, behavior, and constraints on the use of the shim.
- *Ostensible return type*: All functions of a given shim must return the ostensible return type or a type that is implicitly convertible to it.

Shims come in four primary forms, the categories of *attribute shims* (Section 9.2.1), *control shims*, *conversion shims* (Section 9.2.2), and *logical shims*. Shim categories may also be combined into composite categories, the most significant of which is *access shims* (Section 9.3). In this chapter, we will cover only those categories used in Parts II and III to increase the flexibility of the STL extensions: *attribute shims*, *conversion shims*, and *string access shims* (Section 9.3.1).

The full definitions of all currently defined shim categories and all known established shims are available online as part of the **STLSoft** documentation. And my next book, *Breaking Up the Monolith*, will contain a wealth of information about shims and the different manners in which they can be put to use. If you're interested in seeing how far generic programming can be pushed while keeping coupling and performance costs to a minimum, I encourage you to get a copy. (Of course, I'm probably not the most impartial advisor.)

9.2 Primary Shims

9.2.1 Attribute Shims

Attribute shims retrieve attributes or states of instances of the types for which they are defined. An example is the **get_ptr** shim. Listing 9.1 shows some of the constituent functions of this shim. (Note that they are not defined in the same header file in actuality.)

Listing 9.1 Definitions of Common `get_ptr` Overloads

```
// In namespace stlsoft
template <typename T>
T* get_ptr(T* p)
{
  return p;
}
template <typename T>
T* get_ptr(std::auto_ptr<T> const& p)
{
  return p.get();
}
inline void const* get_ptr(unixstl::memory_mapped_file const& mmf)
{
  return mmf.memory();
}
```

Users who wish to write code to work with pointers and smart pointers alike simply code in terms of `stlsoft::get_ptr()`, without worrying about whether they need to invoke `get()`, `memory()`, or some other method to get hold of the real pointer.

9.2.2 Conversion Shims

Conversion shims convert instances of the range of types for which they are defined to a single type. For example, we could define a **to_string** shim, whose ostensible return type is `const std::string`. The following would all be viable overloads:

```
// In namespace my_shims
inline std::string to_string(char const *s)
{
  return std::string(s);
}
inline std::string const& to_string(std::string const &s)
{
  return s;
}
```

```
inline std::string to_string(int i)
{
  char buf[21];
  return std::string(&buf[0], size_t(::sprintf(&buf[0], "%d", i)));
}
```

Note that the second overload returns the nonmutating (`const`) reference passed into it; it does not return a `std::string` instance. This is okay because any use that client code might wish to make of an instance of the ostensible return type `const std::string`, it can equally make with a `const std::string&`, as in the following:

```
int           i   = 101;
char const*   cs  = "abc";
std::string   str("def");
::puts(my_shims::to_string(i).c_str());   // OK
::puts(my_shims::to_string(cs).c_str());  // OK
::puts(my_shims::to_string(str).c_str()); // OK
```

It even works, thanks to the *curious untemporary reference* (Chapter 4), in the following case:

```
std::string const& str1 = my_shims::to_string(i);
std::string const& str2 = my_shims::to_string(cs);
std::string const& str3 = my_shims::to_string(str);
::puts(str1.c_str());
::puts(str2.c_str());
::puts(str3.c_str());
```

`str1` and `str2` are curious untemporary references; `str3` is a plain old reference.

There are cases where the use of conversion shims may not be as straightforward, when the ostensible return type does not have value semantics. For example, we might define a conversion shim **to_file_handle**, whose ostensible return type is `int`. (I say *might* since this would be an altogether too high-level and too powerful, if not plain foolish, generalized tool to have at one's disposal. It's pure pedagogy: Do not be tempted to plug this baby into your code!) Conversion from a file handle to a file handle would be very straightforward:

```
// In namespace my_shims
inline int to_file_handle(int h)
{
  return h;
}
```

So far, so good. But now imagine that we want to convert from a file name (`char const*`). This would require the shim overload to return an instance of a temporary class, which implicitly converts to `int` and which manages a file handle to the file of the given name (see Listing 9.2).

Listing 9.2 An Example `to_file_handle` Conversion Shim and Its Supporting Class

```
// In namespace my_shims
class shim_file_handle
{
public: // Construction
  shim_file_handle(char const* fileName); // Calls ::open()
  ~shim_file_handle() throw();            // Calls ::close()
public: // Conversion
  operator int () const;  // Implicit conv to ostensible return type
private: // Member Variables
  int m_handle;
};
shim_file_handle to_file_handle(char const* fileName)
{
  return shim_file_handle(fileName);
}
```

Suppose that we have a function that takes a file handle and does something with it, perhaps converting its contents to a list of strings:

```
std::list<std::string> readlines(int hFile);
```

Using the **to_file_handle** shim, we can generalize `readlines()` with a function template, as follows:

```
template <typename F>
std::list<std::string> readlines(F const& fileDescriptor)
{
  return readlines(my_shims::to_file_handle(fileDescriptor));
}
```

Now we can read the lines of any file whether it's via its file handle (`int`):

```
int fh = ::open("myfile.txt", . . . );
std::list<std::string>  lines1 = readlines(fh);
```

or file name (`char const*`):

```
std::list<std::string>  lines2 = readlines("myfile.txt");
```

It's important to realize that `readlines()` is now fully generalized and complete. To extend the range of types with which it can work, we need only to define other `to_file_handle()` shim overloads (in the `my_shims` namespace). If we were using a third-party library that had a `File` class, which provides a file handle via, say, a `raw_handle()` method, we could make the class compatible with `readlines()`:

```
// In namespace ThirdPartyPeople
class File;

// In namespace my_shims
inline int to_file_handle(ThirdPartyPeople::File& file)
{
  return file.raw_handle();
}

// Client code
ThirdPartyPeople::File  file("myfile.ext");
std::list<std::string>  lines1 = readlines(file);
```

Furthermore, the `File` type is now compatible with any other functions or classes that manipulate file handles via the **to_file_handle** shim.

That's a good demonstration of the power of shims. However, all this power and wonder can be easily broken if we don't respect the fact that the ostensible return type of the **to_file_handle** shim does not have value semantics. If we'd written the function template overload of `readlines()` in the following way, the behavior would be undefined when used with a file descriptor type of `char const*` (or any other type whose shim returns an object convertible to `int`, rather than `int` itself):

```
template <typename F>
std::list<std::string> readlines(F const& fileDescriptor)
{
  int hFile = to_file_handle(fileDescriptor);
  return readlines(hFile); // Handle has already been closed!
}
```

By the time control has reached the call to `readlines(int)`, the file handle retrieved from `to_file_handle(char const*)` is invalid because the instance of `shim_file_handle` that it actually returned has come and gone and closed the file it had opened on our behalf.

This issue happens only when the return value is stored for later use. Consequently, the use of conversion shims is subject to one strict rule.

Rule: The return values from a conversion shim must never be used outside the expression in which the shim is invoked, except where the ostensible return type has value semantics.

9.3 Composite Shims

Composite shims are a combination of other shim categories. Composite shims are subject to the most exacting restrictions of their constituent types. The only composite category we're interested in throughout this book is the composite of the attribute and conversion categories: the *access shim*.

9.3.1 String Access Shims

The main access shims used throughout the **STLSoft** libraries, in many related libraries (such as **FastFormat**, **Open-RJ**, **Pantheios**, and **recls**), and in much of my commercial work over the last half decade or so are the *string access shims*. We will see these used throughout the book to significantly boost the flexibility of the components described. There are three main shims.

1. The **c_str_ptr** shim and its character-encoding-specific variants **c_str_ptr_a** and **c_str_ptr_w**. Overloads of the functions for these shims return a non-*NULL* pointer to a nul-terminated C-style string representing the converted/translated form of the type for which they are defined. The ostensible return type of **c_str_ptr_a** is `char const*`; for **c_str_ptr_w**, it's `wchar_t const*`; for **c_str_ptr**, it is as appropriate to the type, that is, `c_str_ptr(std::string const*)` returns `char const*`, and `c_str_ptr(std::wstring const*)` returns `wchar_t const*`.
2. The **c_str_data** shim and its character-encoding-specific variants **c_str_data_a** and **c_str_data_w**. Overloads of the functions for these shims return a pointer to a (not necessarily nul-terminated) C-style string representing the converted/translated form of the type for which they are defined. The string does not have to be nul-terminated because **c_str_data** is always used in concert with **c_str_len**. The ostensible return types are the same as for **c_str_ptr** (and variants).
3. The **c_str_len** shim and its character-encoding-specific variants **c_str_len_a** and **c_str_len_w**. Overloads of the functions for these shims return the string length of the converted/translated form returned by the related **c_str_ptr** (or variant) or **c_str_data** (or variant) shim overload. The ostensible return type is `size_t` (or equivalent). (The **c_str_len** shim and variants are actually pure attribute shims, but since they are always used in conjunction with either **c_str_data** or **c_str_ptr**, I find it easier to refer to them all as the string access shims.)

It's hard to overstate the power of string access shims. Consider the **Pantheios** logging API library, which uses the **c_str_data_a** and **c_str_len_a** shims in its application-layer function template suites (which are auto-generated from a script). Listing 9.3 shows the definition of the three-parameter overload of the `log_ALERT()` function template.

Listing 9.3 Use of String Access Shims in Pantheios Application-Layer Functions

```
template <typename T0, typename T1, typename T2>
int log_ALERT(T0 const& v0, T1 const& v1, T2 const& v2)
{
  if(!isSeverityLogged(PANTHEIOS_SEV_ALERT))
  {
    return 0;
  }
  else
  {
    return log_dispatch_3(PANTHEIOS_SEV_ALERT
            , stlsoft::c_str_len_a(v0), stlsoft::c_str_data_a(v0)
            , stlsoft::c_str_len_a(v1), stlsoft::c_str_data_a(v1)
            , stlsoft::c_str_len_a(v2), stlsoft::c_str_data_a(v2));
  }
}
```

The use of separate template types (`T0`, `T1`, and `T2`) for the function parameters combined with the application of the string access shims means that **Pantheios** supports effectively infinite extensibility, with 100% type safety. Furthermore, the elicitation of the parameter values, and all conversion, allocation, copying, and concatenation into a composite output string, is done only *after* it has been ascertained that the given severity level (*`PANTHEIOS_SEV_ALERT`*) is currently being logged. Consequently, **Pantheios** has virtually no performance cost when a given logging level is disabled.

By using the **c_str_data_a** and **c_str_len_a** shims, the **Pantheios** application layer is compatible with *any* type for which these shims are defined, resulting in very straightforward application code, as in the following:

```
void func(std::string const& s1, char const* s2, struct tm const* t)
{
  pantheios::log_DEBUG("func(", s1, ", ", s2, ", ", t, ")");
  . . .
```

or:

```
catch(std::exception& x)
{
  pantheios::log_CRITICAL("Things have gone pear shaped: ", x);
}
```

or:

```
VARIANT   var1  = . . .
CWindow&  wnd1  = . . .
HWND      hwnd2 = . . .

pantheios::log_ERROR("var=", var1, "; wnd=", wnd1, "; hwnd=", hwnd2);
```

If you pass a type for which the requisite shim function overloads are not defined (or are not visible, if you forgot a `#include`), you'll get a pretty lucid (by template library standards) error message, noting that none of the *N* overloads it knows about are compatible with your type.

At the time of writing, the **STLSoft** libraries come with string access shims for all of the following standard or third-party types: `char const*`/`char*`, `wchar_t const*`/`wchar_t*`, `std::string`, `std::wstring`, `std::exception`, and `struct tm`; `struct in_addr`; **ACE**'s `ACE_CString`, `ACE_WString`, `ACE_INET_Addr`, and `ACE_Time_Value`; **ATL**'s `CComBSTR`, `CComVARIANT`, and `CWindow`; **COM**'s `BSTR`, `GUID`, and `VARIANT`; **MFC**'s `CString` and `CWnd` (and other window classes); **UNIX**'s `struct dirent`; and **Windows**' `FILETIME`, `HWND`, `LSA_UNICODE_STRING`, and `SYSTEMTIME`.

Further, every type within the **STLSoft** libraries that is a string, or could have a meaningful representation as a string, also has string access shims defined. Listing 9.4 shows just a few of the

c_str_ptr shim function definitions. Note the overloads for `FILETIME`, `SYSTEMTIME`, and `CWnd`: these are the conversion shims and return instances of classes that are implicitly convertible to `char const*`.

Listing 9.4 Declaration of Various Stock String Access Shims

```
// All in namespace stlsoft

// From stlsoft/shims/access/string.hpp
char const* c_str_ptr(char const*);
char const* c_str_ptr(std::string const&);
char const* c_str_ptr(stlsoft::basic_simple_string<char> const&);
char const* c_str_ptr(stlsoft::basic_static_string<char> const&);

// From winstl/shims/access/string/time.hpp
stlsoft::basic_shim_string<char> c_str_ptr(FILETIME const& t);
stlsoft::basic_shim_string<char> c_str_ptr(SYSTEMTIME const& t);

// From unixstl/shims/access/string/dirent.hpp
char const* c_str_ptr(struct dirent const* d);
char const* c_str_ptr(struct dirent const& d);

// From mfcstl/shims/access/string/cwnd.hpp
c_str_ptr_CWnd_proxy c_str_ptr(CWnd const& w);
```

As long as you don't forget that the string access shims are subject to the rule for conversion shims and that their ostensible return types (`char const*` and `wchar_t const*`) do not have value semantics, you can increase the flexibility and performance of your components without any problems.

Rule: The return value from an access shim must never be used outside the expression in which the shim is invoked.

Other libraries and commercial projects in which I've been involved over the last few years also define overloads of the string access shims in the `stlsoft` namespace and thereby automatically gain a good degree of interoperability with **STLSoft** and with each other. It is important to note that there need be *no coupling* with, or even inclusion of, any part of **STLSoft** to do this because namespaces are open for extension (Section 5.3).

Tip: Use string access shims for high cohesion with little or no coupling.

The upshot of all this is that libraries, such as **FastFormat** and **Pantheios**, that use string access shims have a huge amount of genericity out of the box. Consequently, users can get up and running with little effort; users need only add their own extensions, in the form of additional shim function overloads, as they need to incorporate new types. Use of the shims means that **Pantheios** satisfies the principles of *Composition*, *Diversity*, *Economy*, *Extensibility*, *Generation*, *Least*

Surprise, *Modularity*, *Most Surprise*, *Robustness*, and *Separation* in one fell swoop! (If you feel a little disappointed that we're covering comparatively little of what is a very powerful concept in this book, I hope you'll forgive me. This is, after all, a book about STL extension, in which shims play only a small part, and space is at a premium. It's not a cynical marketing ploy to incline you to buy my next book, *Breaking Up the Monolith*, in which the full spectrum and power of shims is written large. Honest!)

CHAPTER 10

Duck and Goose, or the Whimsical Bases of Partial Structural Conformance

Sometimes it kind of bothers me that our brains are so good at optimizing themselves for the task at hand. They get really good at discarding knowledge that is no longer necessary.

—Sean Kelly

It's harder to read code than to write it.

—Joel Spolsky

10.1 Conformance

10.1.1 Named Conformance

Classically (by which I mean before template generics), types were verified to have the appropriate requirements for an operation by having the same *name* as the types that the operation requires. This is known as *Named Conformance*. Although it seems blatantly obvious, the insight comes when we consider the effect of inheritance. As we all know from those $2,000-a-day courses on object orientation that our employers *kindly* forced us to attend in the 1990s, a type `B` that (publicly) inherits from a type `A` is deemed to be of type `A`. In the vernacular, `B` *is-a* `A`.

```
class A
{};
void func(A const& a);

class B
  : public A
{};
```

Consequently, if we have an operation defined in terms of (references or pointers to) `A`, we can use that operation with instances of `B`. As a refinement of `A`, `B` can provide different behaviors for some or all of the operations it has in common with `A`. In C++, we call this polymorphism, and it is effected by using the virtual function mechanism (which, in all C++ compilers I've ever used, is implemented using virtual function tables and the so-called *vptr*).

```
class A
{
public:
  virtual void print() const    { ::puts("A"); }
};
class B
  : public A
{
public:
  virtual void print() const    { ::puts("This is a buzzy B!"); }
};
void func(A const& a)
{
  a.print();
}

A   a;
B   b;
func(a); // OK. a is of type A
func(b); // OK. b is of type B, which is-a A
```

The matching of requested operation and type is based strictly on name (which can be said to incorporate the names of parent classes), which means that two identical types may not be interchanged. Hence:

```
class C
{
public:
  virtual void print() const    { ::puts("C"); }
};

C   c;
func(c); // Error! C is not an A.
```

Named Conformance provides assurance that the type of the instance on which an operation is to be applied conforms to the type expected by that operation. The disadvantage is that an operation is available only to types related by inheritance from a common type.

Note that, of course, *Named Conformance* is not proof against subversion. We might define a type D as follows:

```
class D
  : public A
{
public:
  virtual void print() const  { ::exit(EXIT_FAILURE); }
};
```

But this is not the kind of thing we can reasonably expect a usable programming language to police for us. As a reviewer of *Imperfect C++* sagely offered regarding some of my more corrugated attempts at enforcing safety: “There’s no accounting for Machiavelli.” I think this is very good advice since it’s a game without end. So, by all means, follow this tip.

Tip: Expend all reasonable efforts when writing libraries to help prevent users from making mistakes in using them, but waste no time attempting to prevent deliberate subversion of correctness.

10.1.2 Structural Conformance

An alternative principle is known as *Structural Conformance*, one that is very important in generic programming. It says that things that have the same structure—name and form—are expected to act the same. Name and often type are not even considered. For example, consider the following generic operation:

```
template <typename T>
void log(T const& t, int level)
{
  if(T::defaultLevel >= level)
  {
    std::cout << t.c_str() << std::endl;
  }
}
```

There is no constraint on the type passed to the function in its signature. It is a nonmutating reference to (`const`) `T`, where `T` is the template parameter. The requirements on `T` within the function body are that the type has a member called `defaultLevel`, which can enter into a greater-than-or-equal comparison with an `int`, and that a member function `c_str()` may be invoked on an instance. The compiler determines the suitability of any type for which this function template will be instantiated solely on those two requirements. Hence, both the following types may be used with `log()`:

```
class M
{
public:
  static const int defaultLevel = 10;
  char const* c_str() const     { return "M"; }
};
class N
{
public:
  static const int defaultLevel = 5;
  char const* c_str() const     { return "N"; }
};
```

```
M   m;
N   n;
log(m, 8);
log(n, 8);
```

Both define a defaultLevel as a member constant and a c_str() method that returns a nul-terminated string.

10.1.3 Duck and Goose

Structural Conformance is sometimes known as the Duck Rule.

The Duck Rule: If it looks like a duck, walks like a duck, and quacks like a duck, it is a duck.

More formally, the Duck Rule has been applied in relation to Ruby's type handling, in what's called *Duck Typing*. Rather than using an instance's type to determine whether a given method may be invoked on it, the presence or absence of the method is the sole determinant. For Ruby, this is carried out at runtime, using Ruby's reflection facilities, rather than at compile time, as it is in C++.

However, this looser matching mechanism, while providing generic programming its great power, also presents many pitfalls. I'm confident that most of you gentle readers would have imagined such forms for the defaultLevel and c_str() members in the example classes M and N. However, there is a much greater degree of syntactic conformance possible. Consider the following classes. First a union in which defaultLevel is an enumerator and c_str is a member variable of the type of a member function class:

```
union X
{
  enum { defaultLevel = 10 };
  struct c_str_fn
  {
    int operator ()() const      { return 0; }
  };
  c_str_fn    c_str;
};
```

Now a structure that has a static defaultLevel and a static c_str() method:

```
struct Y
{
public:
  static int           defaultLevel;
  static char const*   c_str()     { return 0; }
};
int Y::defaultLevel = 0;
```

It could even be the following construction, which *Imperfect C++* readers will recognize defines `defaultLevel` as a static read-only method-based *property* that, when read, invokes the private `get_defaultLevel()` method. I think that most C++ practitioners would be surprised to learn that `log()`'s conditional expression might invoke a function of `T`.

```
struct Z
{
private:
  static int get_defaultLevel();
public:
  static stlsoft::static_method_property_get< int, int
                                            , Z, get_defaultLevel
                                            >   defaultLevel;
  int c_str() const;
};
stlsoft::static_method_property_get<int, int
                                  , Z, Z::get_defaultLevel
                                  >  Z::defaultLevel;
```

Note also that the `c_str()` method returns not a string but an `int`. Since the `std::basic_ostream`'s insertion operator is overloaded for `int` as well as `char const*`, this works seamlessly, but, again, this is unexpected. However, all of `X`, `Y`, and `Z` conform with the requirements of `log()`, and all compile and execute correctly with it.

I've gone to some lengths here to demonstrate how badly *Structural Conformance* can go astray. Such instances of nonconformance can seem abstruse to the point of whimsy, but these are very real risks. The weakness of *Structural Conformance* is that it is easy to have structurally conformant types that do semantically nonconformant things. Because the compiler resolves templates based on symbol name and type, the issue of semantics is entirely overlooked. This is not helped by poorly chosen and inconsistently used verbs and adjectives for method names; for example, the standard uses `erase()` and `clear()` as verbs but `empty()` as an adjective. But that's just the (obvious) tip of the iceberg. In nontrivial applications of template techniques, it is all too easy to be unaware that the Duck is not quacking happy. We'll see good examples of this when we consider the attempts to adapt iterators with incompatible element reference categories in Part III. One consequence of relying on the Duck Rule is that generic template components must use constraints and contract enforcements wherever appropriate and must be well tested.

Recommendation: Above all other types of programming, the development of STL extensions must be accompanied by constraints, contract enforcements, and comprehensive unit-test suites.

As a simple mnemonic to keep the dangers of *Structural Conformance* to the fore, I have coined the Goose Rule.

The Goose Rule: It may look like a duck, it may walk like a duck, and sometimes it may quack like a duck, but a duck it is not. Assuming it's a duck may make you a goose!

Naturally, we'll see instances where we may fall foul of the rule, throughout this book and Volume 2. (My apologies, gentle readers, but I can't pass up an opportunity to punnet, I mean, pun-it! Blame my mum, otherwise a good egg, from whom I have inherited this awful habit. We drive the rest of our family collectively mad when we're cooped up together.)

10.2 Explicit Semantic Conformance

You may be wondering at this point whether the characteristics of *Named Conformance* and *Structural Conformance* may be combined to afford the safety of the former and the flexibility of the latter. The answer is that they can, in several ways.

10.2.1 Concepts

As you may know, many algorithms in the standard library detect iterator refinements by overloading on a type characteristic of the given iterator instance. For example, the `std::distance()` algorithm implementation would be along the lines shown in Listing 10.1.

Listing 10.1 Selection of Worker Functions by Iterator Category

```
// In namespace std
template <typename I>
typename iterator_traits<I>::distance_type
  distance_impl(I from, I to, random_access_iterator_tag)
{
  return to - from;
}
template <typename I>
typename iterator_traits<I>::distance_type
  distance_impl(I from, I to, input_iterator_tag)
{
  typename iterator_traits<I>::distance_type  n = 0;
  for(; from != to; ++from, ++n)
  {}
  return n;
}
template <typename I>
typename iterator_traits<I>::distance_type
  distance(I from, I to)
{
  return distance_impl( from, to
              , typename iterator_traits<I>::iterator_category());
}
```

Listing 10.2 shows the inheritance relationships of the standard iterator category tag types.

Listing 10.2 Inheritance Relationships of the Standard Iterator Category Tag Classes

```
struct input_iterator_tag
{};
struct output_iterator_tag
{};
struct forward_iterator_tag
  : public input_iterator_tag
  , public output_iterator_tag
{};
struct bidirectional_iterator_tag
  : public forward_iterator_tag
{};
struct random_access_iterator_tag
{};
```

Each nonpointer iterator type uses one of these classes to define its member type `iterator_category`, which is used up by the general form of `std::iterator_traits` to define its own member type `iterator_category`. Partial specializations of `std::iterator_traits` for pointers define the member type `iterator_category` as `std::random_access_iterator_tag`. (This is covered in detail in Chapter 38.) Hence, the compiler can select between the two overloads of the worker function `distance_impl()`, according to whether the iterator type `I` is a *random access iterator* or any of the lesser refinements.

10.2.2 Member Type Tagging

Another way in which requirement fit can be definitively established is by the detection of member types. For example, since we've defined an *STL collection* (Section 2.2) as providing iterator ranges via `begin()` and `end()` methods, we may presume to identify such a type to have one or both of the member types `iterator` and `const_iterator`. Using the techniques for detection of member types described in Chapter 13, we can express constraints (Chapter 8) on the types that are acceptable to our components. This technique is widely used throughout this book and Volume 2, in particular for determining the characteristics of iterator adaptor base types (Chapter 41).

10.2.3 Shims

As we'll see throughout this book and Volume 2, *shims* (Chapter 9) extend the notion of *Structural Conformance* to include an explicit semantic for each shim. For example, in defining an overload for the **c_str_ptr** shim for a given type, a user will ensure that the function takes a nonmutating reference to the type and returns either a non-*NULL*, nul-terminated character string containing the equivalent string representation of the instance or a temporary instance of an object that is implicitly convertible to such. Thus, the name of the shim carries with it the semantics. Naturally, this is achieved only by voluntary adherence to the semantics of a given shim, so shims must be named appropriately; hence the distinct naming conventions for the different shim concepts discussed in Chapter 9.

10.3 Intersecting Conformance

The *Law of Leaky Abstractions* (Chapter 6) and the Goose Rule (Section 10.1.3) clash, in what C++ developers may be forgiven for seeing as a paralyzing contradiction. The former tells us that abstractions always leak; the latter that an imperfect abstraction will bite you. However, I think the conflicting imperatives can be tamed in an approach I call *Intersecting Conformance,* which is the fundamental (if only lately formalized) design principle of the **STLSoft** libraries.

Where two or more entities exhibit apparent *Structural Conformance*, we embrace the leaky abstraction and identify what we know to be common. Then we define the abstraction to cover only the intersecting semantically correct *Structural Conformance*, with whatever minor judicious adjustments are appropriate and do not violate the Goose Rule.

This methodology would have left most monolithic adaptations on the strategy-room floor, but it does mean that there are frayed borders to the abstractions, meaning that users who need to use functionality supported by only a subset of components underlying a given abstraction need to step out of the abstraction, thereby mixing the abstraction and the underlying components. Some consider this messy, inelegant, even jejune, but I believe that when we throw C++'s raison d'être, performance, into the mix, it is not only acceptable, it is often the only sensible approach for the implementation of C++ components that exist at a low to mid level of abstraction.

So what does all this mean to the poor programmer attempting to produce robust, efficient, portable code? The answer is simple to state: Design and program in the generic where abstractions are sufficiently clear and efficient, and in the specific where not. Putting this into practice, of course, is more easily said than done. Foremost, it relies on software developers to be experienced and to care about the quality of their work and on library writers to do the same. There's no substitute to be found in a nice, clean, pathologically restrictive, and hopelessly slow framework.

Chapter 11

RAII

I like to tidy up, because it gives me more room to play.

—Ben Wilson

Will you help me to be less naughty, Daddy?

—Harry Wilson

It is reasonable to assume that all C++ practitioners are aware of *Resource Acquisition Is Initialization* (RAII), even if they do not use that term. Essentially, it is about the business of getting hold of a resource—object, memory, file handle, and so on—in the constructor of an object and releasing it in the destructor. Classes that do this employ RAII and are often referred to as wrapper classes.

In Chapter 3 of *Imperfect C++*, I defined a taxonomy of RAII that I have found very useful in my work. This involves the permutations between two characteristics: mutability and resource source.

11.1 Mutability

If a wrapper class provides additional facilities for instances to be assigned a new resource, I classify this as exhibiting *mutable RAII*. If not, I classify it as *immutable RAII*.

Immutable RAII is the simplest to work with since it means that no assignment methods need be provided, either for new resource assignment or for copy assignment. It also means that the destructor can always assume the encapsulated resource is valid. The `glob_sequence` class, discussed in Chapter 17, exhibits immutable RAII.

Conversely, classes providing mutable RAII require most, or all, of the following: default/null construction, copy construction and assignment, and resource assignment. Significantly, they must test—in the destructor and in any `close()` method—whether the resource is "null" before releasing it.

11.2 Resource Source

The second characteristic of classes providing RAII is the manner in which they get hold of the resources they are managing. A class such as `std::string` exhibits *internally initialized RAII*: It creates the resource—a memory buffer containing the characters—that it manages. That resource never sees the outside world. Conversely, a class such as `std::auto_ptr` exhibits *externally initialized RAII*: The resource it manages is passed to it from client code.

Classes exhibiting internally initialized RAII tend to be easier to implement, but more limiting, since their mechanisms for acquiring the resource are already prescribed and fixed. However, they tend to be simpler to use, or, more properly, harder to misuse because there is little or no chance of making a mistake that might lead to a resource leak.

Thankfully, most STL extensions tend to be internally initialized. For many collections that are not containers—they do not own their elements—it can be difficult or practically impossible to facilitate value semantics, so, unlike STL containers, they tend to offer immutable RAII.

CHAPTER 12

Template Tools

Sticks and stones may occur in nature, but not in the form of levers and fulcrums.

—Henry Petroski

I can't sleep nights till I find out who hurled what ball through what apparatus.

—Dr. Niles Crane, *Frasier*

12.1 Traits

Users of the STL are unlikely to have missed the use of traits, principally `char_traits` and `iterator_traits`, in the implementation of the library. A traits class template (also known just as a traits class) defines a protocol for describing a type, and specializations of the traits class describe a specific type. Partial specializations describe a set of types that share common characteristics.

Some traits classes are all about detecting features of types, from which they define their own member values and member types. For example, the `stlsoft::printf_traits` assists the user in defining format strings to use for a given integral type with the `printf()` family functions and the maximum number of characters that will be printed (Listing 12.1).

Listing 12.1 `printf_traits` **Primary Template**

```
// In namespace stlsoft
template <typename T>
struct printf_traits
{
  enum
  {
      size_min // Num chars required for min value (+ nul)
    , size_max // Num chars required for max value (+ nul)
    , size     // The maximum of size_min and size_max
  };
  static char const*    format_a();
  static wchar_t const* format_w();
};
```

There are specializations for the standard integral types, as well as compiler-specific integral types. Listing 12.2 demonstrates why the traits class is very useful when writing generic code using the `printf()` family, since compilers differ in whether they think *`%lld`* or *`%I64d`* is the appropriate format string for a signed 64-bit integer. The values of `size_min` and `size_max` are determined by applying `sizeof()` to the stringized forms of the minimum and maximum constants (which have to use `#defines` to support this).

Listing 12.2 `printf_traits` **Specialization for a Signed 64-Bit Integer**

```
#define STLSOFT_STRINGIZE_(x)      # x
#define STLSOFT_STRINGIZE(x)       STLSOFT_STRINGIZE_(x)
#define STLSOFT_PRTR_SINT64_MIN    -9223372036854775808
#define STLSOFT_PRTR_SINT64_MAX    9223372036854775807

template <>
struct printf_traits<sint64_t>
{
  enum
  {
      size_min = sizeof(STLSOFT_STRINGIZE(STLSOFT_PRTR_SINT64_MIN))
    , size_max = sizeof(STLSOFT_STRINGIZE(STLSOFT_PRTR_SINT64_MAX))
    , size     = (size_min < size_max) ? size_max : size_min
  };
  static char const* format_a()
  {
#if defined(STLSOFT_CF_64_BIT_PRINTF_USES_I64)
    return "%I64d";
#elif defined(STLSOFT_CF_64_BIT_PRINTF_USES_LL)
    return "%lld";
#else
# error Further compiler discrimination is required
#endif /* printf-64 */
  }
  static wchar_t const* format_w(); // Like format_a(), but with L""
};
```

A much more sophisticated example is `stlsoft::adapted_iterator_traits` (Chapter 41), which underpins the flexibility of several iterator adaptors in detecting `const`-ness, iterator category, and element reference category (Chapter 3) and in employing *inferred interface adaptation* (Chapter 13) to “fix” missing iterator types.

Other traits classes are primarily about functionality, where specializations define implementations to functions declared in the primary template. A clear example of this is `winstl::drophandle_sequence_traits`, which has the following minimalist definition:

```
// In namespace winstl
template <typename C>
struct drophandle_sequence_traits
{
  static UINT drag_query_file(HDROP hdrop, UINT index
                             , C* buffer, UINT cchBuffer);
};
```

The sole purpose of this traits class is to invoke either of the Windows API functions `DragQueryFileA()` or `DragQueryFileW()`, as appropriate to the character type `C`.

Most traits classes provide both type/value information and tailored functionality. In the STL, components that use character traits usually do so via a template parameter that is defaulted to a standard traits class. For example, the output stream iterator class `ostream_iterator` has the following definition:

```
// In namespace std
template< typename V                 // The inserted type
        , typename C = char          // The character encoding
        , typename T = char_traits<C> // The traits type
        >
class ostream_iterator;
```

In this case, the traits parameter, `T`, is defaulted to the specialization of `std::char_traits<C>` because most users are unlikely to need to provide a different traits class. In practice, I've never come across a use that didn't accept the default.

When writing STL extensions, the use of traits becomes a mainstream issue. As with standard library components, the traits parameter is defaulted on the expectation that most of the uses to which the component will be put are covered by the default. Where specific or unanticipated behavior is required, the user has a mechanism for customizing behavior without having to take scissors to the code. At the time of writing, there are 56 traits classes in the **STLSoft** libraries; the following subsections describe those used later in this book.

12.1.1 `base_type_traits`

One of my main workers is the traits class `base_type_traits`, whose primary template is shown in Listing 12.3. (It is my convention to represent enumerators that act as distinct member constants in separate enumerations, to distinguish them from regular enumerators representing numbered choices, which are grouped; see Section 17.3.1 for an example.)

Listing 12.3 Primary Template of `base_type_traits`

```
// In namespace stlsoft
template <typename T>
struct base_type_traits
{
  enum { is_pointer          = 0 };
  enum { is_reference        = 0 };
  enum { is_const            = 0 };
  enum { is_volatile         = 0 };
  typedef T           base_type;
  typedef T           cv_type;
};
```

As its name suggests, base_type_traits started out life as a mechanism for stripping a *cv*-qualified type down to its base type. It has evolved to be a detector of various features of a type, such as whether it is a pointer, a reference, const, and so on. The useful work is performed by the partial specializations. Listing 12.4 shows the definition of a couple of these to illustrate the mechanism.

Listing 12.4 Partial Specializations of base_type_traits

```
template <typename T>
struct base_type_traits<T volatile*>
{
  enum { is_pointer           =   1 };
  enum { is_reference         =   0 };
  enum { is_const             =   0 };
  enum { is_volatile          =   1 };
  typedef T            base_type;
  typedef T volatile   cv_type;
};
template <typename T>
struct base_type_traits<T const&>
{
  enum { is_pointer           =   0 };
  enum { is_reference         =   1 };
  enum { is_const             =   1 };
  enum { is_volatile          =   0 };
  typedef T            base_type;
  typedef T const      cv_type;
};
. . . // And so on, for other permutations of const and/or volatile
```

This class is used in several components, including comstl::enumerator_sequence (Section 28.5) and stlsoft::member_selector_iterator (Section 38.3).

12.1.2 sign_traits

sign_traits is used for evaluating the equivalent signed and unsigned forms of integral types and the sign-opposite type. Listing 12.5 shows the primary template and two of its specializations.

Listing 12.5 Primary Template and Two Specializations of the sign_traits **Traits Class**

```
// In namespace stlsoft
template <typename T>
struct sign_traits;

template <>
struct sign_traits<sint32_t>
```

```
{
  typedef sint32_t    type;
  typedef sint32_t    signed_type;
  typedef uint32_t    unsigned_type;
  typedef uint32_t    alt_sign_type;
};
template <>
struct sign_traits<uint32_t>
{
  typedef uint32_t    type;
  typedef sint32_t    signed_type;
  typedef uint32_t    unsigned_type;
  typedef sint32_t    alt_sign_type;
};
```

There are two significant aspects to note. First, the primary template is not defined, merely declared. This means that it can be used only with types for which explicit specializations, such as the two shown, are defined.

Tip: Declare, but do not define, primary templates for traits classes that have a limited and predictable range of compatible types.

Second, the `signed_type` and `unsigned_type` member types are the same for a given integer size, but the `type` and `alt_sign_type` are reversed.

12.1.3 Type Characteristics: Mini-Traits

Traits classes are great for providing a cluster of information about a given type. But sometimes all you want is a single characteristic, for which types—half-heartedly called mini-traits—such as the following examples can be useful.

12.1.4 `is_integral_type`

The standard (C++-03: 3.9.1) defines four *signed integer types*, `signed char`, `short int`, `int`, and `long int`, and four *unsigned integer types*, `unsigned char`, `unsigned short int`, `unsigned int`, and `unsigned long int`. Together with `bool`, `char`, and `wchar_t`, these are the *integral types*. We can make a traits class for these as shown in Listing 12.6.

Listing 12.6 Primary Template of `is_integral_type`

```
// In namespace stlsoft
template <typename T>
struct is_integral_type
{
  enum { value = 0 };
  typedef no_type   type;
};
```

In the general case, specializations of `is_integral_type` will indicate that the specializing type is not an integral type, by having a member constant `value` with a value of *0* and a member type `type` of type `no_type`. (`yes_type` and `no_type` are two distinct **STLSoft** types, used as Booleans in metaprogramming.) To make the traits class useful, we need to specialize it for the integral types, as shown in Listing 12.7.

Listing 12.7 Example Specializations of `is_integral_type`

```
template <>
struct is_integral_type<signed char>
{
  enum { value = 1 };
  typedef yes_type  type;
};
template <>
struct is_integral_type<unsigned char>
{
  enum { value = 1 };
  typedef yes_type  type;
};
```

The specializations define `value` to be *1* and `type` to be `yes_type`. Having both a value and a type makes the mini-traits types easier to use since you do not have to synthesize one characteristic from the other in situ. Note that the nonzero value is always set to *1* but always tested as non-*0*. This makes the effect of an erroneously set nonzero value that is not *1* benign.

Tip: Always represent Boolean metaprogramming constants as *0* and *1*. Always test Boolean metaprogramming constants for *0* and non-*0*.

This kind of thing rapidly gets boring (and error-prone), so, much as I loathe macros, I use one for defining specializations for mini-traits classes:

```
#define STLSOFT_GEN_TRAIT_SPECIALISATION_WITH_TYPE(TR, T, V, MT)  \
  template <> struct TR<T>                                        \
  { enum { value = V }; typedef MT type; };
```

This macro is used as follows:

```
// In stlsoft/meta/is_integral_type.hpp
STLSOFT_GEN_TRAIT_SPECIALISATION_WITH_TYPE( is_integral_type
                                          , bool, 1, yes_type)
STLSOFT_GEN_TRAIT_SPECIALISATION_WITH_TYPE( is_integral_type
                                          , char, 1, yes_type)
STLSOFT_GEN_TRAIT_SPECIALISATION_WITH_TYPE( is_integral_type
                                          , wchar_t, 1, yes_type)
. . . // And signed+unsigned char, short, int, long
```

Most compilers provide extended integer types, for 64-bits among other things, so such types are also conditionally defined in the **STLSoft** headers, as in the following:

```
#ifdef STLSOFT_CF_64BIT_INT_SUPPORT
STLSOFT_GEN_TRAIT_SPECIALISATION_WITH_TYPE( is_integral_type
                                          , sint64_t, 1, yes_type)
STLSOFT_GEN_TRAIT_SPECIALISATION_WITH_TYPE( is_integral_type
                                          , uint64_t, 1, yes_type)
#endif /* STLSOFT_CF_64BIT_INT_SUPPORT */
```

With all appropriate specializations defined, we can use this class to help us discriminate between types at compile time for template metaprogramming. In particular, it's useful in combination with *static assertions* (Section 8.2), in defining constraints (Chapter 8), as follows:

```
template <typename T>
class UseIntegersOnly
{
  . . .
  ~UseIntegersOnly()
  {
    STATIC_ASSERT(0 != is_integral_type<T>::value);
  }
};
```

I prefer to use the destructor because there can be only one, and it's likely to be instantiated in most use cases.

12.1.5 is_signed_type

Another mini-traits class is is_signed_type. It does the same thing as is_integral_type, but only for the signed integer types (C++-03: 3.9.1) and those other signed integral types provided as an extension by compilers, along with the floating-point types float, double, and long double. It is used in constraints in Sections 23.3.4 and 32.6.1.

12.1.6 is_fundamental_type

Mini-traits can be combined, as in is_fundamental_type, which combines the notions of integral type, floating-point type, bool, and void, as shown in Listing 12.8.

Listing 12.8 Definition of the is_fundamental_type **Composite Traits Class**

```
// In namespace stlsoft
template <typename T>
struct is_fundamental_type
{
  enum
  {
    value =   is_integral_type<T>::value
```

```
          ||  is_floating_point_type<T>::value
          ||  is_bool_type<T>::value
          ||  is_void_type<T>::value
  };
  typedef typename select_first_type_if<yes_type
                                       , no_type
                                       , value
                                       >::type       type;
};
```

The determination of the member constant `value` is a logical OR of the `value` member value of each of the constituent traits specializations. An arithmetic OR would also work, but using a logical OR ensures that if any of the constituent mini-traits erroneously defines their true value to be a nonzero value other than *1*, `is_fundamental_type`'s `value` will be *1*, which is what's required.

Tip: Use logical AND and OR when calculating Boolean metaprogramming constants as *0* and *1*, to ensure correct results even when evaluating inappropriately defined true constituent values.

The determination of the member type `type` uses the compile-time IF template, `select_first_type_if`, which is described in Chapter 13.

12.1.7 `is_same_type`

This class template, shown in Listing 12.9, evaluates whether two types are the same.

Listing 12.9 Definition of `is_same_type`

```
// In namespace stlsoft
template <typename T1, typename T2>
struct is_same_type
{
  enum { value = 0 };
  typedef no_type   type;
};
template <typename T>
struct is_same_type<T, T>
{
  enum { value = 1 };
  typedef yes_type  type;
};
```

In the general case, the result, in the member constant `value`, is *0*. In the case of the partial specialization, the `value` is *1*. Although this is one of the simplest mini-traits one could imagine, it's exceedingly powerful and is used throughout the **STLSoft** libraries and in several components described in this volume (Chapters 24 and 41).

12.2 Type Generators

A type generator is a class template whose sole purpose is to discriminate on some facet(s) of its specializing types/values and to define a member type accordingly. Among other things, type generators make up for the lack of typedef templates, also known as template aliases, in the language.

12.2.1 `stlsoft::allocator_selector`

A good example of a type generator is the `stlsoft::allocator_selector` template, which is used to select the appropriate default allocator for most of the class templates in the **STLSoft** libraries. It is defined as shown in Listing 12.10.

Listing 12.10 Definition of the `allocator_selector` Generator Template

```
// In namespace stlsoft
template <typename T>
struct allocator_selector
{
#if defined(STLSOFT_ALLOCATOR_SELECTOR_USE_STLSOFT_MALLOC_ALLOCATOR)
  typedef malloc_allocator<T>     allocator_type;
#elif defined(STLSOFT_ALLOCATOR_SELECTOR_USE_STLSOFT_NEW_ALLOCATOR)
  typedef new_allocator<T>        allocator_type;
#elif defined(STLSOFT_ALLOCATOR_SELECTOR_USE_STD_ALLOCATOR)
  typedef std::allocator<T>       allocator_type;
#else /* ? allocator */
# error Error in discrimination . . .
#endif /* allocator */
};
```

`allocator_selector` is used in place of `std::allocator` in the template parameter lists of many components, as shown in the case of `auto_buffer` (Section 16.2) as follows:

```
template< typename T // The value type
        , size_t   N = 256
        , typename A = typename allocator_selector<T>::allocator_type
        >
class auto_buffer;
```

In the majority of cases, the preprocessor discrimination in the definition of this template causes `std::allocator<T>` to be defined as the `allocator_type`. But by using the indirection of `allocator_selector`, a different allocator can be selected as the default for all components to fulfill exotic requirements just by simple definition of a preprocessor symbol.

12.3 True Typedefs

True typedefs were described in detail in Chapter 18 of *Imperfect C++*, so I'll be brief here. As you undoubtedly know, typedefs in C++ (as in C) are merely aliases. They do not define separate types. For example, the following code is an error:

```
typedef int   type_1;
typedef int   type_2;

void fn(type_1 v) {}
void fn(type_2 v) {} // Error: fn(int) already defined!
```

This code does not define overloads of the `fn()` function, one for each of `type_1` and `type_2`. But you can define those overloads by using true typedefs. The `stlsoft::true_typedef` class template uses the fact that all distinct template specializations (except those related using inheritance) define distinct types. We can define unique types as follows:

```
typedef stlsoft::true_typedef<int, void**>    type_1;
typedef stlsoft::true_typedef<int, void*>     type_2;
```

The second type can be *any* type, as long as each permutation of the two specializing types is unique. Now `type_1` and `type_2` are unique types and can be used to overload `fn()`:

```
void fn(type_1 v) {}
void fn(type_2 v) {} // OK!
```

CHAPTER 13

Inferred Interface Adaptation: Compile-Time Adaptation of Interface-Incomplete Types

It is better to stay silent and be thought a fool, than to speak and remove all doubt.

—Abraham Lincoln

But we don't need to know the Latin bit. Why is everyone always going back to Latin? It was ages ago.

—Karl Pilkington

13.1 Introduction

Adaptor class templates are used to convert a class, or a related group of classes, from an existing interface to a new interface. Consider the `std::stack` class template, which is used to adapt sequence containers to an interface that exhibits the stack operations `push()` and `pop()`, as shown in Listing 13.1. This works because all the methods of `std::stack` are implemented in terms of the public interface of the adapted type: the member types `size_type` and `value_type` and the methods `back()` and `push_back()`.

Listing 13.1 Example Function Template and Test Code

```
template <typename S>
void test_stack(S& stack)
{
  stack.push(101);
  stack.push(102);
  stack.push(103);
  assert(3 == stack.size() && 103 == stack.top());
  stack.pop();
  assert(2 == stack.size() && 102 == stack.top());
  stack.pop();
  assert(1 == stack.size() && 101 == stack.top());
  stack.pop();
  assert(0 == stack.size());
}
```

```
std::stack<int, std::deque<int> >   deq;
std::stack<int, std::vector<int> >  vec;
std::stack<int, std::list<int> >    lst;
test_stack(deq);
test_stack(vec);
test_stack(lst);
```

The problem addressed in this chapter is what to do when the adaptor template makes demands that a potential underlying type cannot directly fulfill. Can we increase the spectrum of adaptable types by adding flexibility to the adaptor? I will introduce the technique of *inferred interface adaptation* (IIA), which is comprised of three template metaprogramming techniques: *type detection*, *type fixing*, and *type selection*. As described in Chapter 41, IIA is useful for other things, such as enabling code to work with new and old standard library implementations alike (which is how I discovered IIA some years ago).

13.2 Adapting Interface-Incomplete Types

Consider the class template `sequence_range` shown in Listing 13.2, which implements the ***Iterator*** pattern for STL collections (those that provide STL iterators via `begin()` and `end()` methods) and presents iteration and access via the `advance()` and `current()` methods. (It's actually a chopped-down version of the **RangeLib** component of the same name; we'll meet this and other ranges more fully in Volume 2.)

Listing 13.2 Initial Version of the `sequence_range` Adaptor Class Template

```
// In namespace rangelib
template <typename C>
class sequence_range
{
public: // Member Types
  typedef typename C::reference       reference;
  typedef typename C::const_reference const_reference;
  typedef typename C::iterator        iterator;
public: // Construction
  sequence_range(C& c)
    : m_current(c.begin())
    , m_end(c.end())
  {}
public: // Iterator Pattern Methods
  reference current()
  {
    return *m_current;
  }
  const_reference current() const
  {
    return *m_current;
  }
  bool is_open() const
  {
    return m_current != m_end;
  }
  void advance()
  {
    ++m_current;
  }
private: // Member Variables
  iterator m_current;
  iterator m_end;
};
```

In order to support mutating and nonmutating access to the elements of the underlying collection, the `current()` method is overloaded. The mutating form returns (the non-`const`) `reference`, the nonmutating form returns `const_reference`. This supports three call forms:

```
typedef sequence_range<std::vector<int> >  range_t;
void f1(range_t& r);       // Invokes r.current()
void f2(range_t const& r); // Invokes r.current()

range_t       &r  = . . .
const range_t &cr = . . .
f1(r);  // non-const passed as non-const - OK
f2(r);  // non-const passed as const - OK
f2(cr); // const passed as const - OK
f1(cr); // const passed as non-const - compile error
```

This is `const` methods 101, and failing to facilitate such typical behavior in an adaptor class would be unnecessarily restrictive. As we'll see, though, meeting such straightforward requirements brings along quite a few complications.

13.3 Adapting Immutable Collections

As we'll see throughout Part II, many real-world STL collections do not provide mutating operations. When the underlying collection does not support mutable (non-`const`) references, there's a problem with the `sequence_range` adaptor defined in the previous section. Consider what happens if we use it with the `glob_sequence` class (see Section 17.3), which has a nonmutating interface, as shown in Listing 13.3.

Listing 13.3 `glob_sequence` **Member Types**

```
class glob_sequence
{
public: // Member Types
  typedef char                                   char_type;
  typedef char_type const*                       value_type;
  typedef value_type const&                      const_reference;
  typedef value_type const*                      const_pointer;
  typedef glob_sequence                            class_type;
  typedef const_pointer                            const_iterator;
  typedef std::reverse_iterator<const_iterator>
                                                   const_reverse_iterator;
  . . .
```

If we try to adapt this class with `sequence_range`, we'll get compile errors in the definition of the member types of `sequence_range<glob_sequence>`. Specifically, the compiler will inform us that the adapted class has no member types `reference` and `iterator`.

We need the adaptor to infer, at compile time, whether it's being used with a type that does not support mutable operations and, if not, define suitable stand-in types based on the (public) interface of the adapted class. In other words, when adapting `glob_sequence`, it should infer `sequence_range`'s `reference` and `iterator` member types as the **`const_`**`reference` and **`const_`**`iterator` member types of `glob_sequence`. Then the effective definition of `sequence_range` would be as shown in Listing 13.4.

Listing 13.4 Effective Definition of `sequence_range`

```
template <typename C>
class sequence_range
{
  . . .
  const_reference current()
  {
    return *m_current;
  }
  const_reference current() const;
  . . .
private: // Member Variables
  const_iterator m_current;
  const_iterator m_end;
};
```

13.4 Inferred Interface Adaptation

IIA consists of three steps.

1. Infer the interface on the adapted type by using type detection (Section 13.4.2).
2. Fix any incompatibilities by using type fixing (Section 13.4.3).
3. Define the adaptor type's interface in terms of the actual or fixed types of the adapted type by using type selection (Section 13.4.1).

Before we find out how IIA works, let's take a look at it in action. Listing 13.5 shows how IIA can be used to infer the appropriate type of the `iterator` member type. (At the urging of a couple of my reviewers who are not native speakers of English, I must observe that *putative* means commonly regarded to be; the candidate to be; supposed; or assumed to be.)

Listing 13.5 First Definition of `sequence_range` Iterator Members

```
template <typename C>
class sequence_range
{
private: // Member Types
  . . .
  // 1. Type Detection
  enum { C_HAS_MUTABLE_INTERFACE = . . . ???? . . . };
  // 2. Type Fixing
  typedef typename typefixer_iterator<C
                                    , C_HAS_MUTABLE_INTERFACE
                                    >::iterator   putative_iterator;
public:
  typedef typename C::const_iterator             const_iterator;
  // 3. Type Selection
  typedef typename select_first_type_if<putative_iterator
                                    , const_iterator
                                    , C_HAS_MUTABLE_INTERFACE
                                    >::type       iterator;
  . . .
```

The member value, `C_HAS_MUTABLE_INTERFACE`, is a compile-time constant that indicates whether the adapted collection type, `C`, provides a mutable interface: This is type detection. We'll look at its definition in a moment. Next, the type fixer template, `typefixer_reference`, is used to define the `putative_iterator` member type: This is type fixing. Finally, a type selector template, `select_first_type_if`, selects between the `putative_iterator` and `const_iterator` types to define the `iterator` member type: This is type selection.

13.4.1 Type Selection

Let's deal with the easy part first, type selection. This is a well-established idiom in template metaprogramming, consisting of a primary template and a partial specialization. The component in the **STLSoft** libraries that does this is the type selector template `select_first_type_if`, shown in Listing 13.6.

Listing 13.6 Definition of the `select_first_type_if` Type Selector Class Template

```
// In namespace stlsoft
template <typename T1, typename T2, bool CHOOSE_FIRST_TYPE>
struct select_first_type_if
{
  typedef T1  type; // The first type
};
template <typename T1, typename T2>
struct select_first_type_if<T1, T2, false>
{
  typedef T2  type; // The second type
};
```

When the third Boolean template parameter is nonzero, the first type is selected. When the third parameter is *`false`*, the second type is selected. Thus `select_first_type_if<int, char, true>::type` is `int`, and `select_first_type_if<int, char, false>::type` is `char`.

13.4.2 Type Detection

The next problem we must solve is arguably the most mind-bending. It involves the use of the *SFINAE* principle to define a template that can detect member types. *SFINAE*, which stands for *Substitution Failure Is Not an Error*, is a mechanism essential to compilers in the identification of function templates. Basically, *SFINAE* states that if the specialization of a function template for a given template argument would result in an error, it is *not* deemed to be an error as long as a good alternative is available. (Thankfully, having a deep and profound understanding of *SFINAE* is not necessary to make good use of it, as ably demonstrated by the tendency of this author to forget the fine details within a few minutes of comprehension—there, it's gone again!—and yet still craft adaptive code that depends on it.) **STLSoft** contains a number of type-detecting components, including the `has_value_type` component (Listing 13.7), used for detecting whether a class defines a `value_type` member type.

Listing 13.7 Definition of the `has_value_type` Member Type Tester Class Template

```
// In namespace stlsoft
typedef struct { char  ar[1]; } one_t; // sizeof(one_t) . . .
typedef struct { one_t ar[2]; } two_t; // . . . != sizeof(two_t)

template <typename T>
one_t has_value_type_function(...);

template <typename T>
two_t
  has_value_type_function(typename T::value_type const volatile*);

template <typename T>
struct has_value_type
{
  enum
  {
    value = sizeof(has_value_type_function<T>(0)) == sizeof(two_t)
  };
};
template<>
struct has_value_type<void>
{
  enum { value = 0 };
};
```

Although this code looks like a horrible indigestible mess, it's actually *reasonably* straightforward once you break it down. For a type `T`, specializing to `has_value_type<T>` involves determining which instantiation of the `has_value_type_function()` template function for `T` best matches an argument of *0*. The second template, involving `typename T::value_type const volatile*`, is more specific than the one taking *any* arguments (indicated in C/C++ by the ellipsis parameter `...`), and it can be matched to *0* (since *0* can be a pointer literal, as well as an integer literal), for any `T` that has a member type `value_type`. This is type detection because `has_value_type<T>::value` will be *1* if `T` has a member type `value_type` and *0* if it does not. If `T` does not have the member type `value_type`, the ellipsis version will be chosen, and *SFINAE* dictates that the specialization does not generate a compile-time error. (Note the specialization of `has_value_type` for `void`. It's not used in this chapter but is required for the application of IIA in Chapter 41.)

We can now see how to determine the value for `C_HAS_MUTABLE_INTERFACE`. We choose a member type that only a mutable collection would have, say, `iterator`, and detect it using an appropriately defined `has_iterator` type detector:

```
enum { C_HAS_MUTABLE_INTERFACE = has_iterator<C>::value };
```

Given the imperfect nature of some standard libraries and/or some STL extensions, it's wise to err on the side of caution and detect several mutable-only member types:

```
enum { C_HAS_MUTABLE_INTERFACE = has_iterator<C>::value &&
                                 has_pointer<C>::value };
```

13.4.3 Type Fixing

We now can detect whether our collection has a mutable interface, and we know how to select a type. All that remains is to fix the types. A naïve attempt at this is shown in Listing 13.8.

Listing 13.8 Incorrect Definition of `sequence_range` Reference Members

```
  enum { C_HAS_MUTABLE_INTERFACE = has_iterator<C>::value &&
                                   has_pointer<C>::value };
  typedef typename select_first_type_if<typename C::reference
                                    , typename C::const_reference
                                    , C_HAS_MUTABLE_INTERFACE
                                    >::type        reference;
  typedef typename C::const_reference              const_reference;
```

The problem here stems from the fact that `select_first_type_if` is specialized with the types `C::reference` and `C::const_reference`. If `C` is a type that does not have a `reference` member type, the specialization of `select_first_type_if`, and therefore of `sequence_range` as a whole, is invalid, and compilation errors ensue. Partial template specialization comes to the rescue again, this time in the form of the `fixer_reference` primary class template and its partial specialization (Listing 13.9).

Listing 13.9 Definition of the `fixer_reference` Type Fixer Class Template

```
// In namespace stlsoft
template <typename T, bool HAS_MEMBER>
struct fixer_reference
{
  typedef typename T::reference   reference;
};
template <typename T>
struct fixer_reference<T, false>
{
  typedef void                    reference;
};
```

The first parameter, `T`, is the collection type. The second parameter is used to indicate whether the collection has a `reference` member type. The primary class template defines the member type `reference` from the `reference` member type of the collection. In the partial specialization, where the second template parameter is *`false`* to indicate that `T` does not have a `reference` member type, the member type `reference` is typedef'd from `void`. This is type fixing. With it, we can now express ourselves in terms of the member type `reference` of collection types that may not define this type. Now the following template expression:

```
typedef typename typefixer_reference< C
                                  , C_HAS_MUTABLE_INTERFACE
                                  >::reference  putative_reference;
```

is eminently compilable, irrespective of whether `C_HAS_MUTABLE_INTERFACE` is *`true`* or *`false`*. If `C_HAS_MUTABLE_INTERFACE` is *`true`*, then `typefixer_reference<C, C_HAS_MUTABLE_INTERFACE>::reference` evaluates to be `C::reference`. Thus, the following:

```
select_first_type_if< putative_reference
                    , const_reference
                    , C_HAS_MUTABLE_INTERFACE
                    >::type
```

evaluates to:

```
select_first_type_if< C::reference
                    , C::const_reference
                    , true
                    >::type
```

which evaluates to `C::reference`. Alternatively, if `C_HAS_MUTABLE_INTERFACE` is *`false`*, then `typefixer_reference<C, C_HAS_MUTABLE_INTERFACE>::reference` evaluates to be `void`. Thus, the following:

```
select_first_type_if< putative_reference
                    , const_reference
                    , C_HAS_MUTABLE_INTERFACE
                    >::type
```

evaluates to:

```
select_first_type_if< void
                    , C::const_reference
                    , false
                    >::type
```

which evaluates to `C::const_reference`.

At no point do we have a type that doesn't exist—in its stead is `void`—and so the code is acceptable to the compiler. Of course, if the adapted type is missing `const_iterator` or `const_reference`, the compiler's going to still be complaining. But expecting adaptors to have intelligence to cope in that case is bordering on the metaphysical; we can reasonably require users of the `sequence_range` adaptor to use it with types that, at least, provide `const_iterator` and `const_reference` member types, along with `begin()` and `end()` methods.

13.5 Applying IIA to the Range

Plugging all this back into the `sequence_range` class template, we have the definition shown in Listing 13.10.

Listing 13.10 Final Definition of `sequence_range` Iterator and Reference Members

```
private: // Member Types
  enum { C_HAS_MUTABLE_INTERFACE = has_iterator<C>::value &&
                                   has_pointer<C>::value };
  typedef typename typefixer_reference< C
                                      , C_HAS_MUTABLE_INTERFACE
                                      >::reference  putative_reference;
  typedef typename typefixer_iterator< C
                                      , C_HAS_MUTABLE_INTERFACE
                                      >::iterator   putative_iterator;
public:
  typedef typename C::const_reference              const_reference;
  typedef typename select_first_type_if<putative_reference
                                      , const_reference
                                      , C_HAS_MUTABLE_INTERFACE
                                      >::type       reference;
  typedef typename C::const_iterator               const_iterator;
  typedef typename select_first_type_if<putative_iterator
                                      , const_iterator
                                      , C_HAS_MUTABLE_INTERFACE
                                      >::type       iterator;
  . . .
  reference current()
  {
    return *m_current; // Now works for mutable and immutable colls
  }
  const_reference current() const;
  . . .
```

Now the adaptor works for types that support mutable and immutable operations *and* for types that support only immutable operations. There are some further complications in the actual definition of `sequence_range` in **STLSoft**'s **RangeLib** to handle parameterization of the adaptor with `const` collection types, but they are also addressed by use of IIA. Feel free to check out the sequence adaptors in **RangeLib** and see for yourself.

We'll see more of this technique when we look at the `adapted_iterator_traits` traits class (Chapter 41), a reusable metaprogramming component that affords a high degree of flexibility to authors of iterator adaptors.

CHAPTER 14

Henney's Hypothesis, or When Templates Attack!

Inconsistency imposes mental friction into a developer's work.

—Scott Meyers

Writing good template libraries in C++ is a delicate balance between exercising the considerable power afforded by template programming and making overly complex and undiscoverable interfaces that confound users. Kevlin Henney has suggested a relationship that captures the essence of this balance, which I call *Henney's Hypothesis*.

Henney's Hypothesis: For each additional [required] template parameter, the potential number of users is halved.

The "[required]" part is my own humble addition to the hypothesis. I think this is an important modification since it's the number of template parameters a user is required to consider in order to use a template component that is the confounding load. For example, when using `std::map`, you do not usually recoil in fright at the notion of four template parameters: key type (`K`), mapped type (`MT`), key comparison predicate, and allocator. The latter two default to `std::less<K>` and `std::allocator<MT>`, which serve the vast majority of cases. The same can be said about function templates. Consider the *N*-parameter versions of the `dl_call()` function (Section 16.6) provided by the **STLSoft** subprojects **UNIXSTL** and **WinSTL**. Listing 14.1 shows a snapshot of the function declarations of the 2- and 32-parameter overloads.

Listing 14.1 Heterogeneous Template Parameters in Generic Function Templates

```
template< typename R, typename L, typename FD
        , typename A0, typename A1
        >
R dl_call(L const& library, FD const& fd
        , A0 a0, A1 a1);

template< typename R, typename L, typename FD
        , typename A0, . . ., typename A30, typename A31
        >
R dl_call(L const& library, FD const& fd
        , A0 a0, . . . , A30 a30, A31 a31);
```

At first blush, these seem like extreme violations of the hypothesis. However, the compiler will infer the types of the library argument, the function descriptor argument, and all the "actual" arguments (`a0 ... a(N-1)`). All that is required of the user is to specify the return type, as in the following lines:

```
CHAR  name[200];
DWORD cch = dl_call<DWORD>( "KERNEL32", "S:GetSystemDirectoryA"
                          , &name[0], STLSOFT_NUM_ELEMENTS(name));
```

In Part II we're going to see some components that violate *Henney's Hypothesis*, in particular the `string_tokeniser` (Section 27.6). In its default use cases, it is eminently discoverable, such as when used to tokenize a string based on a single character delimiter:

```
stlsoft::string_tokeniser<std::string, char> tokens("ab|cde||", '|');
```

This is about as discoverable as you can get, and the code is highly transparent. Conversely, in other reasonable use cases, it is a wild woman's knitting of undiscoverability, such as with the specialization required to tokenize a string based on a character delimiter set shown in Listing 14.2. (You'll be pleased to know there are nicer ways to get character set delimited tokenization; see Sections 27.8.1 and 27.8.2.)

Listing 14.2 Example Violation of *Henney's Hypothesis*

```
template <typename S>
struct charset_comparator; // A comparator (see Section 27.7.5)

stlsoft::string_tokeniser<std::string
                        , std::string
                        , stlsoft::skip_blank_tokens<true>
                        , std::string
                        , stlsoft::string_tokeniser_type_traits<std::string
                                                               , std::string
                                                               >
                        , charset_comparator<std::string>
                        >     tokens("ab\tcde \n", " \t\r\n");
```

As it crops up throughout this book, I'll be pointing out instances where *Henney's Hypothesis* is transgressed, and we'll look at ways to obviate and/or ameliorate its implications. And in Volume 2 I'll discuss advanced metaprogramming techniques for taming overweened template parameter lists.

I hope you'll take note of Kevlin's wisdom in your own work and consider its effects on your users and therefore on the success of your libraries. My own rubric is that if I need to go to the documentation to look up the meaning of more than one template parameter, I need to refactor or provide an alternate interface (see Section 27.8).

CHAPTER 15

The Independent Autonomies of `equal()` Friends

True friendship is never serene.

—Marquise de Sévigné

In *The Ethics*, Aristotle writes of friendships, both between equal friends and between unequal friends, and discusses the mutual responsibilities and obligations necessary to maintain healthy relationships. In this short chapter, I intend to show how *avoiding* friendship simplifies implementations and avoids unnecessary breaking of encapsulation.

15.1 Beware Nonmember Friend Function Abuse

Scott Meyers' principle of the use of nonmember functions over member functions as a tactic for *increasing* encapsulation is both worthy and widely employed. I follow this whenever it's appropriate. There's a common and totally unnecessary abuse of this principle, largely in the provision of comparison operators. Consider the possible definition of the (in)equality operators for the **UNIXSTL** `basic_path` class shown in Listing 15.1.

Listing 15.1 `basic_path` **with Operator Methods**

```
// In namespace unixstl
template< typename C // Character type
        , typename T = filesystem_traits<C>
        , typename A = std::allocator<C>
        >
class basic_path
{
public: // Member Types
  typedef basic_path<C, T, A>   class_type;
  . . .
public: // Comparison
  bool operator ==(class_type const& rhs) const;
  bool operator ==(C const* rhs) const;
  . . .
```

That's fine as far as it goes, but it means that any comparison involving a `basic_path` instance must have a `basic_path` instance on the left-hand side of the operator:

```
unixstl::basic_path<char>  p1;
unixstl::basic_path<char>  p2;

p1 == p2;                  // Well-defined
p1 == "some-file-name";    // Well-defined
"other-file-name" == p2;   // Compile error
```

In order to accommodate the latter syntactic form, the operator may be defined as a nonmember function. It's common to see this using the `friend` operator, as shown in Listing 15.2.

Listing 15.2 `basic_path` **with Friendship to Operator-Free Functions**

```
template <. . .>
class basic_path
{
  . . .
public: // Comparison
  friend bool operator ==(class_type const& lhs
                        , class_type const& rhs) const;
  friend bool operator ==(class_type const& lhs
                        , C const*          rhs) const;
  friend bool operator ==(C const*          lhs
                        , class_type const& rhs) const;
  . . .
```

Now there are all kinds of strange and subtle rules about the relationships among classes, their nonmember friend functions, C++ namespaces, the linker, and so on. Some compilers require that you define the function within the class body. Some require a previous declaration. I cannot elucidate them all for you here because I don't know them! This is quite deliberate. Each time I've tried to digest these rules and then apply them with a few separate compilers, I've had a bad day. (A sample program on the CD demonstrates the messiness involved.)

It's extremely easy to follow Scott's advice and yet still have classes with succinct definitions, tight encapsulation, and discoverable interfaces. Instead of defining such methods, I define a public nonoperator member function—either `equal()` or `compare()`—and then implement the nonmember comparison functions in terms of the member function. For the `basic_path` class template, the definition of operators == and `!=` are implemented as shown in Listing 15.3.

Listing 15.3 `basic_path` **with the** `equal()` **Method and Operator-Free Functions**

```
template <. . .>
class basic_path
{
  . . .
public: // Comparison
```

```
  bool equal(class_type const& rhs) const;
  bool equal(C const* rhs) const;
  . . .
};

template<typename C, typename T, typename A>
bool operator ==( basic_path<C, T, A> const&  lhs
                , basic_path<C, T, A> const&  rhs)
{
  return lhs.equal(lhs);
}
template<typename C, typename T, typename A>
bool operator ==( basic_path<C, T, A> const&  lhs
                , C const*                    rhs)
{
  return lhs.equal(rhs);
}
template<typename C, typename T, typename A>
bool operator ==( C const*                    lhs
                , basic_path<C, T, A> const&  rhs)
{
  return rhs.equal(lhs);
}
. . . // Same (but !) for operator !=() x 3
```

Similar implementations can be done for any classes that require <, <=, >, or >= comparison, via a `compare()` or similar method. To be sure, this practice violates the letter of Scott's principle by adding a new member function or functions, in this case, `equal()`. But it results in an overall implementation that is highly transparent, removes the need for friendship, and avoids the various portability headaches attendant with the complex, and not always well-supported, `friend` function definition rules. Using this technique enables me to keep the use of the `friend` keyword throughout all of the **STLSoft** libraries to double figures. All the examples drawn from my libraries throughout the book will use this technique.

Tip: Avoid abuse of friendship with nonmember comparison operator functions by defining a nonoperator nonmutating unary public member function, in terms of which *nonfriend* nonmember binary comparison operators can be derived.

15.2 Collections and Their Iterators

The one common case where I believe friendship is a useful and helpful tool is in the definition of collections and their associated iterator classes. Iterators often hold onto resources that should not be made visible to client code. The corresponding members are naturally defined `private`. The collections pass such resources into the iterator instances via a conversion constructor. But if the conversion constructor were publicly accessible, it would be easy for client code to inappropriately initialize an iterator. As a consequence, the conversion constructor is made private, and the collection class, which is the only entity with any business calling it, a friend of the class. Examples of this include the `readdir_sequence` (Chapter 19), `Fibonacci_sequence` (Chapter 23), and `environment_map` (Chapter 25) collections.

Other cases where I've used friendship already involve tight coupling, so the use of `friend` itself does not add further coupling. Examples include output iterators using the ***Dereference Proxy*** pattern (Chapter 35), the `CArray_adaptor_base` and its defined class and instance adaptor classes (Chapter 24), and the allocator adaptor classes that I'll discuss in Volume 2.

CHAPTER 16

Essential Components

The will to win is nothing without the will to prepare.

—Juma Ikanga

Programmers are people who are willing to work very hard to solve a problem once so they never have to deal with it again.

—Sean Kelly

16.1 Introduction

This chapter describes five components from the **STLSoft** libraries that are used in the implementations of many of the STL extensions described throughout Parts II and III. They include one smart pointer that applies RAII to arbitrary types (`stlsoft::scoped_handle`), two components that deal with memory (`stlsoft::auto_buffer` and `unixstl/winstl::file_path_buffer`), a traits class for discriminating and abstracting file system differences between different operating systems (`unixstl/winstl::filesystem_traits`), and a suite of functions for safely and succinctly invoking functions from dynamic libraries (`unixstl/winstl::dl_call`).

16.2 `auto_buffer`

Stack memory allocation is exceedingly fast but has to be of a fixed size known at compile time. Heap memory is much slower but can be of any size, as determined at runtime. `auto_buffer` is an **STLSoft** class template that provides optimized memory acquisition for local-scope allocation, merging the speed advantage of stack memory with the dynamic-size flexibility of heap memory. (Although C99's variable-length arrays and the nonstandard `alloca()` attempt to achieve this compromise, neither achieve it sufficiently for most C++ purposes; see Chapter 32 of *Imperfect C++* for the full discussion.)

`auto_buffer` uses the simple artifice of maintaining a fixed-size internal buffer, from which allocation is made, if possible. If the requested size exceeds the internal buffer, the memory is acquired from the heap, via the parameterizing allocator. Consider the following code:

```
size_t                               n = . . .
stlsoft::auto_buffer<wchar_t, 64>    buff(n);

std::fill_n(&buff[0], L'\0', buff.size());
```

If n is 64 or less, no allocation will take place and the expression &buff[0] will evaluate to the address of the first element in the 64-element wchar_t array member—known as the *internal* buffer—of buff. Being a member variable of buff, the memory of that array is in the same stack frame as buff (and n). If n is greater than 64, the internal buffer is not used, and the constructor of buff will attempt to acquire an *external* buffer from its allocator, which will either fulfill the request or throw std::bad_alloc.

The purpose of auto_buffer is twofold.

1. It provides a simple abstraction over dynamically sized raw memory buffers that are used in the implementation of STL extensions.
2. It provides significant speed optimization in the common case where most requirements can be filled within a small, known size. It is especially useful when working with C APIs and, as you'll see throughout the book, is a very widely used component.

16.2.1 It's Not a Container!

auto_buffer provides some methods that are to be found in STL containers, but it is *not* a standard-conformant container. It does not do any initialization or destruction of its contents. Although auto_buffer allows resizing, contents are moved via bitwise copying (by memcpy()) rather than in the manner of in-place construction and destruction required by the standard of *containers*. Although auto_buffer provides swap(), it proscribes both copy construction and copy assignment.

Furthermore, only plain old data (POD) types may be stored in auto_buffers. (Any type that has a meaningful representation in C is POD.) This is enforced by a constraint (Chapter 8) in the constructor:

```
template <. . .>
auto_buffer::auto_buffer(. . .)
{
  stlsoft_constraint_must_be_pod(value_type);
  . . .
```

The class interface includes empty(), size(), resize(), swap(), and mutating and nonmutating forms of begin() and end() (and rbegin() and rend()), but these are provided for the convenience of users implementing classes in terms of it, rather than as any kind of intimation that it is, or should be used like, an *STL container*. As shown in Chapter 22, this facilitates very simple and transparent implementations of collection types in terms of auto_buffer.

16.2.2 Class Interface

Listing 16.1 shows the interface of auto_buffer.

Listing 16.1 Definition of the `auto_buffer` Class Template

```
template< typename T // The value type
        , size_t   N = 256
        , typename A = typename allocator_selector<T>::allocator_type
        >
class auto_buffer
  : protected A
{
public: // Member Types
  . . . // Various common types: value_type, pointer, etc.
public: // Construction
  explicit auto_buffer(size_type dim);
  ~auto_buffer() throw();
public: // Operations
  void                      resize(size_type newDim);
  void                      swap(class_type& rhs);
public: // Size
  bool                      empty() const;
  size_type                 size() const;
  static size_type          internal_size();
public: // Element Access
  reference                 operator [](size_type index);
  const_reference           operator [](size_type index) const;
  pointer                   data();
  const_pointer             data() const;
public: // Iteration
  iterator                  begin();
  iterator                  end();
  const_iterator            begin() const;
  const_iterator            end()const;
  reverse_iterator          rbegin();
  reverse_iterator          rend();
  const_reverse_iterator    rbegin() const;
  const_reverse_iterator    rend()const;
private: // Member Variables
  pointer     m_buffer;
  size_type   m_cItems;
  value_type  m_internal[N];
private: // Not to be implemented
  auto_buffer(class_type const&);
  class_type& operator =(class_type const&);
};
```

Only two methods affect the size and hence the internal arrangement: `resize()` and `swap()`. The static method `internal_size()` returns the size of the internal buffer for a given specialization of the class template. All the other methods have the semantics you would expect from a container (even though `auto_buffer` isn't one). Both subscript operator overloads have precondition enforcements to check that the given index is valid.

Note that it uses the `allocator_selector` generator template, discussed in Section 12.2.1, in order to select the appropriate allocator for the given compiler, library, or context. For simplicity, you may just think of this template parameter as `std::allocator<T>`.

16.2.3 Copying

`auto_buffer` does not define a copy constructor, and for very good reason: Providing copy construction would allow the compiler to generate implicit copy constructors for classes implemented in terms of it. Because it manages uninitialized, or rather externally initialized, memory for POD types, this would lead to bugs in cases where it's not reasonable to simply copy the elements, such as where the elements are pointers, as we'll see with `unixstl::glob_sequence` (Chapter 17). By proscribing the `auto_buffer` copy constructor, authors of types built from it are forced to consider the implications, as shown in the definition of the copy constructor of `winstl::pid_sequence` (Section 22.2).

16.2.4 Nice Allocators Go Last

The definition of `auto_buffer` in all versions of **STLSoft** up to version 1.9 had the following template parameter list:

```
template< typename T
        , typename A = typename allocator_selector<T>::allocator_type
        , size_t   N = 256
        >
class auto_buffer;
```

Without beating about the bush, this was a straight-out mistake. Just about every time I've used an `auto_buffer` I've cursed my mistake, as the dimension is almost always explicitly specified, and the allocator only rarely.

Tip: Absent another overriding requirement, always favor making the allocator argument the last in a class template's template parameter list.

Making the change from version 1.8 to version 1.9 involved a lot of effort, particularly in all the dependent code, especially in other libraries that depend on **STLSoft**. One thing I always do, which aids in this and other cases, is put version information, in the form of preprocessor discriminatable symbols, into the individual source files. If you have a look through any of my libraries, you'll see such information placed into headers, as follows:

```
// File: stlsoft/memory/auto_buffer.hpp

#define STLSOFT_VER_STLSOFT_MEMORY_HPP_AUTO_BUFFER_MAJOR      5
#define STLSOFT_VER_STLSOFT_MEMORY_HPP_AUTO_BUFFER_MINOR      0
#define STLSOFT_VER_STLSOFT_MEMORY_HPP_AUTO_BUFFER_REVISION   5
#define STLSOFT_VER_STLSOFT_MEMORY_HPP_AUTO_BUFFER_EDIT       146
```

Other libraries that use the `auto_buffer` component have been made backward-compatible by discriminating on this version information.

Tip: Place preprocessor discriminatable version information in library header files to enable users to maintain backward-compatible implementations.

16.2.5 `swap()`

As it is a low-level component, `auto_buffer` defines a `swap()` method to support efficiency and exception safety. However, it is important to recognize that it does not always support constant-time efficiency. In the case where one or both instances involved in the swapping are using their local buffers, the contents of the buffers must be exchanged by copying. This is not as bad as it sounds, however, because such `memcpy()`-ing is highly optimized bread and butter to modern processors, and more significantly, the internal buffer is (or should be!) of modest size by design (e.g., see Section 16.4).

16.2.6 Performance

Since one of the two raisons d'être of `auto_buffer` is efficiency, it would be remiss—or uncharacteristically modest, at least—for me not to mention just how efficient it can be. Performance tests have shown that where the required allocation can be fulfilled from the internal buffer, `auto_buffer` averages around 3% (and as little as 1% for some compilers) of the cost of `malloc()/free()`. In cases where the allocation must be fulfilled by the parameterizing allocator, `auto_buffer` averages 104% (and as little as 101% for some compilers) of the cost of `malloc()/free()`. Thus, when the buffer size is chosen appropriately, there are significant performance benefits to be had.

16.3 `filesystem_traits`

Several **STLSoft** subprojects define traits classes to help abstract away differences between operating systems and operating system APIs, and between variants of API functions for different character-encoding schemes. Both **UNIXSTL** and **WinSTL** define a traits class template `filesystem_traits`, which provides abstraction of, among other things, general string handling, file system entry name manipulation, file system state testing, file system control operations, and more.

16.3.1 Member Types

A basic step in defining traits is to establish the member types. The `filesystem_traits` traits templates include the types shown in Listing 16.2.

Listing 16.2 Member Types for `filesystem_traits`

```
// In namespace unixstl / winstl
template <typename C>
struct filesystem_traits
{
```

```
public: // Member Types
  typedef C                      char_type;
  typedef size_t                 size_type;
  typedef ptrdiff_t              difference_type;
  typedef ????                   stat_data_type;
  typedef ????                   fstat_data_type;
  typedef ????                   file_handle_type;
  typedef ????                   module_type;
  typedef ????                   error_type;
  typedef filesystem_traits<C>   class_type;
  . . .
```

The types identified as ***????*** depend on the operating system (and character encoding), as shown in Table 16.1.

Table 16.1 Member Types Between Systems and Character Encodings

Member Type	**UNIX**	**Windows**	
		ANSI/Multibyte	**Unicode**
`stat_data_type`	`struct stat`	`WIN32_FIND_DATAA`	`WIN32_FIND_DATAW`
`fstat_data_type`	`struct stat`	`BY_HANDLE_FILE_INFORMATION`	
`file_handle_type`	`int`	`HANDLE`	
`module_type`	`void*`	`HINSTANCE`	
`error_type`	`int`	`DWORD`	

Here the parameterizing character type (`C`) is represented as a member type by `char_type`, rather than `value_type`, since there's no meaningful value type for `filesystem_traits`.

16.3.2 General String Handling

The first set of functions is for general string handling and merely wraps those of the C standard library, as indicated in Table 16.2. I won't show them in code form since their signatures correspond directly with the ones they wrap.

Table 16.2 Standard and Operating System String Functions Used in Traits

filesystem_traits<C> Method	Equivalent C Standard Function	
	char Specialization	wchar_t Specialization
str_copy	strcpy	wcscpy
str_n_copy	strncpy	wcsncpy
str_cat	strcat	wcscat
str_n_cat	strncat	wcsncat
str_compare	strcmp	wcscmp
str_compare_no_case (**WinSTL** only)	lstrcmpiA	lstrcmpiW
str_len	strlen	wcslen
str_chr	strchr	wcschr
str_rchr	strrchr	wcsrchr
str_str	strstr	wcsstr

winstl::filesystem_traits provides str_compare_no_case() because Windows file system names are case-insensitive.

You might be wondering at the ugly, verbose names. Well, the C standard stipulates (C-99: 7.26.10, 7.26.11) that any function names that begin with str, mem, or wcs and a lowercase letter may be added to the *<stdlib.h>* and *<string.h>* header files at a future time; they are reserved, in other words. It further stipulates (C-99: 7.1.3) that all macro names and identifiers in any part of the standard are reserved.

Tip: When writing libraries, take care to familiarize yourself with the standards' restrictions on valid symbols, and take care to avoid using symbols from the standard libraries as names in your own libraries.

It's not always possible to avoid all symbols that appear in the standard, but where the standard makes specific mention of names to avoid, it would be foolish to ignore it.

16.3.3 File System Name Handling

Different operating systems have both different file system conventions and different APIs with which to manipulate them. Abstracting these completely is probably not possible, and getting close to 100% is quite a challenge, but there is still considerable scope for useful abstraction. The next set of methods attempts to abstract away most of these differences, and these methods are widely used throughout the **UNIXSTL** and **WinSTL** subprojects.

```
public: // File System Names
  static char_type        path_separator();
  static char_type        path_name_separator();
  static size_type        path_max();
  static char_type const* pattern_all();
```

These four methods return operating-system-specific names for common concepts in file system path manipulation. `path_separator()` returns the symbol used to delimit multiple paths: `':'` on UNIX and `';'` on Windows. `path_name_separator()` returns the symbol used to delimit path name components, that is, files, directories, and volumes: `'/'` on UNIX and `'\\'` on Windows. `path_max()` returns the maximum path length for the host system. `pattern_all()` returns the wildcard pattern for "everything" suitable for the host system's native search facilities: on UNIX, `'*'`, as recognized by the shell and **glob** API (Chapter 17); on Windows, `"*.*"`, as recognized by the **FindFirstFile**/**FindNextFile** API (Chapter 20).

One of the hassles of manipulating file system paths as C-style strings is determining the presence or absence of a trailing path name separator and adding or removing it in order to ensure correctly formed composite paths. Three functions are provided to handle this. `has_dir_end()` tests whether such a separator exists. `ensure_dir_end()` adds one if it doesn't exist. `remove_dir_end()` removes it if it exists. All take a nul-terminated character string. `ensure_dir_end()` will write at most one additional character, and it is the caller's responsibility to ensure that there is sufficient space to do so.

```
static bool            has_dir_end(char_type const* dir);
static char_type*      ensure_dir_end(char_type* dir);
static char_type*      remove_dir_end(char_type* dir);
```

The functions in the next set test aspects of path names.

```
static bool            is_dots(char_type const* dir);
static bool            is_path_name_separator(char_type ch);
static bool            is_path_rooted(char_type const* path);
static bool            is_path_absolute(char_type const* path);
static bool            is_path_UNC(char_type const* path);
```

`is_dots()` tests whether the given string is one of the dots directories, either `"."` or `".."`. (We'll see this used in Chapters 17 and 19.) `is_path_name_separator()` tests whether a character is a path name separator. You might wonder why this is offered when the user could just test against `path_name_separator()`. Well, the `/` character is also acceptable, in many cases, as a path name separator on Windows. Using `is_path_name_separator()` helps to keep a bit less of the file system abstraction from leaking out.

On UNIX, every name in the file system is under one root, `/`. However, on Windows, there are three different kinds of nonrelative, or rooted, paths. This is because Windows has drives and supports network connections that use the Universal Naming Convention (UNC) notation. A fully drive-qualified path begins with a drive letter, a colon (`:`), a path name separator, and the remainder of the path, for example, *`H:\Publishing\Books\XSTL`* (or *`H:/Publishing\Books/XSTL`*). A rooted path merely starts with a path name separator, without specifying the drive, for example, *`\Publishing\Books\ImpC++`*. A UNC path begins with a double backslash followed by the server name, a backslash and share name, and then the remaining part of the path, for example, *`\\JeevesServer\PublicShare1/Directory0\Directory01`*. Note that the first three slashes in a UNC path *must* be backslashes. Confusingly, it's also possible to drive qualify a path that's still relative, for example, the path referred to by *`H:abc\def`* actually depends on the current working directory for drive *`H`*.

The different types of rooted paths are serviced by the other three `is_path_??()` methods. `is_path_rooted()` returns *`true`* if the given path corresponds to any of the three rooted types. `is_path_absolute()` returns *`true`* only if the given path is a fully drive-qualified path or a UNC path. `is_path_UNC()` returns *`true`* only if the given path is a UNC path. With these three functions, I've found it possible to work at a surprisingly high level of abstraction (see Section 10.3) when writing code that must work equivalently on both UNIX and Windows. (Naturally, on UNIX `is_path_absolute()` simply calls `is_path_rooted()`, and `is_path_UNC()` always returns *`false`*.)

The three remaining methods in this section are used to translate relative paths to absolute paths:

```
static size_type  get_full_path_name( char_type const* fileName
                                    , size_type cchBuffer, char_type* buffer
                                    , char_type** ppFile);
static size_type  get_full_path_name( char_type const* fileName
                                    , size_type cchBuffer, char_type* buffer);
static size_type  get_short_path_name(char_type const* fileName
                                    , size_type cchBuffer, char_type* buffer);
```

Readers familiar with path name manipulation on the Windows platform will grok the semantics of these functions immediately. The first overload of `get_full_path_name()` takes a file name, writes its absolute form into the buffer defined by `buffer` and `cchBuffer`, and returns a pointer to the rightmost name in `*ppFile` (as long as the buffer length is sufficient to return the full translated path; otherwise, it's set to *`NULL`*). The second overload simply omits the `ppFile` parameter. `get_short_path_name()` returns the Windows short name equivalent, for example, *`H:\AAAAAA~2`* for *`H:\aaaaaaaaaaaaaaaaaaaaaaa`*. On UNIX, it just calls `get_full_path_name()`.

Note that these functions do not guarantee to canonicalize paths, that is, they might not remove *`"/./"`* or resolve *`"/../"`*. Nor do they require that the path actually exist, which is why the **UNIXSTL** version is not implemented in terms of the UNIX function `realpath()`.

16.3.4 File System State Operations

Nice as it is to be able to play around with file names, it's much more exciting to play with actual files. The next set of methods provides the means to ask questions about entries on the file system.

```
public: // File System State
  static bool    file_exists(char_type const* path);
  static bool    is_file(char_type const* path);
  static bool    is_directory(char_type const* path);
  static bool    stat(char_type const* path
                          , stat_data_type* stat_data);
  static bool    fstat(file_handle_type fd
                          , fstat_data_type* fstat_data);
```

All of these retrieve information about an actual file system entry. `file_exists()` returns *`true`* if `path` names an existing entry. `is_file()` and `is_directory()` return *`true`* if `path` exists *and* is of the requisite type. (`is_file()` and `is_directory()` use `stat()`, not `lstat()`, on UNIX. To test for a link, use `unixstl::filesystem_traits::lstat()`.) `stat()` retrieves information about `path`, filling in an instance of `stat_data_type` (see Section 16.3.1) if successful. `fstat()` retrieves `fstat_data_type` information about an open file. `file_exists()`, `is_file()`, and `is_directory()` are implemented in terms of `stat()`.

These functions cover most state requirements, but they don't cover all bases adequately. First, it's reasonably common to execute more than one of `file_exists()`, `is_file()`, and `is_directory()` on a particular file, which means multiple system calls. It is more efficient to make a single call to `stat()`, but the `struct stat` and `WIN32_FIND_DATA` structures are completely different in form and in the flags they use to represent file types.

As a consequence, the traits classes define a further four (for UNIX, where `stat_data_type` and `fstat_data_type` are identical) or eight (for Windows, where they are different types) methods, which take a pointer to an information structure and return a simple Boolean. In this way it's possible to make multiple tests on the nature of a file system entry in a portable manner with only a single system call.

```
static bool_type    is_file(stat_data_type const* stat_data);
static bool_type    is_directory(stat_data_type const* stat_data);
static bool_type    is_link(stat_data_type const* stat_data);
static bool_type    is_readonly(stat_data_type const* stat_data);

static bool_type    is_file(fstat_data_type const* stat_data);
static bool_type    is_directory(fstat_data_type const* stat_data);
static bool_type    is_link(fstat_data_type const* stat_data);
static bool_type    is_readonly(fstat_data_type const* stat_data);
```

16.3.5 File System Control Operations

There is another set of operations that interacts with the file system and modifies its contents, or the relationship a given process has with it. The six operations are obvious, and I won't comment on them other than to point out that all but one, like most of the other traits methods, return a Boolean to indicate success.

```
public: // File System Control
  static size_type get_current_directory(size_type cchBuffer
                          , char_type* buffer);
  static bool      set_current_directory(char_type const* dir);
  static bool      create_directory(char_type const* dir);
  static bool      remove_directory(char_type const* dir);
  static bool      unlink_file(char_type const* file);
  static bool      rename_file(char_type const* currentName
                          , char_type const* newName);
```

16.3.6 Return Types and Error Handling

You perhaps have noticed that many of the traits methods have a Boolean return type. Since analogous functions in different operating system APIs have different ways to indicate errors, this is the most portable manner of abstracting success and failure. If users need extended error information, they can use `get_last_error` and `set_last_error()`:

```
static error_type   get_last_error();
static void         set_last_error(error_type er = error_type());
```

(Note that these functions are thread-specific so long as their underlying operating systems allow; practically, this means every operating system you're likely to use the traits classes on.)

16.4 `file_path_buffer`

Some operating systems have fixed maximum lengths for file system paths; others do not. In order to abstract away these differences and to use efficient (small) fixed buffers where appropriate, file path buffer classes are provided by and used in the **STLSoft** libraries. These requirements are ideally suited to something that uses an `auto_buffer` (Section 16.2) and initializes it to the requisite size to be suitable to store any valid path on a given system. Both **UNIXSTL** and **WinSTL** define such a class template, `basic_file_path_buffer`, as shown in Listing 16.3.

Listing 16.3 Declaration of `basic_file_path_buffer`

```
// In namespace unixstl / winstl
template< typename C // The character type
        , typename A = typename allocator_selector<T>::allocator_type
        >
class basic_file_path_buffer
{
public: // Member Types and Constants
  . . . // Typedefs of value_type, allocator_type, class_type, etc.
  enum
  {
    internalBufferSize = . . .
  };
public: // Construction
  basic_file_path_buffer()
    : m_buffer(1 + calc_path_max_())
  {}
  basic_file_path_buffer(class_type const& rhs)
      : m_buffer(rhs.size())
  {
    std::copy(rhs.m_buffer.begin(), rhs.m_buffer.end()
            , &m_buffer[0]);
  }
  class_type& operator =(class_type const& rhs)
  {
    std::copy(rhs.m_buffer.begin(), rhs.m_buffer.end()
            , &m_buffer[0]);
    return *this;
  }
public: // Operations
  void swap(class_type& rhs) throw();
public: // Accessors
  value_type const* c_str() const;
  reference         operator [](size_t index);
  const_reference   operator [](size_t index) const;
  size_type         size() const;
private: // Implementation
  static size_t calc_path_max_();
private: // Member Variables
  stlsoft::auto_buffer<C, internalBufferSize, A>  m_buffer;
};
```

The default constructor initializes the `auto_buffer` member `m_buffer` with the maximum path for the given platform, plus one for a terminating nul character. Since `auto_buffer` throws `std::bad_alloc` if the requested size specified to its constructor is larger than its internal buffer and cannot be provided by its allocator, a constructed instance of a file path buffer is guaranteed to be always large enough for any valid path for the given platform. Note that the default constructor does *not* perform any initialization of the `m_buffer` contents, not even to set the zeroth element to `'\0'`: A file path buffer must be treated just as if it were an appropriately sized raw array of the given character type. (In debug compilations, the buffer contents are initialized with a `memset()` of `'?'` to catch anyone who makes the wrong assumption.)

The copy constructor and copy assignment operator manually copy the contents because `auto_buffer` deliberately does not provide copy semantics (Section 16.2.3). The `swap()` method is directly implemented in terms of `auto_buffer::swap()`, and the three accessor methods are directly implemented in terms of `auto_buffer::data()`.

The only two unknowns in the picture are the default size of the internal buffer of `m_buffer` (shown as `internalBufferSize`) and the behavior of `calc_path_max_()`, both of which depend on the operating system and are elucidated in the following subsections.

16.4.1 basic_??

Naturally, you've noticed that the actual name for the template is `basic_file_path_buffer`. In this I follow the convention of the standard library, such that templates whose primary parameterization is on character type begin with `basic_`, for example, `basic_path`, `basic_findfile_sequence`, `basic_string_view`, and so on.

Tip: Use the `basic_` prefix for class templates whose primary template parameter defines the class's character type.

Note that all such types are known by their non-`basic_` name—when I talk about `findfile_sequence`, I mean `basic_findfile_sequence<>`—and they reside in files following the same convention, for example, *<unixstl/filesystem/filesystem_traits.hpp>*.

For a while, I dallied with the use of _a and _w postfixes for predefined specializations of such types, for example, drop_handle_sequence_a (basic_drophandle_sequence<char>), path_w (basic_path<wchar_t>), and so on. However, these days I tend to follow the standard by using the non-basic_ name to mean the char-specialization, for example, unixstl::path (unixstl::basic_path<char>), and using the prefix w for the wchar_t specialization, for example, inetstl::wfindfile_sequence (inetstl::basic_findfile_sequence<wchar_t>). The exception to this is with the **WinSTL** project, which, like the Windows headers, uses the unadorned name form for the TCHAR specialization and uses a and w decorators for ANSI/multibyte and Unicode specializations, respectively. Though this sounds inconsistent, it actually works quite intuitively when doing Windows programming, and I've never had a problem (or a user complaint) regarding this scheme.

16.4.2 UNIX and *PATH_MAX*

Some UNIX systems that have a fixed maximum path length define the *PATH_MAX* preprocessor symbol to be the maximum number of bytes in a path, not including the nul-terminating character. Other UNIX variants that do not define a fixed limit at compile time instead indicate their limit via the pathconf() function, which is used to elicit various file-system-related runtime limits:

```
long pathconf(char const* path, int name);
```

To elicit the maximum path, the constant *_PC_PATH_MAX* is specified for the second (name) parameter, and the result returned is the maximum relative to the given path. If that or any other limit cannot be elicited, the function returns *-1*. Thus, to determine the maximum path length on the system, the recommended approach is to specify the root directory "/" and add *1* to the result (if non-negative). Hence, the default buffer size and calc_path_max_() are defined as shown in Listing 16.4.

Listing 16.4 Size Calculations for UNIXSTL's basic_file_path_buffer Class Template

```
  . . .
  enum
  {
#ifdef PATH_MAX
    internalBufferSize =  1 + PATH_MAX
#else /* ? PATH_MAX */
    internalBufferSize =  1 + 512
#endif /* PATH_MAX */
  };
  enum
  {
    indeterminateMaxPathGuess = 2048
  };
  . . .
  static size_t    calc_path_max_()
  {
#ifdef PATH_MAX
    return PATH_MAX;
#else /* ? PATH_MAX */
    int pathMax = ::pathconf("/", _PC_PATH_MAX);
    if(pathMax < 0)
    {
      pathMax = indeterminateMaxPathGuess;
    }
    return static_cast<size_t>(pathMax);
#endif /* PATH_MAX */
  }
```

The member constant `indeterminateMaxPathGuess` is, obviously, a guess to be used when `pathconf()` cannot determine a root-relative maximum path length. Thus, it is possible that a **UNIXSTL** file path buffer can be constructed with insufficient space for any valid paths. Therefore, it's customary to always include specification of the size (`size()`) when manipulating file path buffers. Furthermore, the **UNIXSTL** class also offers a `grow()` method, not provided by its **WinSTL** analog, that attempts to double the allocated space each time it is called.

16.4.3 Windows and *MAX_PATH*

The Windows headers define the constant *MAX_PATH* as 260, and most Windows API functions that deal with file system entity names prescribe the provision of buffers of this dimension. Windows 9x family operating systems cannot support any path names longer than this limit. Windows NT family operating systems do support longer path lengths, up to 32,767. To manipulate these longer paths, however, you must use the wide string versions of the API functions—for example, `CreateFileW()`, `CreateDirectoryW()`, and so on—and prefix the path name with `"\\?\"`. When using the ANSI forms of the API functions—for example, `CreateFileA()`, `CreateDirectoryA()`, and so on—you are limited to 260.

Therefore, the buffer capacity must be 32,767 + 4 only for wide string mode compilation on NT family systems. Compilation mode is easily detected by a compile-time check of the size of the character type (via `sizeof(C)`), and the operating system family by a runtime check of the high-bit of the return value of the `GetVersion()` Windows API function, shown in Listing 16.5.

Listing 16.5 Size Calculations for WinSTL's `basic_file_path_buffer` Class Template

```
  . . .
  enum
  {
    internalBufferSize  =   1 + PATH_MAX
  };
  . . .
  static size_t    calc_path_max_()
  {
    if( sizeof(C) == sizeof(CHAR) ||      // ANSI specialization
        (::GetVersion() & 0x80000000))  // Windows 9x
    {
      // Windows 9x, or NT with ANSI char type
      return _MAX_PATH;
    }
    else
    {
      return 4 + 32767;
    }
  }
```

16.4.4 Using the Buffers

Using the buffers is straightforward. As local variables or as member variables, a fully constructed buffer should be treated just like an array of characters, as in the following:

```
unixstl::file_path_buffer buff;
::getcwd(&buff[0]. buff.size());
```

and:

```
winstl::basic_file_path_buffer<WCHAR> buff;
::GetCurrentDirectoryW(buff.size(), &buff[0]);
```

We'll see the use of these buffers throughout the book since they represent a convenient abstraction of the nontrivial path size calculations across different operating systems and also offer a speed optimization, à la `auto_buffer`, in the majority of use cases.

16.5 `scoped_handle`

The last class template I want to talk about is the `scoped_handle` smart pointer class template, which is used to provide guaranteed cleanup, by a caller-specified function, of a resource within a given scope. It may be used to control the lifetime of resources (`FILE*`) allocated by the C **File Stream** API:

```
{
  FILE*                         file = ::fopen("file.ext", "r");
  stlsoft::scoped_handle<FILE*> h2(file, ::fclose);

  throw std::runtime_error("Are we going to leak?");

} // fclose(file) called here as exception processing passes through
```

or the resources (`void*`) allocated by the C **Memory** API:

```
{
  stlsoft::scoped_handle<void*>  h3(::malloc(100), ::free);

  ::memset(h3.get(), 0, 100);

} // free() called here
```

It can handle resources whose "null" value is non-*0* (or non-*NULL*), as in the following:

```
int fh = ::open("filename.ext", O_WRONLY | O_CREAT
                              , S_IREAD | S_IWRITE);

if(-1 != fh)
{
  stlsoft::scoped_handle<int>    h1(fh, ::close, -1);

  . . . // Use fh

} // close(fh) called here
```

It also works with functions exhibiting the most popular calling conventions used on Windows—**cdecl**, **fastcall**, and **stdcall**—as shown here:

```
{
  void* vw = ::MapViewOfFile(. . .);

  stlsoft::scoped_handle<void*> h4(vw, ::UnmapViewOfFile);

} // UnmapViewOfFile(vw), a stdcall func, is called here
```

`UnmapViewOfFile()` is a **stdcall** function. The constructor template overloads of `scoped_handle` handle the difference in calling conventions for us.

`scoped_handle` even has a specialization for a handle type of `void`, in which case it can be used to invoke functions with no parameters, as in the following code for initialization and guaranteed uninitialization of the **WinSock** libraries:

```
WSADATA   wsadata;
if(0 != ::WSAStartup(0x0202, &wsadata))
{
  stlsoft::scoped_handle<void>  h4(::WSACleanup);

  . . . // Use the WinSock API

} // WSACleanup() called here.
```

The implementation of `scoped_handle` is outside the scope of the discussions of this book. But I do want to point out that it does not use virtual functions or macros, nor does it allocate any memory.

And it is extremely efficient, involving nothing more than casting function pointers to degenerate form and storing them, along with the resource handle, as members of the `scoped_handle` instance, for invocation by the destructor. The casting involved is outside the scope of the rules of the C++ language, but *only when dealing with platform-specific calling conventions*, which are themselves outside the scope of the language rules. When dealing with functions without explicit calling conventions, the implementation operates entirely within the rules of the language.

Note that the use of *any* form of RAII, whether a generic component such as `scoped_handle` or a specific class (e.g., `AcmeFileScope`), to enforce the closure of a file has nontrivial implications that are outside the scope of this book. The same can apply to other resources whose resource cleanup function may fail. All `scoped_handle` does is to ensure that the function is called. It does not offer any assistance in handling any failure of that function.

16.6 dl_call()

The **UNIXSTL** and **WinSTL** subprojects define a powerful suite of overloaded functions, named `dl_call()`, enabling the invocation of functions in dynamic libraries using a natural syntax. Both implementations make good use of string access shims (Section 9.3.1) to provide wide type compatibility, and the Windows version also handles the three common calling conventions **cdecl**, **fastcall**, and **stdcall**.

For example, say that we want to dynamically invoke the Windows API function `GetFileSizeEx()`, which is **stdcall** and resides in the *KERNEL32.DLL* dynamic library. Its signature is as follows:

```
BOOL __stdcall GetFileSizeEx(HANDLE hFile, LARGE_INTEGER* pSize);
```

To invoke this dynamically, we could do the following:

```
LARGE_INTEGER size;
HANDLE        h = ::CreateFile(. . .);

if(!winstl::dl_call<BOOL>("KERNEL32", "S:GetFileSizeEx", h, &size))
{
  . . .
```

The first argument to `dl_call()` specifies the dynamic library from which the function is to be loaded. It must either be a string (`char const*`, or any string type for which the **c_str_ptr** string access shim is defined) or a handle to an already loaded dynamic library (`void*` on UNIX, `HINSTANCE` on Windows). The second argument is the identifier of the dynamic function within the given library. It must be a string (`char const*`, or any string type for which the **c_str_ptr** shim is defined) or may be a function descriptor instance (described below).

When the function identifier is a string, it may optionally be preceded by a calling convention specifier, from which it is separated by a colon `':'`. The specifiers are `"C"` (or `"cdecl"`) for **cdecl**, `"F"` (or `"fastcall"`) for **fastcall**, and `"S"` (or `"stdcall"`) for **stdcall**. If no specifier is given, it defaults to **cdecl**. (**cdecl** is the default calling convention used by all C/C++ compilers, unless they're explicitly instructed by a command-line flag to do otherwise.)

Hence we could have invoked `dl_call()` using the alternate form:

```
winstl::dl_call<BOOL>("KERNEL32", "stdcall:GetFileSizeEx", h, &size)
```

but not as follows:

```
winstl::dl_call<BOOL>("KERNEL32", "GetFileSizeEx", h, &size)
```

which would result in a crash due to the confused stack that would result from the calling convention mismatch between the expected **stdcall** and the given (defaulted) **cdecl**.

All subsequent arguments pertain to the dynamic function, as you would have written them were it called in normal fashion. The template magic inside `dl_call()` handles all the rest for you. Both **UNIXSTL** and **WinSTL** versions support between 0 and 32 arguments, which should cover most requirements. If not, a Ruby generator script is also provided so you can change this to your requirements. (Of course, if you're building or using dynamic library functions with more than 32 arguments, you might want to consider putting in a call to the men in white coats.)

Since we know that `GetFileSizeEx()` is **stdcall**, we can avoid the slight cost in parsing the calling convention (and the opportunity for an error to creep into the function identifier string) by using a function descriptor. The creator function template `fn_desc()` is provided for convenience and may be used either in compile-time form:

```
winstl::dl_call<BOOL>("KERNEL32"
            , winstl::fn_desc<STLSOFT_STDCALL_VALUE>("GetFileSizeEx")
            , h, &size)
```

or in runtime form:

```
winstl::dl_call<BOOL>("KERNEL32"
            , winstl::fn_desc(STLSOFT_STDCALL_VALUE, "GetFileSizeEx")
            , h, &size)
```

The former is *slightly* more efficient and suitable when you know, as you usually do, the calling convention you will be using at compile time. The latter allows for the occasional case, where you might be invoking entry points with different calling conventions, for example, if you're handling old and new plug-ins for a given application.

Using `dl_call()` with an already loaded library is equally straightforward:

```
HINSTANCE hinst = ::LoadLibrary("PSAPI.DLL");
winstl::dl_call<BOOL>(hinst, "S:GetFileSizeEx", h, &size);
```

And, of course, no self-respecting library that I'd put my name on is anything but efficient. The internals of the functions are a little hairy since they need to do a lot of work to discriminate between the different types of library and function descriptors. But the vast majority of that is inlined, and what little remains is inconsequential when compared with the business of loading and fixing up entry points, never mind the cost of the actual called functions themselves.

And that's it for the introductory component of our journey. This is the last chapter that doesn't contain oodles of STL lore betwixt my feeble attempts at wit.

PART TWO

Collections

Much, perhaps most, of the effort in STL extension involves the adaptation of different collection APIs to the *STL collection* concept (Section 2.2). Consequently this part, which is all about that activity, is the biggest in the book. One of the chapters is concerned with adapting an actual container (Chapter 24); all the others describe the adaptation of operating system and third-party APIs.

The STL extension issues covered include collection type, iterator category, element reference category, mutability, coherence of the elements accessible via adaptations with respect to their underlying representations, runtime determination of iterator category (Chapter 28), specialization of standard algorithms (Chapter 31), external iterator invalidation (Chapters 26 and 33), and iterator behaviors that don't quite fit in the established categories (Chapters 26 and 28).

The collection functionality is from such diverse areas as file systems (Chapters 17–21); infinite mathematical sequences (Chapter 23); string tokenization (Chapter 27); COM enumerators and COM collections (Chapters 28–30); networking and I/O (Chapter 31); system processes, environment variables, and configuration (Chapters 22 and 25, Section 33.3); GUI controls (Section 33.2); and Z-order management (Chapter 26). The subject areas were chosen to provide as wide a spectrum of the STL extension issues as possible, while keeping the material accessible and relevant; with the exception of the material on the Fibonacci sequence (Chapter 23), all the extensions are practical and in common use in open-source and commercial projects.

There are seventeen chapters and intermezzos in this part. (Two more intermezzos are available only on the CD.)

Chapter 17 describes the adaptation of an elements-en-bloc API (Section 2.2) to an immutable STL collection (Section 2.2.1), `glob_sequence`, whose iterators are contiguous (Section 2.3.6). It demonstrates the advantages in expressiveness, robustness, and performance that are available from the use of STL extensions. The following intermezzo, Chapter 18, discusses the mistakes in the original design of `glob_sequence` and describes mechanisms for increasing the flexibility of its constructor templates that can be generally applied to a wide range of components. Chapter 19 describes the adaptation of an element-at-a-time API (Section 2.2) to an immutable STL collection, `readdir_sequence`, whose iterators are input iterators (Section 1.3.1). It illustrates how adaptations of the lesser iterator category of input iterator tend actually to be more complex since they require shared iterator state. Chapter 20 also describes the adaptation of an element-at-a-time API to an immutable STL collection, but this time in the form of a collection class template, `basic_findfile_sequence`, which facilitates the multiple models of character encoding popular on the Windows platform. The subject of file system adaptation is completed by an intermezzo, Chapter 21. This describes an FTP enumeration API that is syntactically similar to the file system enumeration API adapted in Chapter 20, but whose semantics are such that an entirely different adaptation is called for.

A first look at system API adaptations is offered in Chapter 22. The STL adaptation aspects are relatively straightforward—providing immutable STL collections with contiguous iterators—but providing consistent implementations and behavior in the face of operating system and compiler variation necessitates some extra cunning.

The only notional collection, representing an infinite mathematical sequence, the Fibonacci sequence, is described in Chapter 23. Ostensibly a very simple component, the practical limitations of integral (and floating-point) numerical representation raise several interesting challenges. Possible implementations are provided and their pros and cons discussed. The final solution relies on the use of a simple but powerful template technique that assists the compiler in discriminating between logically distinct types.

Chapter 24 is at the other end of the STL extension adaptation spectrum. It involves the prosaic matter of adapting a nonstandard container to emulate the syntax and semantics of a standard container, in this case, `std::vector`. It illustrates that, due to the significant divergence in the allocation of memory, representation of elements, and handling of errors, the adaptation is a nontrivial undertaking that involves significant compromise and imposes several restrictions on the semantics of the final result. Nonetheless, the chapter demonstrates that, with a little ingenuity and a lot of determination, a useful and largely STL-compliant container can be obtained from a nonstandard one. Included on the CD is a related intermezzo, Beware Nonpointer Contiguous Iterators, which describes some compiler-specific foibles that made the adaptation even more challenging than is portrayed in Chapter 24.

The second of the system API adaptations follows in Chapter 25. This adaptation raises two important challenges. The easy one is how to adequately abstract the differences in the operating system APIs that manage environment variables. This is achieved by the use of traits (Section 12.1). The considerably more complex challenge is how to reliably and robustly arbitrate access to the range of elements held in a process-global variable. The solution to this involves reference-counted iterator/collection shared snapshots. Included on the CD is a related intermezzo, Handling ADL Gremlins, which describes a technique for cajoling some errant compilers into properly locating nonmember operator functions of *non*-template classes.

Chapter 26 describes the challenges involved in adapting a collection whose arrangement and/or contents may change asynchronously. The main issue is the imperfect match of the collection's iterators to any established iterator category, a problem with a surprisingly self-involved solution.

Violation of *Henney's Hypothesis* (Chapter 14), the accidental evolution of good software, and the contradiction of a component having a good class interface and a bad template interface are the subjects of Chapter 27. In addition, we'll see just how lightweight and, therefore, how fast, some STL collection adaptations can be.

The Component Object Model (COM), a language-independent binary standard for component software, defines its own enumeration and collection models that are quite different from those of the STL. The largest chapter in the book, Chapter 28, covers the adaptation of COM's ***IEnumXXXX*** enumerator protocol to an STL collection, `enumerator_sequence`, that has input or forward iterators. The challenges include the safe handling of reference-counted interfaces, the management of COM resources, exception safety, element caching, COM's non-`const`-correctness, and the contradiction between the runtime determination of the cloneability of COM's enumerators with the compile-time specification of the cloneability of STL's iterators. Though the treatment is decidedly nontrivial, the end result is an adaptation that is exception-safe, resource-safe, succinct, flexible, discoverable, and, when compared to raw C/C++ equivalents, exceedingly expressive. The following intermezzo, Chapter 29, discusses how a mistake in the original design of `enumerator_sequence` can be easily corrected by using the type inference mechanisms discussed in Chapter 13. The COM collection model and its adaptation to the STL collection concept in the form of the `collection_sequence` component are discussed in Chapter 30. The chapter covers how to handle optional features of adapted collections by compile-time and runtime measures and also illustrates how, with a little inside knowledge of `enumerator_sequence`, the collection implementation may be simplified without any sacrifice of robustness.

Chapter 31 is all about refusing to compromise between abstraction and performance when using high-throughout I/O APIs. In addition to describing a mechanism for linearizing multiple discontiguous memory blocks, it illustrates a technique for using low-level, high-speed block transfer functions in combination with standard algorithms, via the legal specialization of elements of the standard library.

In Chapter 32, the influence of scripting languages such as Python and Ruby is felt in C++, whereby a template-based technique may be employed to allow a collection to act as both an array, with integral subscripting, and an associative array, with named element lookup.

The final chapter in this part, Chapter 33, discusses the general problem of external changes to a collection while it is being enumerated (via an STL adaptation), through side effects of the current thread, by action of another thread in the same process, or even by action of an external process. The problems (and solutions) are illustrated with examples from GUIs, system configuration registries, and XML libraries.

This page intentionally left blank

CHAPTER 17

Adapting the glob API

Compromise and accommodation do not end with the design of hardware.

—Henry Petroski

Being lazy may look easy, but it's actually quite hard to pull off.

—Unattributed

17.1 Introduction

In this chapter we're going to take a look at the UNIX **glob** API, the first of four file system enumeration APIs that will be covered in Part II. It provides users with the capability to conduct searches using the rich wildcard matching facilities provided by the UNIX shell(s). Although the `glob()` function is very feature rich and comparatively complex to use compared to the other file system enumeration APIs, it presents a reasonably simple interface for STL extension, which is why we're covering it first.

17.1.1 Motivation

Imagine that you are writing an automatic documentation tool for your software library. You want to programmatically search for all the algorithm files in the various subprojects and then use the results in two separate operations, processing in reverse order for the second. Assume the operations are declared as follows:

```
void Operation1(char const* entry);
void Operation2(char const* entry);
```

Using `glob()`, you could implement this along the lines shown in Listing 17.1.

Listing 17.1 Enumeration of a File System Using the glob API

```
std::string libraryDir = getLibDir();
glob_t      gl;
int         res = ::glob( (libraryDir + "/*/*algo*").c_str()
                        , GLOB_MARK
                        , NULL
                        , &gl);

if(GLOB_NOMATCH == res)
```

```
{
  return 0; // No matching entries to process
}
else if(0 != res)
{
  throw some_exception_class("glob() failed", res);
}
else
{
  // First operation
  { for(size_t i = 0; i < gl.gl_pathc; ++i)
  {
    struct stat st;
    if( 0 == ::stat(gl.gl_pathv[i], &st) &&
        S_IFREG == (st.st_mode & S_IFREG))
    {
      Operation1(gl.gl_pathv[i]);
    }
  }}
  // Second operation
  { for(size_t i = 0; i < gl.gl_pathc; ++i)
  {
    char const* item = gl.gl_pathv[gl.gl_pathc - (i + 1)];
    size_t      len  = ::strlen(item);
    if('/' != item[len - 1]) // Not a directory
    {
      Operation2(item);
    }
  }}

  size_t n = gl.gl_pathc;
  ::globfree(&gl);
  return n;
}
```

In the next section, we'll explore how this code works, discover all the problems, and bemoan the amount of code involved in such a conceptually simple operation. Before that, let's take a look at the STL-ified version, using the **UNIXSTL** `glob_sequence` class, as shown in Listing 17.2.

Listing 17.2 Enumeration of a File System Using `glob_sequence`

```
using unixstl::glob_sequence;
glob_sequence  gls(getLibDir(), "*/*algo*", glob_sequence::files);

std::for_each(gls.begin(), gls.end(), std::ptr_fun(Operation1));
std::for_each(gls.rbegin(), gls.rend(), std::ptr_fun(Operation2));

return gls.size();
```

I trust you'll agree that this is a clear win in discoverability, expressiveness, and flexibility of the client code and that this looks like a compelling case where STL extension can pay dividends. The one possible fly in the ointment might be performance. Have we paid a cost for the abstraction? I created a test program to find out. (`Operation1()` and `Operation2()` were implemented to simply invoke `strlen()` on the given entry.)

Table 17.1 gives the results. The timings were obtained by running the two client forms from Listings 17.1 and 17.2 several hundred times and recording all the execution times. The figures shown are the average of the lowest ten times for each.

Table 17.1 Performance of a Raw API and an STL-Adapted Class

Operating System	Using Raw API	Using `glob_sequence`
Linux (700MHz, 512MB Ubuntu 6.06; GCC 4.0)	1,045 ms	1,021 ms
Win32 (2GHz, 1GB, Windows XP; VC++ 7.1)	15,337 ms	2,517 ms

As you can see, not only is there no performance penalty in using `unixstl::glob_sequence`, there's actually about a 2% gain. Not a bad deal, then.

Just out of interest, I also ran the test program on Windows, using a UNIX emulation-layer library (included on the CD). Note the difference in relative costs of the larger number of visits to the kernel in Windows than in UNIX. This will be of interest to us in the next few chapters.

17.2 The glob API

The **glob** API comprises a structure and two functions, whose declarations are shown in Listing 17.3. The `glob()` function attempts to match the given `pattern`, with behavior moderated according to the given `flags`. If successful, the results, in the form of an array of pointers to C-style strings, are placed into the caller-supplied structure pointed to by `pglob`. The caller may optionally supply a callback function that will be invoked with the path and an `errno` code for any entry that cannot be successfully enumerated. `globfree()` releases any resources allocated by `glob()`.

Listing 17.3 Types and Functions in the glob API

```
struct glob_t
{
  size_t gl_pathc;
  char** gl_pathv;
  size_t gl_offs;
};

int glob( char const* pattern
        , int         flags
        , int        (*errfunc)(char const* errPath, int eerrno)
        , glob_t*     pglob);

void globfree(glob_t* pglob);
```

Note that the precise definition of the `glob_t` structure varies between variants of UNIX. Some use `int` instead of `size_t`; some have additional fields. I'll discuss only the version shown here, which is that defined by POSIX.

If `glob()` succeeds, it returns *0*. Possible error returns are *GLOB_NOSPACE* (memory exhaustion), *GLOB_ABORTED* (read error), and *GLOB_NOMATCH* (no matches found). You can specify a host of flags, including those prescribed by POSIX.

- *GLOB_ERR*: Stop searching and return upon any read error (rather than ignoring the error and continuing the search).
- *GLOB_MARK*: Append a slash to each directory returned.
- *GLOB_NOSORT*: Don't sort the path names. The default behavior is to sort, which presumably has some cost that can be avoided by specifying this flag.
- *GLOB_DOOFFS*: Leave a number of pointers (specified in `gl_offs` prior to the call) vacant for use by the caller. This is useful for preparing `execv` command arrays.
- *GLOB_NOCHECK*: Return the search pattern as the single result in the case where no matches are found.
- *GLOB_APPEND*: Treat `pglob` as pointing to the results of a prior call to `glob()`, and append to the results already contained within in it.
- *GLOB_NOESCAPE*: Meta characters may not be escaped by the backslash.

Some UNIX variants support other nonstandard flags, including the following.

- *GLOB_ALTDIRFUNC*: Use alternate file search functions (specified in additional `glob_t` members, not shown in the definition given earlier) for the operations. This allows globbing from other locations, for example, from files in a tape archive.
- *GLOB_BRACE*: Allow the use of braces (`{}`) to specify alternates, for example, *"{*.cpp,makefile*}"* is equivalent to two calls with *"*.cpp"* and *"makefile*"* patterns.
- *GLOB_NOMAGIC*: If the pattern contains no wildcards, return it as a result if it does not match an entry on the file system.
- *GLOB_TILDE*: Tilde expansion is carried out, so that you can search for such things as *"~/*.rb"* or *"~/sheppie/*.[rp][py]"*.
- *GLOB_ONLYDIR*: Only directories are matched. This is advisory only and cannot be relied on to skip all nondirectory entries.
- *GLOB_TILDE_CHECK*: If *GLOB_TILDE* and a tilde pattern are specified, ignore *GLOB_NOCHECK* and return *GLOB_NOMATCH* on error.
- *GLOB_PERIOD*: Ordinarily, files and/or directories that begin with *'.'* are elided from matching when the leaf-most part of the pattern begins with a wildcard. Specifying this flag allows any such files to appear in the search.
- *GLOB_LIMIT*: Limit the number of entries to that specified by the caller in the additional `gl_matchc` member. In that case, *GLOB_NOSPACE* is returned and the `glob_t` structure is valid (and must be released with `globfree()`).

Since `glob()` may potentially traverse a large part of the file system, it may encounter errors in the search, such as denial of access to some directories. In order to feed back information on such errors to the caller without terminating the search, `glob()`'s third parameter is an error function. Unfortunately, there is no provision for a user-specified context (e.g., `void*`) parameter to be passed into `glob()` and through to the error function, which means that this facility is pretty useless in multithreaded contexts.

17.2 Decomposition of the Longhand Version

Let's now take a look again at the longhand version (Listing 17.1) and mark off the pertinent aspects and the problems in code order.

Line **1**: `getLibDir()` returns the directory where your headers are located.

Line **2**: `gl` is declared, but not initialized, which is fine since neither *`GLOB_APPEND`* nor *`GLOB_LIMIT`* is specified to `glob()`.

Line **3**: The ugly construction `(libraryDir + "/*/*algo*").c_str()` is necessary to get a composite pattern rooted in your library's directory. If we are prepared to modify the `libraryDir` instance, we could write this as the slightly prettier `libraryDir.append("\\*.h").c_str()` since `std::basic_string::append()` returns a self-reference. In either case, we are forced to call `c_str()`. This is a classic case where string access shims (Section 9.3.1) can make client code simpler, more flexible, and more transparent, as we'll see when we start writing the extension class in the next section.

Line **4**: *`GLOB_MARK`* is specified here, which we'll take advantage of in lines 31–33 to assist us in eliding directories from the results.

Lines **8**–**16**: We must manually test the return code to detect whether the search has encountered an error, failed to find any matches, or succeeded in finding one or more matches. In the case of a failure, I've used a notional exception class that can carry both message and error code.

We *could* have combined the tests on lines 8 and 12 to allow the execution to proceed to line 19 in the case of *`GLOB_NOMATCH`*, but that would mean making an unsound assumption. Although I've yet to find an implementation of `glob()` that, on error, does not set the `gl_pathc` and `gl_pathv` members of the `glob_t` instance to *`0`* and *`NULL`*, I have not found any documentation that stipulates that such behavior is required. So we opt to use the given form. In any case, despite being more succinct, it would be less transparent to the reader (i.e., the poor maintenance programmer, who is likely to be you!).

Lines **19**–**27**: Here we enumerate through the array of pointers returned to us by `glob()` and invoke `Operation1()` on each file. Since we are processing only files, and not directories, we must test the type of each entry before passing to `Operation1()`. The `stat()` system call elicits the file information for a given path, including its file type returned in the `st_mode` member of the `struct stat` structure. Hence, we can elide everything that is not a regular file by checking against `S_IFREG` on a successful call to `stat()`. Note that file systems are dynamic things, and it's possible that an entry returned by `glob()` may have been deleted from the file system by the time `stat()`, or `Operation1()`, is called.

Lines **29**–**37**: Here we enumerate through the array of pointers returned to us by `glob()` and invoke `Operation2()` on each file. This loop uses an alternative method of eliding entries. `glob()` provides the facility to append each directory found with a slash by specifying the *`GLOB_MARK`* flag (line 4), presumably so that you do not have to do so when using the results to build up paths. We can now use this facility to elide directories by checking whether the last character of each entry is a slash. Although this involves `strlen()` doing a linear search through each entry, it's reasonable to assume that this will be much faster than calling `stat()` since there is no system call involved. When appending a slash does not impede the use that the client code intends to make of the results, this is a good technique to keep down runtime costs.

Lines **39**–**41**: In order for the code to return the number of elements processed, we need to store the value of `gl.gl_pathc` before calling `globfree()`. Documentation on the **glob** API usually says something along the following lines: "The `globfree()` function releases the dynamically allocated storage from an earlier call to `glob()`." We *could* infer that it leaves the `size_t` members alone and just return `gl.gl_pathc` after the call to `globfree()`, but again this would be ill advised.

Although it might seem like gratuitous pedagogy for me to have presented these two loops and their differing mechanisms for elision of directory entries, there's a serious, real-world point to it: Either the elision of entries must be carried out each time the loop is enumerated, or the filtered results must be copied for reuse.

Thus, we have several obvious flaws with the use of the raw **glob** API. We must prepare and present a one-piece search pattern. We must test the return value and react locally to possible error returns. We must elide file types that are not required. We must manually copy any information from the returned structure before freeing it. We must perform elision each time we enumerate through the results. And that's not the full picture. The code, as written, has some not so obvious issues.

First, if `getLibDir()` returns a directory with a trailing slash, the search pattern will contain a double backslash. Although I can get my main Linux machine to happily glob (and return) paths with doubled slashes—for example, *`/home/matthew//recls`*—I am not aware of any documentation that mandates that this will be so for all `glob()` implementations. And it's unlikely that we can assume `Operation1()`, `Operation2()`, and all such client functions will be so tolerant.

Second, `glob()` does not canonicalize absolute paths. In other words, if your search pattern is `"../*"`, and your current directory is *`/home/matthew/Dev`*, you will receive results beginning with `"../"` and not with *`/home/matthew`*. Although this is by no means a bug, the absence of such functionality can be occasionally inconvenient.

Third, the array of pointers in the `glob_t` structure is of type `char**`, rather than `char const**`. Thus, it's possible for badly written client code to scribble in the contents of the entries. It's a good bet that some or most implementations of `glob()` would be tolerant of that (as long as the extent of the entries were not exceeded), but it's still a potential source of highly vexing porting bugs. Better if it were not allowed in the first place.

Although it's not demonstrated by the example given, when searching for entries beginning with `'.'`, you must manually elide the `"."` (current directory) and `".."` (parent directory) entries from the search in cases where you are following the directories, in order to avoid double processing and/or infinite loops. In my experience, the majority of file system enumeration applications do not need the dots directories.

Finally, the code as written is not exception safe. If either `Operation1()` or `Operation2()` throws an exception, the resources associated with `gl` will be leaked. We could address this by using a scoping class, for example, removing the explicit call to `globfree()` from line 40, and inserting the following line, using `scoped_handle` (Section 16.5) at line 18:

```
stlsoft::scoped_handle<glob_t*>   r(&gl, ::globfree);
```

But that solves just two of the problems: exception safety and the need to explicitly copy `gl.gl_pathc`. And it does so by placing an obligation on the user of the **glob** API to remember to also use a scoping class. As such, it's not really much of a solution. What we need is a full-featured ***Façade*** (or ***Wrapper***) class for `glob()` that will address all the shortcomings.

17.3 `unixstl::glob_sequence`

From its first release, **UNIXSTL** has had a `glob()` wrapper class, albeit subject to several important revisions, ingeniously called `glob_sequence`. Before we look at its interface and the implementation, we should pause and consider what the characteristics of the **glob** API imply about the nature of such an STL collection.

Since `glob()` returns an array of pointers to C-style strings to its caller, we can readily deduce that its iterator type is *contiguous iterator* (Section 2.3.6) and that it therefore also supports reverse iteration automatically.

We can infer from the (gaps in the) API specification that, despite the fact that `gl_pathv` is of type `char**` (rather than `char const*`, or `char* const*`, or even `char const* const*`), we should not attempt to write into the storage pointed to by individual entries, nor should we attempt to move any of the entries around within the entry array. Hence, `glob_sequence` is an immutable collection.

Inevitably, one major aspect of `glob_sequence` is to own the resources, applying RAII (Chapter 11) in calling `globfree()` in its destructor. As such, `glob_sequence` will support fixed element references (Section 3.3.2).

Finally, because file systems may change at any time as a result of actions by arbitrary processes, the class will provide only a snapshot of file system contents. The picture of the collection it provides may become out of date while it is being used.

17.3.1 Public Interface

Now let's look at the public interface of the `glob_sequence` class. It is defined in the `unixstl` namespace. Listing 17.4 shows a minimal class definition, divided into characteristic sections.

Listing 17.4 Declaration of `glob_sequence`

```
// In namespace unixstl
class glob_sequence
{
public: // Member Types
  typedef char                                  char_type;
  typedef char_type const*                      value_type;
  typedef value_type const&                     const_reference;
  typedef value_type const*                     const_pointer;
  typedef glob_sequence                         class_type;
  typedef const_pointer                         const_iterator;
  typedef std::reverse_iterator<const_iterator>
                                                const_reverse_iterator;
  typedef std::allocator<value_type>            allocator_type;
  typedef size_t                                size_type;
  typedef ptrdiff_t                             difference_type;
private:
  typedef filesystem_traits<char>               traits_type;
public: // Member Constants
  enum
  {
        includeDots     =   0x0008
    ,   directories     =   0x0010
    ,   files           =   0x0020
    ,   noSort          =   0x0100
    ,   markDirs        =   0x0200
    ,   absolutePath    =   0x0400
    ,   breakOnError    =   0x0800
    ,   noEscape        =   0x1000
    ,   matchPeriod     =   0x2000
    ,   bracePatterns   =   0x4000
    ,   expandTilde     =   0x8000
  };
public: // Construction
  template<typename S>
  explicit glob_sequence(S const& pattern, int flags = noSort);
  template< typename S1
          , typename S2
          >
  glob_sequence(S1 const& directory, S2 const& pattern
                                    , int flags = noSort);
  ~glob_sequence() throw();
public: // Size
  size_type size() const;
  bool      empty() const;
public: // Element Access
  const_reference operator [](size_type index) const;
public: // Iteration
  const_iterator          begin() const;
  const_iterator          end() const;
  reverse_const_iterator  rbegin() const;
  reverse_const_iterator  rend() const;
  . . .
private: // Member Variables
  . . .
};
```

The breakdown of the class is nice and neat, with just seven nonconstructor methods, all of which are nonmutating.

17.3.2 Member Types

Insofar as `glob_t` could be thought of as a container, its value type would be `char*`. However, as discussed earlier, that could allow client code to scribble on the buffer internals, so `glob_sequence`'s value type is `char const*`. As we'll see, this results in a little casting here and there, but better that inside a library component than to leave the onus on users to do the right thing.

As a consequence of `glob_sequence` being an immutable collection, neither `reference` nor `pointer` nor `iterator` members are provided. (See Chapter 13 for a discussion of the ramifications of this.)

17.3.3 Member Variables

The state of a `glob_sequence` instance is represented in five member variables, shown in Listing 17.5 in an extract of the class definition.

Listing 17.5 `glob_sequence` **Member Variables**

```
private: // Member Variables
  typedef stlsoft::auto_buffer<value_type, 128
                                , allocator_type>   buffer_type_;
  const int         m_flags;
  char_type const** m_base;
  size_t            m_numItems;
  buffer_type_      m_buffer;
  glob_t            m_glob;
};
```

In addition to the `glob_t` member `m_glob`, there are four other members. `m_flags` holds the validated form of the constructor flags. `m_numItems` holds the count of items notionally held by the sequence; due to the filtering functionality provided by `glob_sequence`, the actual number of items is likely to be different from that indicated in `m_glob.gl_pathc`.

The remaining two items are the most interesting. `m_base` is a pointer to the first valid entry pointer to be made accessible to the user. Just as with `m_numItems`, this is not necessarily the same value as that held in `m_glob.gl_pathv`. The `m_buffer` member, an instance of auto_buffer<value_type> (Section 16.2), is used in the case where the array of entries returned by glob() has to be manipulated. If required, each entry—a pointer, not a string, remember—from m_glob.gl_pathv to be presented to the user is copied into it, wherein they may be sorted safely.

17.3.4 Flags

When writing STL collections that support the specification of flags, you have two options: either accept *all* API flags to be passed through to the underlying API, or define extension-specific flags that will be translated by the extension into API flags and accept *only* those values. Although it may not seem immediately obvious, there is no hope of success in attempting to support both forms in one because APIs evolve, and at any time the API could define a flag whose value you are already using for a member constant of your class.

In this case, the choice between the two options is simple. Several facilities offered by glob_sequence do not form part of the semantics of glob(), as represented by the directories, files, and absolutePath flags. As described earlier, some implementations support the *GLOB_ONLYDIR* flag, but not all do so, and no implementations provide a direct facility for returning only files. Hence, glob_sequence supports filtering of files and/or directories depending on whether the user has specified the directories and/or files flags. Also, glob() always gives pattern-relative results; glob_sequence provides the additional facility of returning results in absolute form, in response to absolutePath.

Rule: When providing flag customization in an API ***Façade***, either strictly pass through the API flags to the underlying API functions, or provide flags from a separate value-space and map the values to the API flags. Do not mix the two approaches.

All the other supported flags—noSort (*GLOB_NOSORT*), markDirs (*GLOB_MARK*), breakOnError (*GLOB_ERR*), noEscape (*GLOB_NOESCAPE*), matchPeriod (*GLOB_PERIOD*), bracePatterns (*GLOB_BRACE*), expandTilde (*GLOB_TILDE*)—are translated into their **glob** API values and passed through without further manipulation. However, because some of these are supported only on certain platforms, they are defined in the class definition conditionally, as shown in Listing 17.6.

Listing 17.6 glob_sequence **Member Constants**

```
  public: // Member Constants
    enum
    {
      . . .
#ifdef GLOB_PERIOD
      , matchPeriod     =   0x2000
#endif /* GLOB_PERIOD */
#ifdef GLOB_BRACE
      , bracePatterns   =   0x4000
#endif /* GLOB_BRACE */
#ifdef GLOB_TILDE
      , expandTilde     =   0x8000
#endif /* GLOB_TILDE */
```

To be sure, this is a pretty ugly trick, but there isn't a sensible practical alternative. We could proscribe the use of these flags with `glob_sequence`, but that would unnecessarily hamstring users who need to use these flags on platforms where they are supported. Or we could dummy the values of the requisite enumerators to *0*, or just ignore them, where the underlying flag is not supported, but that would mean that `glob_sequence` would have runtime behavior different from that advertised by its interface on some platforms. Neither option would win many friends. Rather, in this case we deliberately let the abstraction leak.

Tip: Don't pretend support in ***Façades*** for features that have significant semantic differences between the underlying APIs. Be circumspect about purporting support for features that have significant differences in efficiency or complexity.

Note that several flags are not appropriate for use with a wrapper class such as `glob_sequence`, including *GLOB_DOOFFS*, *GLOB_APPEND*, and so on. These are not supported.

Flags are validated in the private `static` method `validate_flags_()`, shown in Listing 17.7. You may not admire the verbosity of the code, but I prefer to keep things in order and in alignment, as an aide-mémoire when such duplication (from the enumeration itself) is required. The double-*0* trick simplifies updating (whether manual or automatic) of such lists. It works because `x | 0` gives `x`, for all `x`. (Use `~0` when doing `&` lists, since `x & ~0` gives `x`, for all `x`.)

Listing 17.7 Validation of Constructor Flags via the `validate_flags_()` Method

```
/* static */ int glob_sequence::validate_flags_(int flags)
{
  const int validFlags  = 0
                        | includeDots
                        | directories
                        | files
                        | noSort
                        | markDirs
                        | absolutePath
                        | breakOnError
                        | noEscape
#ifdef GLOB_PERIOD
                        | matchPeriod
#endif /* GLOB_PERIOD */
#ifdef GLOB_BRACE
                        | bracePattern
#endif /* GLOB_BRACE */
#ifdef GLOB_TILDE
                        | expandTilde
#endif /* GLOB_TILDE */
                        | 0;

  UNIXSTL_MESSAGE_ASSERT( "Specification of unsupported flags"
                        , flags == (flags & validFlags));
  if(0 == (flags & (directories | files)))
  {
    flags |= (directories | files);
  }
  if(0 == (flags & directories))
  {
    // It's more efficient to not bother doing a separate dots check
    // if all directories are being elided.
    flags |= includeDots;

    // Since we're not going to return directories to the caller, and
    // it's more efficient to believe the glob() directory marking
    // rather than calling stat(), we add the markDirs flag here.
    flags |= markDirs;
  }
  return flags;
}
```

This function actually performs three important tasks. First, it validates the flags specified to the constructor against the recognized flags for the given platform. The form of this validation is as a precondition enforcement (Section 7.1) via the `UNIXSTL_MESSAGE_ASSERT()` macro. This macro defaults to a call to `assert()` but may be redefined to anything the user requires and so could be active in a release build.

The second task is to default the search to include both `files` and `directories` when neither is specified explicitly. This is nothing more than a useful service to users who can thus simply specify behavior-modifying flags and take "search everything" as a given.

Finally, there's an implementation optimization. In the case where directories are *not* being returned to the user of `glob_sequence`, the `includeDots` and `markDirs` flags are added to the user's specification. This means that the detection of dots directories (via `strcmp()` with `"."` and `".."`) and the testing of file type (via `stat()`) can be avoided since we can just remove everything that comes back with a trailing slash. We'll see how this works later when we look into the implementation.

17.3.5 Construction

Thanks to the work done by `validate_flags_()` and another private method, `init_glob_()` (Section 17.3.8), the implementations of the constructors and destructor are remarkably straightforward, as shown in Listing 17.8.

Listing 17.8 `glob_sequence` **Constructors and Destructor**

```
typename <typename S>
glob_sequence::glob_sequence(S const& pattern, int flags)
  : m_flags(validate_flags_(flags))
  , m_buffer(1)
```

```
{
  m_numItems = init_glob_(NULL, stlsoft::c_str_ptr(pattern));
  UNIXSTL_ASSERT(is_valid());
}

typename< typename S1
        , typename S2
        >
glob_sequence::glob_sequence(S1 const& directory, S2 const& pattern
                                                 , int flags)
  : m_flags(validate_flags_(flags))
  , m_buffer(1)
{
  m_numItems = init_glob_(stlsoft::c_str_ptr(directory)
                        , stlsoft::c_str_ptr(pattern));
  UNIXSTL_ASSERT(is_valid());
}
. . .
inline glob_sequence::~glob_sequence() throw()
{
  UNIXSTL_ASSERT(is_valid());
  ::globfree(&m_glob);
}
```

In both constructors, validate_flags_() is used to initialize the m_flags member with validated flags, and then init_glob_() is called within the constructor body. This is to avoid member initializer list-ordering issues, since the proper functioning of init_glob_() depends on m_flags and m_buffer being already initialized. Such a mix of initializer list and in-body construction is generally not desirable and should raise a developer's warning flag, which in part explains the validation of the class invariant (via is_valid()) at the end of each constructor body and the start of the destructor (see Chapter 7).

The directory and pattern parameters are passed to init_glob_() via the **c_str_ptr** string access shim. In this way, any type for which the **c_str_ptr** shim is defined may be used, including char const*, std::string, and so on.

Once constructed, the contents of glob_sequence are not changed until destruction: glob_sequence is an immutable collection. Indeed, this is an example of *immutable RAII* (Section 11.1). Hence, the destructor always has just the simple task of calling globfree() to release the resources allocated by the call to glob(); m_buffer clears up after itself.

(Note: This is not the whole story with the constructors of glob_sequence, as will be revealed in Chapter 18, the intermezzo following this chapter. It describes a general technique required when using constructor templates in combination with enumeration arguments without getting snared by the traps of implicit conversions.)

17.3.6 Size and Element Access

The methods `size()`, `empty()`, and `operator []()` (Listing 17.9) are all about as simple as you can get, a consequence of the simplicity of the data representation of the **glob** API.

Listing 17.9 Size Methods

```
size_t glob_sequence::size() const
{
  UNIXSTL_ASSERT(is_valid());
  return m_numItems;
}
bool glob_sequence::empty() const
{
  UNIXSTL_ASSERT(is_valid());
  return 0 == size();
}
const_reference glob_sequence::operator [](size_type index) const
{
  UNIXSTL_MESSAGE_ASSERT( "index out of range in glob_sequence"
                        , index < 1 + size());
  UNIXSTL_ASSERT(is_valid());
  return m_base[index];
}
```

Although there are no mutating methods in the class, it's good contract programming form (Chapter 7) to check the class invariant in all public methods.

Tip: Check class invariants at the start (and end) of all public methods, including nonmutating ones, to maximize early detection of overwrites in errant code elsewhere in the process.

The downside of this policy is that it can appear at first that your code is erroneous, when in fact it's merely being a good C++-itizen and letting users know that *something* is wrong. Our primary concern is correctness, not politics, but you should still be ready with a photocopy of this page if this should happen to you. Managers are notoriously obtuse when it comes to grokking contract programming.

17.3.7 Iteration

The `glob_sequence` iterator type and methods are entirely straightforward: Because `glob()` returns a contiguous block of pointers, we should be able to support the contiguous iterator concept. In other words, the `const_iterator` type is a typedef to `const_pointer` (i.e., `char const* const*`), and `const_reverse_iterator` to `std::reverse_iterator<const_iterator>`. Indeed, this was the reason that I chose to cover this STL collection first, because the application of STL concepts to `glob()` is eminently straightforward (albeit the additional functionality lends a nontrivial amount of effort). Thus the iteration methods are very simple, as shown in Listing 17.10.

Listing 17.10 Iteration Methods

```
const_iterator glob_sequence::begin() const
{
  return m_base;
}
const_iterator glob_sequence::end() const
{
  return m_base + m_numItems;
}
const_reverse_iterator glob_sequence::rbegin() const
{
  return const_reverse_iterator(end());
}
const_reverse_iterator glob_sequence::rend() const
{
  return const_reverse_iterator(begin());
}
```

17.3.8 init_glob_()

The remainder of the implementation resides in the `init_glob_()` private nonstatic worker function, which has the form shown in Listing 17.11.

Listing 17.11 Main Structure of the `init_glob_()` Worker Method

```
size_t glob_sequence::init_glob_( char_type const* directory
                                , char_type const* pattern)
{
  int              glob_flags = 0;
  file_path_buffer scratch; // Scratch buffer for directory / pattern
  size_t           numItems;

  . . . // Derive absolute path, if required
  . . . // Translate flags

  if(0 != ::glob(. . . , &m_glob))
  {
    throw glob_sequence_exception();
  }

  stlsoft::scoped_handle<glob_t*>   cleanup(&m_glob, ::globfree);

  . . . // Elide dots directories if required
  . . . // Filter out files or directories from the results
  . . . // Resort the elements if sorting required and any filtered

  cleanup.detach();
  return numItems;
}
```

Note the use of the `scoped_handle`. You might wonder why we need to apply RAII to the member variable `m_glob`, when the `glob_sequence` class itself manages that resource, invoking `globfree()` on it in its destructor. The reason is that at this point we're still *inside* a `glob_sequence` constructor. C++ guarantees to automatically destroy only fully constructed instances, so throwing an exception from inside the constructor of a class will *not* result in the destructor for that instance being invoked. We therefore take responsibility for the result until the `init_glob_()` call returns (successfully) to the constructor. (The constructor must not make any subsequent calls that have the potential to throw an exception, of course.) The call to `scoped_handle::detach()` ensures that the destructor of the `cleanup` instance does nothing; this effects the transfer of responsibility to the `glob_sequence` instance.

Tip: Remember that C++ causes the automatic destruction of only fully constructed instances. Prefer to use scoping classes for member instances, or, where that's overkill or inappropriate, remember to release already-allocated resources explicitly in constructor bodies when an exception is fired.

With the basic structure of the function established, let's consider the details. We start with the absolute path processing, shown in Listing 17.12.

Listing 17.12 Absolute Path Processing in `init_glob_()`

```
if( NULL == directory &&
    absolutePath == (m_flags & absolutePath))
{
    static const char_type s_thisDir[] = ".";
    directory = s_thisDir;
}
// If a directory is given, then ...
if( NULL != directory &&
    '\0' != *directory)
{
  size_t len;
  // ... optionally turn it into an absolute directory, ...
  if(absolutePath == (m_flags & absolutePath))
  {
    len = traits_type::get_full_path_name(directory
                                  , scratch.size(), &scratch[0]);
  }
  else
  {
    traits_type::str_copy(&scratch[0], directory);
    len = traits_type::str_len(scratch.c_str());
  }
  // ... ensure that it has a trailing path name separator, and ...
```

```
    traits_type::ensure_dir_end(&scratch[0] + (len ? len - 1 : 0));
    // ... prefix directory onto pattern.
    traits_type::str_cat(&scratch[0] + len, pattern);
    pattern = scratch.c_str();
  }
```

If the `absolutePath` flag is specified and a directory is not, `directory` is pointed to a nonmutable static string containing `"."`. Being static, this persists for the lifetime of the program (or the link unit, at least), so it's quite safe to use. And since it's a *local* static, we don't have to define it in an implementation file, so the component can stay 100% header-only, and we need not concern ourselves with the linker.

Tip: Use nonmutable local static character string literals to make well-known values available to components without incurring the inconvenience of separate definition.

Then the `directory`, if specified, is concatenated with the `pattern`. If `absolutePath` is specified, the `get_full_path_name()` method of the `traits_type` (`unixstl::filesystem_traits`; see Section 16.3) is used to first convert `directory` to an absolute form, in the `scratch` file path buffer (Section 16.4) variable. If not, it is simply copied into `scratch`. Another `traits_type` member, `ensure_dir_end()`, places a trailing slash if one does not exist, and then `pattern` is concatenated onto the result. Thus, the doubled slash problem is avoided.

Note that `strcat()` is given an offset destination (`&scratch[0] + len`) in order to avoid the gratuitous waste of cycles in having it find the nul-terminator from the start of the buffer. You might think that `strcpy()` could be used again, but that would cause an error in the case when a slash is appended. Thus `strcat()` is used and will find the nul-terminator in either the first or second character it looks at.

The translation of the flags from the `glob_sequence` values to the **glob** API values is just a lot of `if` statements, with non-POSIX options surrounded by the appropriate preprocessor discrimination. The only nontrivial case is that of *`GLOB_ONLYDIR`*, which is specified when the `directories` flag is specified *without* the `files` flag (Listing 17.13).

Listing 17.13 Flags Processing in `init_glob_()`

```
if(m_flags & noSort)
{
  glob_flags |= GLOB_NOSORT;
}
if(m_flags & markDirs)
{
  glob_flags |= GLOB_MARK;
}
. . .
```

```
#ifdef GLOB_ONLYDIR // If this is not defined, we rely on stat
  if(directories == (m_flags & (directories | files)))
  {
    glob_flags |= GLOB_ONLYDIR;
  }
#endif /* GLOB_ONLYDIR */
#ifdef GLOB_TILDE
  if(m_flags & expandTilde)
  {
    glob_flags |= GLOB_TILDE;
  }
#endif /* GLOB_TILDE */
```

At this point, the `glob()` function is invoked. If it returns a result other than *0*, an instance of `glob_sequence_exception` is thrown, returning the `glob()` return code to the caller. Otherwise, the main post-`glob()` processing commences, involving two main tasks: elision of dots directories and filtering out of files or directories. If either of these is to proceed, however, we must copy the contents of `m_glob.gl_pathv` into `m_buffer`, wherein they can be safely manipulated. This involves resizing `m_buffer` from its initial size of *1* to the actual number of items returned in `m_glob.gl_pathc` (Listing 17.14).

Listing 17.14 Copying of Entries into a Buffer Member in `init_glob_()`

```
    if( 0 == (m_flags & includeDots) ||
        (directories | files) != (m_flags & (directories | files)))
    {
      m_buffer.resize(numItems);
      ::memcpy(&m_buffer[0], base, m_buffer.size() * sizeof(char_type*));
    }
```

If the `resize()` operation fails and throws an exception, the `scoped_handle` instance `cleanup` will ensure that `::globfree()` is called before the exception is propagated to the caller of the `glob_sequence` constructor.

Now that we have a buffer whose contents we can manipulate with assurance, we can see about eliding the dots directories (Listing 17.15).

Listing 17.15 Elision of Dots Directories in `init_glob_()`

```
      char**  base = &m_buffer[0];
      if(0 == (m_flags & includeDots))
      {
        bool    bFoundDot1  = false;
        bool    bFoundDot2  = false;
        char**  begin       = base;
        char**  end         = begin + numItems;
        for(; begin != end && (!bFoundDot1 || !bFoundDot2); ++begin)
        {
          bool bTwoDots = false;
          if(is_dots_maybe_slashed_(*begin, bTwoDots))
          {
            if(begin != base)
            {
              std::swap(*begin, *base);
            }
            ++base;
            —numItems;
            (bTwoDots ? bFoundDot2 : bFoundDot1) = true;
          }
        }
      }
```

This works by enumerating each element in the array in turn and checking to see whether it's one of the dots directories. The `is_dots_maybe_slashed_()` private static worker method returns *`true`* if the path is a dots directory—*`".", "../"`*, or *`"/home/petshop/../"`*, but not `".bashrc"`—and indicates whether it's a dots directory of one or two dots. If the given entry is a dots directory, it is swapped with whatever is at `*base`, and `base` is incremented. Once both dots files are found, the processing stops since we can be quite confident that no self-respecting operating system is going to return more than one of either type of dots directory. Thus, the dots files end up at the start of the contents of `m_buffer`, and `base` refers to the first remaining entry in the set after them. Hence, we've removed the dots files without having to do wholesale `memmove()` calls or reallocation.

The final major action is to filter out files and/or directories. It's a mechanism similar to that for dots elision—swap with `*base` and advance—but it involves tests against file type, as shown in Listing 17.16.

Listing 17.16 File and Directory Filtering in `init_glob_()`

```
    if((m_flags & (directories | files)) != (directories | files))
    {
      file_path_buffer  scratch2;
      char_type**       begin = base;
      char_type**       end   = begin + numItems;
      for(; begin != end; ++begin)
      {
        struct stat       st;
        char_type const*  entry = *begin;
        if(files == (m_flags & (directories | files)))
        {
          UNIXSTL_ASSERT(markDirs == (m_flags & markDirs));
          if(!traits_type::has_dir_end(entry))
          {
            continue; // Nonmarked entry, hence a file, so accept
          }
        }
        else
        {
          if(markDirs == (m_flags & markDirs))
          {
            if(traits_type::has_dir_end(entry))
            {
              continue; // Marked entry, hence a directory, so accept
            }
          }
          else if(0 != ::stat(entry, &st))
          {
            // We could throw an exception here, but it might just be
            // the case that a file has been deleted subsequent to
            // having been included in the glob list. As such, it
            // makes more sense to just kick it from the list.
          }
          else if(S_IFDIR == (st.st_mode & S_IFDIR))
          {
            continue; // A directory, so accept it
          }
        }
        // Swap out with whatever is at base[0]
        std::swap(*begin, *base);
        ++base;
        —numItems;
      }
    }
```

If the user has asked for `files` only, the directories will be marked, and we can just look for, and filter out, any entries with a trailing slash. If the user has asked for `directories` only, it depends on whether he or she also asked for trailing slashes. If so, we can similarly filter out files by the absence of a trailing slash. If not, `stat()` must be called and the results tested against `S_IFDIR`. Note that if `stat()` fails, the assumption is that the operating system entry has been removed or cannot be accessed, in which case it's logical to allow it to fall through and be swapped out.

The final processing element is to reinstate the sort (if required) and assign `m_base` from `base` (Listing 17.17).

Listing 17.17 Optional Sorting of Elements in `init_glob_()`

```
    if( 0 == (m_flags & noSort) &&
        numItems != static_cast<size_t>(m_glob.gl_pathc))
    {
      std::sort(base, base + cItems);
    }
```

All that remains is to call `detach()` on the `scoped_handle` instance and return the number of items (shown earlier in Listing 17.11).

17.4 Decomposition of the Shorthand Version

Now that we understand the workings of glob_sequence, we can take a look again at the shorthand version (Listing 17.2).

Line **1**: Use the glob_sequence from the unixstl namespace. We could instead explicitly qualify, but we'd have to qualify both the type of gls and of the files enumerator flags, so it's less typing to use a using declaration.

Line **2**: Construct a glob_sequence instance, passing in the return of getLibDir(), the search pattern, and the filter enumerator flag files. The constructor template invokes the **c_str_ptr** shim on both directory (std::string) and pattern (char const*) arguments, to elicit C-style strings (char const*) from each. The flags parameter will be passed through validate_flags_(), which will add the includeDots and markDirs flags in order to minimize filtering overhead. The strings and flags are then passed into init_glob_(), which invokes glob() and processes the results. The argument handling within init_glob_() means that it is irrelevant whether getLibDir() returns a trailing slash or not: The resultant path will be correctly formed. If init_glob_() succeeds, the glob_sequence instance is fully constructed and owns the resources of its m_glob member, which will be released in the destructor, called after line 7. If init_glob_() fails, it throws an exception, and control passes to the caller.

Line **4**: The full range iterators are elicited from gls via glob_sequence::begin() and glob_sequence::end() and passed to std::for_each() along with the std::ptr_fun-bound address of Operation1(). std::for_each() will enumerate through the range, passing each element to Operation1().

Line **5**: A reverse enumeration is effected by calling the glob_sequence::rbegin() and glob_sequence::rend() methods and passing the resultant reverse iterators and a std::ptr_fun-bound Operation2() to std::for_each().

Line **7**: glob_sequence::size() is invoked to return the number of elements retrieved by the globbing. Since it's not possible to call this method on the instance after its destruction, there's no issue with accessing an out-of-date value.

17.5 Summary

After such a long discussion, it behooves us to look at what has been achieved.

- const-correctness: Users of glob_sequence are now prevented from making any const-incorrect actions.
- Encapsulation: Users of glob_sequence need not be concerned that the elements arrived in one array and have, perhaps, been partially transferred to another. They use the established STL idioms of access via iterators and/or subscript.
- RAII: The resources are now managed within the glob_sequence instance and are automatically released upon its destruction.
- Filtering: Selection of files and/or directories and elision of the (usually unwanted) dots directories are all handled by the class and do not place any burden on users.

- Flexibility: `glob_sequence` may be constructed on any string type for which string access shims have been defined.
- Power: `glob_sequence` provides additional flexibility over `glob()`, insofar as it is able to translate the search directory to an absolute form, and automatically ensures that directory and pattern are spliced appropriately.
- Efficiency: The value type is `char const*`; measures such as the use of *`GLOB_MARK`* attempt to take advantages of possible implementation optimizations without incurring likely costs absent such optimizations. The cost of one construction (with inherent search) is amortized across an arbitrary number of accesses to the resultant sequence (in forward, reverse, or random access fashion).

To be sure, the implementation of `glob_sequence` is quite involved. That's typical of many general-purpose library components since they must work (correctly) in a range of contexts. One of the reasons I've gone into such depth here is because I believe it's essential to present real-world examples for such an important topic as STL extension, and real-world examples have dark corners. It's my intention throughout the book to highlight such things, not to gratuitously distract you from the main message of STL extension, but rather as a regular reminder to you (and me) how such issues can (and usually do) impact the design of STL extensions, their semantics, robustness, and, importantly, efficiency. Remember, as noted in Chapter 6, abstractions are leaky!

In this case, virtually all the complexity is contained within the `init_glob_()` method and pertains to interacting with the functionality of the underlying **glob** API. The STL-related aspects—contiguous iterators, fixed element references, immutable collections, and so on—are all eminently straightforward; indeed, perhaps as straightforward as they come. Even for people who don't like or use the STL, `glob_sequence` is still a great choice. Since it supports random access, they can just use `size()` and the subscript operator.

In Chapter 19, we will see the polar opposite situation, wherein another UNIX enumeration API, which has very simple semantics, results in complex STL extension functionality and presentation of more restrictive STL concepts. But first, that little intermezzo I mentioned earlier.

CHAPTER 18

Intermezzo: Constructor Clashes and Design That Is, If Not Bad, At Least Ill-Conceived for Seamless Evolution

Good design adds value faster than it adds cost.

—Thomas C. Gale

At this point, I must confess that the constructors shown in Listing 17.8 and described in Section 17.3.5 are not actually the real form. To understand the actual form and the reasons for differing from the ideal presented earlier, we must consider the history of the class.

The original version of the class had just two, non-template, constructors:

```
glob_sequence(char_type const*  directory
            , int               flags = noSort); // NT1
glob_sequence(char_type const*  directory
            , char_type const*  pattern
            , int               flags = noSort); // NT2
```

For good or ill, this means that updates to the class must endeavor to maintain backward compatibility. Ostensibly, the constructor forms mooted in Section 17.3.5 appear to do this quite well:

```
template<typename S>
explicit glob_sequence(S const& pattern, int flags = noSort); // T1
template< typename S1
        , typename S2
        >
glob_sequence(S1 const& directory, S2 const& pattern
                                 , int flags = noSort); // T2
```

Indeed, the second overload uses two template parameters to support a heterogenous mix of string types for the `directory` and `pattern` arguments.

Tip: Maximize the flexibility of function/method templates by using a separate template parameter for each relevant function parameter.

Unfortunately, the presence of the default parameters means that there are possible ambiguities in the interpretation of some constructions. Consider the following declarations:

```
string_type   s = "*";
glob_sequence gls1("*");                    // Works with NT1 and T1
glob_sequence gls2("*"
                  , glob_sequence::noSort); // Works with NT1; not T1
glob_sequence gls3("*", glob_sequence::noSort |
                  glob_sequence::markDirs); // Works with NT1 and T1
```

With `T1` and `T2` available, the compiler will select the three-parameter form when constructing `gls2`. The reason is quite subtle but is of importance to anyone writing portable, flexible libraries. Although enumerators are implicitly convertible to `int`, they are *not* of type `int`. In fact, `glob_sequence::noSort` is actually of type `glob_sequence::`*`_anonymous_enum_`*, where *`_anonymous_enum_`* is a compiler-dependent generated name. (Comeau 4.3.3 and Intel 8 report the type as `glob_sequence::<unnamed>`; Digital Mars as `glob_sequence::__unnamed`; GCC as `glob_sequence::<anonymous enum>`. Visual C++ 7.1 is arguably more meaningful, but decidedly less helpful, in referring to it as `' '`.)

Thus, when presented with any type other than `int`, the compiler will select the other overload. Since arithmetic operations on enumerators are of type `int`, as is the case with `noSort | markDirs`, the construction of `gls3` is fine. Perversely, we can make the construction of `gls2` compilable—with most compilers, anyway—as follows:

```
glob_sequence gls2("*"
               , glob_sequence::noSort | 0); // Works with NT1 and T1
```

But of course it's complete nonsense to expect anyone to use libraries that require such a measure. The sensible approach is to define additional overloads to enable the use of a single enumerator. To do this, we need to give the enumerator a tag name that we can then refer to in the function signature (Listing 18.1).

Listing 18.1 Flexibility Enhancements to `glob_sequence`

```
class glob_sequence
{
  . . .
public: // Member Constants
  enum search_flags
  {
    . . .
  };
public: // Construction
  template<typename S>
```

```
explicit glob_sequence(S const& pattern, int flags = noSort); // T1
template<typename S>
explicit glob_sequence(S const& pattern, search_flags flag); // T1b
template< typename S1
        , typename S2
        >
glob_sequence(S1 const& directory, S2 const& pattern
                                  , int flags = noSort); // T2
template< typename S1
        , typename S2
        >
glob_sequence(S1 const& directory, S2 const& pattern
                                  , search_flags flag); // T2b
. . .
```

Tip: Overload by using `int` and enumeration types for arguments of enumeration type that are used as flags and may be combined.

CHAPTER 19

Adapting the opendir/readdir API

Don't write 200 lines of code when 10 will do.

—Hal Fulton

STL is the meat and grist of your dissertation,
stick with it and use it like a heart patient wants to use butter.

—George Frazier

19.1 Introduction

In this chapter we're going to look at the simple UNIX **opendir**/**readdir** API and see how, despite having far simpler semantics than the **glob** API (Chapter 17), the business of writing a serviceable STL extension for it is more involved and results in a collection with more restricted semantics.

The extension described here will be the first involving iterators of class type, something it's very important to get right. Consequently, I'll first show you the wrong way to write such classes, and then I'll show you the right way.

19.1.1 Motivation

Consider that we wish to enumerate the subdirectories of an application directory, given by `getWorkplaceDirectory()`, and to store the full path of each one in a vector of strings, for later use, perhaps to display to a user in a dialog. Listing 19.1 shows how we would achieve that using the raw **opendir**/**readdir** API.

Listing 19.1 Enumerating Directories Using the opendir/readdir API

```
std::vector<std::string> getWorkplaceSubdirectories()
{
  std::string              searchDir = getWorkplaceDirectory();
  std::vector<std::string> dirNames;
  DIR*                     dir = ::opendir(searchDir.c_str());
  if(NULL == dir)
  {
    throw some_exception_class("Cannot enumerate dirs", errno);
  }
  else
  {
    struct dirent* entry;
    if('/' != searchDir[searchDir.size() - 1])
    {
      searchDir += '/';
    }
    for(; NULL != (entry = ::readdir(dir)); )
    {
      if( '.' == entry->d_name[0] &&
          ( '\0' == entry->d_name[1] ||    // "."
            ( '.' == entry->d_name[1] &&
              '\0' == entry->d_name[2])))  // ".."
      {
        // Dots directory, so skip it
      }
      else
      {
        struct stat st;
        std::string entryPath = searchDir + entry->d_name;
        if(0 == ::stat(entryPath.c_str(), &st))
        {
          if(S_IFDIR == (st.st_mode & S_IFDIR))
          {
            dirNames.push_back(entryPath);
          }
        }
      }
    }
    ::closedir(dir);
  }
  return dirNames;
}
```

Contrast this with the alternate version using `readdir_sequence` shown in Listing 19.2.

Listing 19.2 Enumerating Directories Using `readdir_sequence`

```
std::vector<std::string> getWorkplaceSubdirectories()
{
  using unixstl::readdir_sequence;
  readdir_sequence  rds(getWorkplaceDirectory()
    , readdir_sequence::directories | readdir_sequence::fullPath);
  return std::vector<std::string>(rds.begin(), rds.end());
}
```

Just as with `glob_sequence` (Section 17.3), it's a fait accompli in terms of number of lines, robustness (particularly exception safety), expressiveness, and discoverability. Once again, the area of doubt is in performance. And once again we're in for a pleasant surprise. The results shown in Table 19.1 were obtained in the same manner as for `glob_sequence`; this time the task was to enumerate the contents of the top-level directory in the current **STLSoft** distribution directory, on the respective platform.

Table 19.1 Performance of the Raw API and the STL-Adapted Class

Operating System	Using Raw API	Using `readdir_sequence`
Linux (700MHz, 512MB Ubuntu 6.06; GCC 4.0)	2,487 ms	2,091 ms
Win32 (2GHz, 1GB, Windows XP; VC++ 7.1)	16,040 ms	15,790 ms

Another all-around win. Let's look at how to achieve this.

19.1.2 The opendir/readdir API

The **opendir**/**readdir** API consists of four standard and two nonstandard functions and a structure (which has one mandatory field), as shown in Listing 19.3.

Listing 19.3 Types and Functions in the opendir/readdir API

```
struct dirent
{
  char  d_name[]; // Name of the enumerated entry
  . . .           // Other, nonstandard, fields
};
struct DIR;       // Opaque type, representing open directory search
DIR*           opendir(const char* dir);  // Starts search of dir
int            closedir(DIR*);            // Close search
struct dirent* readdir(DIR*);             // Read next entry
void           rewinddir(DIR*);           // Restart search
long int       telldir(DIR*);             // Get current search pos
void           seekdir(DIR*, long int);   // Reposition search
```

The first four functions—`opendir()`, `closedir()`, `readdir()`, and `rewinddir()`—are prescribed by the POSIX standard; `telldir()` and `seekdir()` are extensions. We will focus on the use of the first three functions for the remainder of this chapter. `opendir()` opens a directory search for a given path and, if successful, returns a non-*NULL* value of the opaque type `DIR*`, which then identifies the search instance. `readdir()` retrieves each successive entry in the search, returning *NULL* when all entries have been enumerated or on error. `closedir()` closes the search and releases any resources allocated by `opendir()` and/or `readdir()`. The value returned by `readdir()` is a pointer to `struct dirent`, which must have at least the field `d_name`, which is either a character buffer or a pointer to a character buffer, containing the name of the entry in the search directory.

19.2 Decomposition of the Longhand Version

Let's now look again at the longhand version (Listing 19.1) and mark the pertinent aspects and problems in code order.

Line **3**: We take a mutable copy of the workplace directory so that we may, if necessary, append a trailing slash in lines 13–16, so that when we need to concatenate it with each entry for passing the full path to `stat()` (line 30), it will be validly formed.

Line **4**: Declare an instance of `std::vector<std::string>`, to which we will add the directory entries as they're enumerated.

Lines **5–9**: Initiate a search, using `opendir()`, and throw a suitable exception if it fails.

Line **13**: If `getWorkplaceDirectory()` returns an empty string, this line will access an element at `size_type(0)−1`, which is likely to be *0xFFFFFFFF* or some equally huge value, depending on the size of `size_type`. Whatever its actual value, it will most certainly result in an access violation and a crash. To make this line safe, we'd have to also test `searchDir.size()` against *0*. I don't know about you, but this is the kind of thing that gives me a toothache when writing application code.

Lines **19–25**: Some versions of `readdir()` include the dots directories—`"."` and `".."`—in their enumeration results, so these lines test for their presence, while being careful not to elide entries that merely begin with one or two dots. (It's not pretty, is it?)

Line **29**: Concatenate the search directory and the current enumerated entry name to form a full path for the entry. Note that this is a *new* instance of `std::string` for every enumerated entry, including the allocation (and release) of memory to store the result.

Lines **30–32**: Use the `stat()` system call to test whether the given entry is a directory. Passing the entry name alone to `stat()` would work only when the enumeration is conducted in the current directory, which is clearly not the case here.

Line **34**: Add the full path name to the container.

Line **39**: Close the search, which releases any resources associated with it.

Line **41**: Return the filtered results to the caller.

As with the **glob** API in the previous chapter, the longhand form is a combination of verbosity, inconvenience, inefficiency, and subtle bugs. Ensuring a trailing slash on `searchDir`, manually testing for dots directories, and having to call `c_str()` on string instances are inconvenient and lead to verbose code. Having to call `stat()` itself to filter out directories is also something you would ideally leave to a library. The most obvious inefficiency is that a new instance of `entryPath` means at least one visit to the heap for each enumerated entry, but there's also more subtle inefficiency in the fact that the explicit declaration of `dirNames` means that the function return is available only for the named return value optimization (NRVO) and not the return value optimization (RVO). (When compared with the runtime costs of enumerating file system contents, the failure to apply an RVO is likely to bite not a lot. But this situation can crop up in enumerations whose relative runtime costs are considerably different, so I thought I'd bring it to your attention.)

Even if we could endure the other issues, the fact is that this code is not exception safe. Lines 15, 29, and 34 all have the potential to throw an exception, in which case the release of `dir` at line 39 will never happen.

Tip: Mixing C++ (particularly STL) coding with C APIs invariably means exposing many holes through which resources can leak, usually, but not exclusively, as a result of exceptions. Prefer to use ***Façade*** (or ***Wrapper***) classes where available and suitably functionality-rich and efficient.

Absent such classes, consider writing your own. Even if all you gain is RAII and a few defaulted parameters, this is a big improvement in robustness (not to mention that the experience will gain you a concomitant leap in quality of practice).

19.3 `unixstl::readdir_sequence`

Before we define the `readdir_sequence` class, we should look at what the **opendir**/**readdir** API implies about the characteristics of an STL extension. Here are the important features.

- The **opendir**/**readdir** API provides indirect access to the elements of the collection, indicating that an iterator class, rather than a pointer, is required.
- The current directory entry enumeration position may be advanced one element at a time, but only wholly rewound. Neither *bidirectional* nor *random access* iteration (Section 1.3) requirements are fulfilled by a facility for wholesale rewind, so the API will support, at best, the *forward iterator* category.
- Each call to `readdir()` advances the position of the underlying search object. Hence, a given search, initiated by `opendir()`, can support only single-pass iteration, so the iterator category will be *input iterator*.
- The API provides no facility for changing the contents of a directory, so it supports only non-mutating access.
- `readdir()` returns `struct dirent*`, whose only standard-prescribed (and, therefore, portable) member is `d_name`, holding a nul-terminated string of the given entry. This suggests that the collection value type should be `char const*`.
- There is no guarantee that successive calls to `readdir()` return a pointer to the same `struct dirent` instance with overwritten contents or to separate instances. This suggests that the iterator instance should hold a pointer to `struct dirent`, rather than a pointer to its `d_name` member.
- Each call to `opendir()` produces a separate search. Thus, the invocation of `opendir()` should be associated with a call to the `readdir_sequence::begin()` method. It is not yet clear (and actually turns out not to matter) whether the collection calls `opendir()` and passes the resultant `DIR*` to the iterator class or whether the iterator class calls `opendir()` using information provided to it by the collection.
- Similarly, it is not yet clear whether the collection makes the first call to `readdir()`, to commence the search proper, or whether that is done by the iterator.
- The search is advanced by subsequent calls to `readdir()`, which should happen within the iterator's increment operator.
- In order for a single collection instance to support multiple distinct enumerations, the iterator class should be given ownership of the `DIR*` search handle, since it is the increment of the

iterator that brings a search to a close (whether by advancing to the `end()` position or by the iterator instance going out of scope).

Given this analysis, we can stipulate the following: Value type will be `char const*`; element reference category will be transient; iterator category will be input iterator; and collection mutability will be immutable. We can postulate an interface for `readdir_sequence` along the lines of that shown in Listing 19.4.

Listing 19.4 Initial Version of `readdir_sequence`

```
// In namespace unixstl
class readdir_sequence
{
private: // Member Types
  typedef char                            char_type;
public:
  typedef char_type const*                value_type;
  typedef std::basic_string<char_type>    string_type;
  typedef filesystem_traits<char_type>    traits_type;
  typedef readdir_sequence                class_type;
  class                                   const_iterator;
public: // Member Constants
  enum
  {
      includeDots   = 0x0008
    , directories   = 0x0010
    , files         = 0x0020
    , fullPath      = 0x0100
    , absolutePath  = 0x0200
  };
public: // Construction
  template <typename S>
  readdir_sequence(S const& dir, int flags = 0);
public: // Iteration
  const_iterator  begin() const;
  const_iterator  end() const;
public: // Size
  bool empty() const;
public: // Attributes
  string_type const&  get_directory() const; // Always trailing '/'
  int                 get_flags() const;
private: // Implementation
  static int          validate_flags_(int flags);
  static string_type  validate_directory_(char const* directory
                                        , int         flags);
private: // Member Variables
  const int         m_flags;
  const string_type m_directory;
private: // Not to be implemented
  readdir_sequence(class_type const&);
  class_type& operator =(class_type const&);
};
```

19.3.1 Member Types and Constants

The member types include a `string_type` (which we'll need later), the value type, and a forward declaration of the nested class `const_iterator`. There are two main options when writing collection-specific iterators. They can be defined either, as here, as nested classes whose names correspond to the member type role they're fulfilling or as separate classes, for example, `readdir_sequence_const_iterator`, which are then introduced as member types via `typedef`. A number of factors are involved in deciding between the two, such as whether the collection is a template, the level of (human) tolerance for verbose type names, whether the iterator type can serve for more than one collection (Section 26.8), and so on.

Tip: Define iterator classes that are used for one collection class as nested classes. This reduces namespace pollution and makes a clear point about the relationship of the iterator and its sequence to users.

The constants moderate the collection behavior. `includeDots`, `directories`, and `files` have the same meaning as they did for `glob_sequence`. We'll defer discussion of `fullPath` and `absolutePath` until later in the chapter (Section 19.3.11).

The `traits_type` is defined from `unixstl::filesystem_traits` (Section 16.3), which provides abstraction of various facilities required in the implementation. The member type `char_type`, defined to be `char`, is used throughout the class definition to cater for the possibility that the class may one day be converted to a template in order to work with the wide-character analog of the **opendir**/**readdir** API. (It defines the `wDIR` and `struct wdirent` types and manipulates them using `wopendir()`, `wreaddir()`, and so on.)

Tip: Avoid *DRY SPOT* violations by defining member types to provide a single type from which other types are defined.

`traits_type` is defined to be public even though, by rights, it should be private, because `const_iterator` needs to see it.

19.3.2 Construction

Unlike the hoops incurred with `glob_sequence` (Section 17.3.5, Chapter 18), the public construction methods of `readdir_sequence` comprise a single constructor, made flexible by use of the **c_str_ptr** string access shim (Section 9.3.1), as shown in Listing 19.5.

Listing 19.5 `readdir_sequence` **Constructor Template**

```
class readdir_sequence
{
  . . .
```

```
public: // Construction
  template <typename S>
  readdir_sequence(S const& dir, int flags = 0)
    : m_directory(stlsoft::c_str_ptr(dir))
    , m_flags(validate_flags_(flags))
  {}
  . . .
```

The `validate_flags_()` method is not further discussed here since it performs a service equivalent to the method of the same name in `glob_sequence` (Section 17.3.4).

According to the *Law of the Big Two*, there's no need for a destructor since the only resources associated with an instance are bound up in the `string_type`—the compiler-generated version suffices. Thus, this type could, without any additional input from the author, support the *Assignable* and *CopyConstructible* requirements of the *STL container* concept (C++-03: 23.1;3). However, the copy constructor and copy assignment operator have been proscribed. Why? The reason is that `readdir_sequence`, just like `glob_sequence` (and pretty much any other file system enumeration API), provides only a snapshot of the system at a given point in time. Disallowing copy semantics prevents the user from easily forgetting this fact.

Tip: Consider proscribing operations from your types for conveying information on appropriate use, as well as for classic robustness and correctness reasons, particularly for collection types that provide snapshots of their underlying collections.

19.3.3 Iteration and Size Methods

The iteration methods `begin()` and `end()` return instances of `const_iterator`. There are no mutating forms and no reverse iteration forms, due to the constraints attendant in the nonmutating single-pass nature of the **opendir/readdir** API.

As for the size methods, the characteristics of the **opendir/readdir** API mean that only `empty()` is defined; there is no `size()` method. This may seem like a glaring omission, and in a way it is. But because the API gives an element at a time, the only portable way to implement a `size()` method would be as follows:

```
size_type readdir_sequence::size() const
{
  return std::distance(begin(), end());
}
```

In terms of the method semantics, this implementation is perfectly valid. However, rather than being a constant-time operation—something expected (though not required!) of standard containers (C++-03: 23.1), and generally expected of all STL extension collections—it is likely to be along the lines of O(n) (depending on the implementation of the underlying **opendir/readdir** API). Hence, the syntax and semantics may be the same, but the complexity would be significantly different. It's a Goose Rule thing (Section 10.1.3).

Therefore, in order to communicate to users of the class that such an action is potentially expensive, the method is omitted. Users can still do it manually if they require, but the absence of the method leaves no user in any doubt about the ramifications.

Tip: Where appropriate, omit methods from STL extension classes that deliver complexities that differ significantly compared with those from STL concepts. In short: Don't promise what you can't deliver.

As a consequence, users who need to do multiple passes or to know the extent before enumeration are more likely to do a single pass of the directory using `readdir_sequence` and store the results in a container, for example, `std::vector<std::string>`. You might wonder at the apparent duplication of values and the multiple reallocations involved in their being added (via `std::back_inserter()` or equivalent) an element at a time. However, the fact is that each enumeration of N directory entries is at least $2 + N$ system calls, whereas the creation and storage of copies of those elements may, depending on the memory allocation optimizations provided by your standard library, incur no system calls at all. It's very hard to imagine any operating system where the former would prove to be faster. (Tests on my Mac OS X and Windows XP boxes, repeatedly enumerating 512 files in a single directory, show that the strategy of taking copies is between two and three times as fast as enumerating the directory contents each time. The test program is included on the CD.)

The `empty()` method *is* provided, however, because it involves only the "opening" of the search (via one call each to `opendir()` and `readdir()`), which will be, within small variation, a constant-time operation (albeit not one without a nontrivial fixed expense, of course). There are no element access methods because the underlying API is single-pass.

19.3.4 Attribute Methods

Because, as discussed earlier, we've proscribed copy semantics, the methods `get_directory()` and `get_flags()` are provided in case a user may wish to repeat a search (albeit that its results may differ, of course).

`get_directory()` provides nonmutating access to the `m_directory` internal member, which is guaranteed to have a trailing path name separator (`'/'`), by `validate_directory_()`. `get_flags()` returns the effective flags, after they've been validated. For example, if `includeDots` is specified to the constructor, `get_flags()` will return `includeDots | directories | files`.

Although each method may return values different from those specified to the constructor, using these values with another instance would result in identical search results as long as the file system contents do not change.

19.3.5 `const_iterator`, Version 1

Now we come to the definition of `const_iterator`, a nested class within `readdir_sequence`. A first attempt at this might have the definition shown in Listing 19.6. There are several things wrong or missing from this definition, but we'll take it as a base from which to work toward a proper implementation.

Listing 19.6 Initial Version of `readdir_sequence::const_iterator`

```
class readdir_sequence::const_iterator
{
public: // Member Types
  typedef char const*          value_type;
  typedef const_iterator       class_type;
private: // Construction
  friend class readdir_sequence; // Give sequence access to conv ctor
  const_iterator(DIR* dir, string_type const& directory, int flags);
public:
  const_iterator();
  ~const_iterator() throw();
public: // Iterator Methods
  class_type& operator ++();
  class_type  operator ++(int);
  char const* operator *() const;
  bool        equal(class_type const& rhs) const;
private: // Member Variables
  DIR*            m_dir;
  struct dirent*  m_entry;
  int             m_flags;
};
bool operator ==( readdir_sequence::const_iterator const& lhs
                , readdir_sequence::const_iterator const& rhs);
bool operator !=( readdir_sequence::const_iterator const& lhs
                , readdir_sequence::const_iterator const& rhs);
```

The iterator is given ownership of the `DIR*` via a private conversion constructor because we don't want to allow the underlying API to leak out of the ***Façade***. Thus, `readdir_sequence` is declared as a friend of the class, so it can invoke this constructor, in its `begin()` method, as shown in Listing 19.7.

Listing 19.7 Iteration Methods

```
readdir_sequence::const_iterator readdir_sequence::begin() const
{
  DIR* dir = ::opendir(m_directory.c_str());
  if(NULL == dir)
  {
    throw readdir_sequence_exception("Can't enumerate dir", errno);
  }
  return const_iterator(dir, m_directory, m_flags);
}
readdir_sequence::const_iterator readdir_sequence::end() const
{
  return const_iterator();
}
```

The implementations of the major functions of `const_iterator` are as follows. The constructor initializes the members and then calls `operator ++()` to move the search to the first entry (or `end()` if the search result set is empty):

```
readdir_sequence::const_iterator::const_iterator(DIR* dir
                          , string_type const& directory, int flags)
  : m_directory(directory)
  , m_dir(dir)
  , m_entry(NULL)
  , m_flags(flags)
{
  operator ++();
}
```

The destructor releases the `DIR*`, if it hasn't already been released by `operator ++()`:

```
readdir_sequence::const_iterator::~const_iterator() throw()
{
  if(NULL != m_dir)
  {
    ::closedir(m_dir);
  }
}
```

`operator *()` simply returns (a pointer to) the entry name, doing a little precondition test before it does so.

```
char const* readdir_sequence::const_iterator::operator *() const
{
  UNIXSTL_MESSAGE_ASSERT("Dereferencing invalid iterator"
                        , NULL != m_dir);
  return m_entry->d_name;
}
```

The `equal()` method, used to support (in)equality comparison (Chapter 15), is implemented in terms of `m_entry`.

```
bool
 readdir_sequence::const_iterator::equal(const_iterator const& rhs)
  const
{
  UNIXSTL_ASSERT(NULL == m_dir || NULL == rhs.m_dir ||
                 m_dir == rhs.m_dir);
  return m_entry == rhs.m_entry;
}
```

```
bool operator ==( readdir_sequence::const_iterator const& lhs
                , readdir_sequence::const_iterator const& rhs);
bool operator !=( readdir_sequence::const_iterator const& lhs
                , readdir_sequence::const_iterator const& rhs);
```

Unfortunately, there's a subtle bug here. I mentioned earlier that `readdir()` may return, on each subsequent call, (a pointer to) the same `struct dirent` instance with different contents or may return different instances. The implementation of `equal()` fails in the former case. We're not going to correct this bug now since it will be automatically fixed for us when we address a much worse bug, which we'll discuss in the next section.

That just leaves us with the preincrement and postincrement operators. The postincrement operator conforms to the canonical form shown in Listing 19.8.

Listing 19.8 Canonical Form of the Postincrement Operator

```
const_iterator readdir_sequence::const_iterator::operator ++(int)
{
  class_type  r(*this);
  operator ++();
  return r;
}
```

That's the last time I'll draw out a postincrement operator implementation in full. In the future I'll just refer to this canonical form, and you'll (hopefully) know what I'm talking about. (It's exactly the same with postdecrement operators, of course, for iterator types that support decrementation.)

Tip: Implement postincrement/postdecrement operators in canonical form, via the preincrement/predecrement forms.

The preincrement operator is something of a behemoth, as shown in Listing 19.9.

Listing 19.9 Implementation of the Initial Version of the Preincrement Operator

```
const_iterator& readdir_sequence::const_iterator::operator ++()
{
  UNIXSTL_MESSAGE_ASSERT("Incrementing invalid iterator"
                        , NULL != m_dir);
  for(;;)
  {
    errno = 0;
    m_entry = ::readdir(m_dir);
    if(NULL == m_entry)
    {
      if(0 != errno)
      {
       throw readdir_sequence_exception("Enumeration failed", errno);
      }
    }
    else
    {
      if(0 == (m_flags & includeDots))
      {
        if(traits_type::is_dots(m_entry->d_name))
        {
          continue; // Don't want dots; skip it
        }
      }
      if((m_flags & (directories | files)) != (directories | files))
      {
        traits_type::stat_data_type st;
        string_type                 scratch(m_directory);
        scratch += m_entry->d_name;
        if(!traits_type::stat(scratch.c_str(), &st))
        {
          continue; // Failed to stat. Assume it is dead, so skip it
        }
        else
        {
          if(m_flags & directories) // Want directories
          {
            if(traits_type::is_directory(&st))
            {
              break;  // It is a directory, so accept it
            }
          }
          if(m_flags & files) // Want files
          {
            if(traits_type::is_file(&st))
            {
              break;  // It is a file, so accept it
            }
          }
          continue; // Not a match, so skip this entry
        }
      }
    }
    break; // Break out of loop, to accept the entry
  }
  if(NULL == m_entry) // Check whether enumeration is complete
  {
    ::closedir(m_dir);
    m_dir = NULL;
  }
  return *this;
}
```

Although, as I've already conceded, there are fundamental problems with this implementation, it contains many of the features of a correct implementation.

- There is a single call to `readdir()`.
- It correctly handles *NULL* returns from `readdir()` (via set and test of `errno`).
- It evaluates equality in terms of `m_dir`.
- It elides dots directories via a call to `filesystem_traits::is_dots()`. This tests the entry name before a possible call to `stat()`, which is more efficient.
- It correctly passes the full path to `stat()`.
- It filters out files or directories. It uses `filesystem_traits::is_directory()` and `filesystem_traits::is_file()`, in preference to tests such as the far less transparent and error-prone `if(S_IFREG == (st.st_mode & S_IFREG))`.
- The constructor calls `operator ++()` to effect the first `readdir()` call in a manner consistent with all subsequent `readdir()` calls.
- If there are no more elements, `operator ++()` closes the search handle and sets it to *NULL*, to indicate to `equal()` that it is complete. The destructor tests against *NULL* in order to close iterators that have not reached `end()`.

Tip: In cases where the iterator needs an initial positioning, via an equivalent call to the same underlying API function that effects subsequent movements, prefer to organize your implementation so that the iterator constructor calls `operator ++()` (or a function common to both), and avoid special cases (and extra testing).

Note that this version supports the `includeDots`, `files`, and `directories` flags. We'll get to the support for `fullPath` and `absolutePath` when we've pointed out all the problems with this version.

19.3.6 Using Version 1

Let's take it for a spin. The following code compiles and executes perfectly:

```
typedef unixstl::readdir_sequence seq_t;

seq_t rds(".", seq_t::files);

for(seq_t::const_iterator b = rds.begin(); b != rds.end(); ++b)
{
  std::cout << *b << std::endl;
}
```

However, the alternate form, using the `std::copy()` algorithm, crashes after printing out all the entries in the current directory:

```
readdir_sequence  rds(".", readdir_sequence::files);

std::copy(rds.begin(), rds.end()
        , std::ostream_iterator<char const*>(std::cout, "\n"));
```

The reason is pretty fundamental. Standard algorithm iterator parameters are by-value, which means that the iterator arguments you pass to them are copied.

Tip: Always remember that the standard algorithms take their iterator arguments by-value (except for `std::advance()`).

We've made a cardinal C++ programming error in the definition of `const_iterator`: The class directly manages resources but does not have copy semantics explicitly defined. The implicitly defined copy constructor and/or assignment operator supplied by the compiler simply copy the `m_dir` and `m_entry` pointer values, so when the second "owner" of the `DIR*` calls `closedir()`, it meets a sticky end. We can attempt to address this very simply by proscribing the copy constructor and copy assignment operator (Listing 19.10).

Listing 19.10 Proscription of Copy Operations for `const_iterator`

```
private: // Not to be implemented (for now)
  const_iterator(class_type const& rhs);
  class_type& operator =(class_type const& rhs);
};
```

Alas, this is more kill than cure since there's now no way to access the iterator returned from `begin()`. About the only thing we can do is call `readdir_sequence`'s constructor!

19.3.7 `const_iterator`, Version 2: Copy Semantics

The problem with the first implementation is that each instance of the constructor is greedy with its underlying iteration state and doesn't want to share. This has the obvious problem of crashing now and then, but it also has the subtler issue of incorrect equality comparison. So we need to act like a canny parent and find a way for them all to play nice, without ado. The solution is shown in Listing 19.11. The nested `const_iterator` class itself declares a nested class, `shared_handle`. You might have a couple of questions about this class: Why are the `AddRef()` and `Release()` methods in `ThisCase()`, rather than `this_case()`? Why do they return non-`void`? The answers are that the case follows a convention (borrowed from COM) that indicates such types are readily compatible, without further adaptation, with **STLSoft**'s lock shims and `ref_ptr` class template, and the non-`void` return allows me to follow the ref-count in the debugger by watching the contents of the Intel EAX register.

Listing 19.11 Definition of the `shared_handle` **Nested Class**

```
struct readdir_sequence::const_iterator::shared_handle
{
public: // Member Types
  typedef shared_handle   class_type;
public: // Member Variables
  DIR*      m_dir;
  sint32_t  m_refCount;
public: // Construction and Lifetime Management
  explicit shared_handle(DIR* d)
    : m_dir(d)
    , m_refCount(1)
  {}
  sint32_t AddRef()
  {
    return ++m_refCount;
  }
  sint32_t Release()
  {
    sint32_t  rc = --m_refCount;
    if(0 == rc)
    {
      delete this;
    }
    return rc;
  }
private:
  ~shared_handle() throw()
  {
    UNIXSTL_MESSAGE_ASSERT("Context destroyed with outstanding refs!"
                         , 0 == m_refCount);
    if(NULL != m_dir)
    {
      ::closedir(m_dir);
    }
  }
private: // Not to be implemented
  shared_handle(class_type const&);
  class_type& operator =(class_type const&);
};
```

This class acts as a ref-counted handle for the `DIR*` resource and is responsible for providing shared ownership and for releasing the resource, via a call to `closedir()`, when the last reference is released.

Rule: When implementing input iterators over an element-at-a-time API that does not provide shared access to the enumeration state, you must use a shared context to obtain correct copying and comparison semantics between iterator instances.

Naturally, such handle classes can be abstracted and made generic. There are four reasons I've not done so here: reduced coupling, portability, discoverability, and stability of implementation. Making such handle class templates is a nontrivial undertaking, both to understand and to make portable across compilers; it's more involved than `scoped_handle` (Section 16.5). It also introduces compile-time coupling, although that's not terribly significant here. Finally, there's no particularly strong motivation, since this ref-counted handle class is not going to change (and has not done so throughout the lifetime of the `readdir_sequence` class). Using the shared handle, the actual definition of `const_iterator` is as shown in Listing 19.12.

Listing 19.12 Definition of `readdir_sequence::const_iterator`

```
class readdir_sequence::const_iterator
  : public std::iterator<. . .> // To be determined . . .
{
public: // Member Types
  typedef char const*        value_type;
  typedef const_iterator     class_type;
private: // Construction
  friend class readdir_sequence; // Give sequence access to conv ctor
  const_iterator(DIR* dir, string_type const& directory, int flags);
public:
  const_iterator();
  const_iterator(class_type const& rhs);
  ~const_iterator() throw();
  class_type& operator =(class_type const& rhs);
public: // Iteration
  class_type& operator ++();
  class_type  operator ++(int);
  char const* operator *() const;
  bool        equal(class_type const& rhs) const;
private: // Member Variables
  struct shared_handle;
  shared_handle*  m_handle;
  struct dirent*  m_entry;
  int             m_flags;
  string_type     m_scratch;
  size_type       m_dirLen;
};
```

19.3.8 `operator ++()`

The only changes required for a full and correct definition of the preincrement operator are shown in Listing 19.13.

Listing 19.13 Implementation of the Preincrement Operator: Use of Shared Handle

```
const_iterator& readdir_sequence::const_iterator::operator ++()
{
  UNIXSTL_MESSAGE_ASSERT("Incrementing invalid iterator"
                        , NULL != m_handle);
  for(;;)
  {
    . . .
  }
  if(NULL == m_entry) // Check whether enumeration is complete
  {
    UNIXSTL_ASSERT(NULL != m_handle);
    m_handle->Release();
    m_handle = NULL;
  }
  return *this;
}
```

Rather than calling `closedir()` when the file entries are exhausted, the iterator releases the shared handle. One more change to the operator is required, but that has to do with handling full paths and will be discussed in Section 19.3.11.

19.3.9 Iterator Category and Adaptable Member Types

As I alluded to at the start of the chapter, `readdir_sequence::const_iterator` has nonmutating single-pass semantics (i.e., input iterator), and it exhibits the transient element reference category. This is communicated to the outside world by a specialization of `std::iterator`, from which it derives. Given the input iterator and transient categories, we could expect to see the specialization as shown in Listing 19.14.

Listing 19.14 Possible Definition of Iterator Types

```
class readdir_sequence::const_iterator
  : public std::iterator< std::input_iterator_tag
                        , readdir_sequence::value_type
                        , ptrdiff_t
                        , readdir_sequence::value_type*
                        , readdir_sequence::value_type&
                        >
  . . .
```

This is all right and proper, but it leads to a slight anomaly in this case. Given the definition above, the iterator's `reference` type is `char const*&`. This actually causes a compilation error in the dereference operator. Let's draw out the method with typedefs expanded:

```
char const*& readdir_sequence::const_iterator::operator *()
{
  return m_dir->d_name; // Error: reference cannot bind to non-lvalue
}
```

The problem, as the compiler suggests, is that a reference can only be bound to an *lvalue*, and `m_dir->d_name` is an *rvalue*. To support this return type, the iterator type would have to maintain a pointer member of type `char const* const*` called, say, `m_pref`, and use it thus:

```
char const*& readdir_sequence::const_iterator::operator *()
{
  m_pref = &m_dir->d_name[0];
  return m_pref;
}
```

Not surprisingly, I am loath to implement such a thing. There's nothing to choose between returning `char const*` and `char const*&` in terms of efficiency, and returning the latter provides only the pyrrhic advantage of allowing client code to alter what `m_pref` points to. Consequently, `std::iterator` is specialized for `readdir_sequence::const_iterator` as if it supports only the by-value temporary category (Section 3.3.5), as shown in Listing 19.15.

Listing 19.15 Definition of the Iterator Element Reference Category

```
class readdir_sequence::const_iterator
  : public std::iterator< std::input_iterator_tag
                        , readdir_sequence::value_type, ptrdiff_t
                        , void, readdir_sequence::value_type
                        >
  . . .
```

19.3.10 operator ->()

Providing `operator ->()` for `readdir_sequence::const_iterator` is a nonissue since its value type is `char const*`, which obviously isn't a class type. Were `struct dirent` to have several POSIX standard members, rather than just `d_name`, it might have been appropriate to make the value type `struct dirent`, in which case `operator ->()` would have been defined to return `struct dirent const*`. But it's not, so we don't.

19.3.11 Supporting fullPath and absolutePath

We'll now complete the examination of `readdir_sequence` and come to the final implementation of `const_iterator`, as we consider the `fullPath` and `absolutePath` flags.

Another of the bugbears about using the **opendir**/**readdir** API is that the results are only the names of the entries. In order to get the full path, you must concatenate them with the search directory, as shown earlier in Listing 19.9. If `fullPath` is specified, each entry returned is a concatenation of the search directory—which always has a trailing slash, remember—and the entry name.

This is implemented by changing the logic inside `operator ++()` that concatenates the directory and the entry name before calling `stat()` to use the member variables `m_scratch` and `m_dirLen`, as shown in Listing 19.16.

Listing 19.16 Implementation of the Preincrement Operator: Path Concatenation

```
const_iterator& readdir_sequence::const_iterator::operator ++()
{
  . . .
    if((m_flags & (fullPath | directories | files)) !=
          (directories | files))
    {
        // Truncate the scratch to the directory path, ...
        m_scratch.resize(m_dirLen);
        // ... and add the file
        m_scratch += m_entry->d_name;
    }

    if((m_flags & (directories | files)) != (directories | files))
    {
        // Test file type, using stat
        traits_type::stat_data_type st;
        if(!traits_type::stat(m_scratch.c_str(), &st))
        {
          . . .
}
```

In the case where directories or files need to be filtered, or the `fullPath` flag is specified, the full path is constructed by truncating `m_scratch` (which is copy constructed in the `const_iterator` constructor from the `readdir_sequence`'s directory) to a length of the `const` member `m_dirLen` (which is initialized to the length of the `readdir_sequence`'s directory). Thus, the string `m_scratch` is reused, and the number of constructions of `string_type` per `const_iterator` instance drops from 1 per enumerated entry to 1 in total. Also, because the string is trimmed back only to the directory, it's likely that there will be few reallocations, if any, during the enumeration of all the entries.

Indeed, there's actually a further optimization. In cases where the *`PATH_MAX`* symbol is defined (see Section 16.4), the string type is actually a specialization of `stlsoft::basic_static_string`, as shown in Listing 19.17. This is a `basic_string`-like class template that maintains a fixed internal array of characters—in this case, of *`PATH_MAX + 1`* dimension—and does no heap allocation at all.

Listing 19.17 Member Type Definitions When *`PATH_MAX`* Is Defined

```
private: // Member Types
  typedef char                                  char_type;
public:
  typedef char_type const*                      value_type;
#if defined(PATH_MAX)
  typedef stlsoft::basic_static_string< char_type
                                      , PATH_MAX
                                      >   string_type;
#else /* ? PATH_MAX */
```

```
  typedef std::basic_string<char_type>    string_type;
#endif /* !PATH_MAX */
  typedef filesystem_traits<char_type>    traits_type;
  . . .
```

Because `m_scratch` is guaranteed to have the full path when `fullPath` is specified, `operator *()` may return the appropriate value (Listing 19.18).

Listing 19.18 Implementation of the Dereference Operator

```
char const* readdir_sequence::const_iterator::operator *() const
{
  UNIXSTL_MESSAGE_ASSERT( "Dereferencing invalid iterator"
                        , NULL != m_entry);
  if(readdir_sequence::fullPath & m_flags)
  {
    return m_scratch.c_str();
  }
  else
  {
    return m_entry->d_name;
  }
}
```

If the search directory specified is a relative one, the results enumerated with the flag `fullPath` will also be relative. So, the final aspect to the `readdir_sequence` is the support for `absolutePath` that ensures, via `prepare_directory_()`, that the directory specified to the constructor is absolute (Listing 19.19). *NULL* or an empty string is interpreted to mean the current working directory; as we saw in `glob_sequence::init_glob_()` (Section 17.3.8), a nonmutable local static character string is used to provide the value.

Listing 19.19 Implementation of the `prepare_directory_()` Worker Method

```
string_type
 readdir_sequence::prepare_directory_(char_type const* directory
                                    , int              flags)
{
  if( NULL == directory ||
      '\0' == *directory)
  {
    static const char_type  s_thisDir[] = ".";
    directory = s_thisDir;
  }
  basic_file_path_buffer<char_type>   path;
  size_type                           n;
  if(absolutePath & flags)
  {
    n = traits_type::get_full_path_name(directory, path.size()
                                      , &path[0]);
    if(0 == n)
    {
      throw readdir_sequence_exception("Failed to get path", errno);
    }
  }
  else
  {
    n = traits_type::str_len(traits_type::str_n_copy( &path[0]
                                    , directory, path.size()));
  }
  traits_type::ensure_dir_end(&path[n - 1]);
  directory = path.c_str();
  return directory;
}
```

And that's it.

19.4 Alternate Implementations

Unlike `glob_sequence`, it's not immediately clear why an STL extension of the **opendir**/ **readdir** API should take the form of a collection rather than an iterator. Well, the fact is that there is no overriding, compelling reason. For reasons of habit, style, and consistency, I chose to do it that way. Because I have tended to do most extensions as collections, rather than as iterators, that has become a habit and therefore a style. For consistency, I prefer it that way and save stand-alone iterator classes for iterator adaptors.

But that's just me. You may choose to do it the other way. There are pros and cons to both approaches.

19.4.1 Storing the Elements as a Snapshot

Reason for: If you need to process the same result set multiple times, you will pay for enumeration costs only once.

Reason against: Snapshots age, so to get a fresh result you'd need to create a new instance.

It's not a big thing, to be sure, but in the case where you need to process a directory's fresh contents twice and process each result set only once, you're paying for what you don't need. In cases where you wish to repeatedly process each result set, you can simply and easily copy into a container, as demonstrated in the original motivation function (see Listing 19.2).

19.4.2 Storing the Elements as an Iterator

The following code shows how we might implement `getWorkplaceSubdirectories()` in terms of such an iterator class:

```
std::vector<std::string> getWorkplaceSubdirectories()
{
  using unixstl::readdir_iterator;
  return std::vector<std::string>(
              readdir_iterator(getWorkplaceDirectory()
                             , readdir_iterator::directories
                             | readdir_iterator::fullPath)
            , readdir_iterator());
}
```

Reason for: This approach can be more succinct in some circumstances. The implementation of `getWorkplaceSubdirectories()` shown here is now only one statement, albeit the constraints of space in this forum coupled with my obsession with code alignment means it's two extra lines of code. I'm not sure that it's an improvement on the `readdir_sequence` version, though; for me it's less obvious, but I am, of course, biased by my own habits and style.

Reason for: This form is more "honest" in the sense that each search is a new, and potentially different, traversal of the file system.

Reason against: An iterator is modeled on a pointer, so any kind of operation called on class type iterator instances, for example, `readdir_iterator::get_directory()`, seems unnatural. And to qualify member constants in terms of an iterator seems positively baroque. Further, as we'll see in Volume 2, adapting collections is a simple and rewarding technique, rendering the argument of iterator-only succinctness moot, to say the least.

Reason against: This approach doesn't work with collection adaptors.

19.5 Summary

Even though the **opendir**/**readdir** API has very simple and straightforward semantics, its element-at-a-time nature means that it can support only the input iterator category and, consequently, an iterator class is required. I have demonstrated a design for a collection based around a collection class and a nonmutating iterator class, and we examined the relationship between the two, noting the following important features.

- The iterator class must share state to fulfill single-pass semantics (and not crash!).
- The omission of the `size()` method, because it cannot be constant time, communicates to users the performance ramifications of using the collection, that is, that a file system enumeration API cannot provide the complexity characteristics expected of STL collections.
- Implementation in terms of the `filesystem_traits` utility component, to abstract away various tests, simplifies the implementation and provides a measure of future compatibility should the class be changed to a template to support both `char` and `wchar_t` file system enumeration.
- The addition of functional enhancements, such as directory/file filtering and elision of the dots directories, simplifies client code and enhances overall robustness and, potentially, performance.

- The use of private static worker methods for preprocessing constructor arguments allows class members to be `const`, which improves robustness and communicates design decisions forward to any future maintainers of the component.
- The use of the `equal()` comparison method (Chapter 15) supports non-`friend` comparison operators, improving portability and transparency.

You might wonder whether the implementation is overcomplicated by some of the efficiency measures. It does not transgress the *Principle of Optimization* since the design is not compromised. It does somewhat transgress the *Principle of Clarity,* but general-purpose library code must not significantly detract from performance. The performance tests discussed in the start of the chapter show this is not the case for `readdir_sequence`, so I'd argue it was worth it in this case.

C H A P T E R 2 0

Adapting the FindFirstFile/FindNextFile API

Design must always conform to constraint, must always require choice, and thus must always involve compromise.

—Henry Petroski

Writing public libraries is the best way to make sure you write good code. It's like: write good code as thou knowest thou should, or shame thyself in public.

—Adi Shavit

20.1 Introduction

In this chapter we build on the knowledge gained from the previous chapters on `glob_sequence` (Chapter 17) and `readdir_sequence` (Chapter 19), by examining the file system enumeration facilities available on Windows in the form of the **FindFirstFile**/**FindNextFile** API. Although this API is similar to **opendir**/**readdir** API in that it is single-pass and returns one element at a time, it also has some features in common with the **glob** API and some additional features all its own. We will consider the ramifications of these features for the design of an STL collection, the class template `winstl::basic_findfile_sequence`.

20.1.1 Motivation

In customary style, we'll look at a longhand version first (Listing 20.1). After the last two chapters, I found myself short on interesting hypothetical scenarios, so I did a quick search of my source database and discovered the following code from a very old utility of mine. I mention the "very old" aspect since it's not exactly beautiful and is far from representative of my current style. (Any codebuncles contained within are entirely the fault of my former self, and I take responsibility only so far as a modern politician is wont to do.) The only thing I've changed is to add in the three **Pantheios** logging calls—it formerly posted Windows-y messages to its host process—and to remove the few comments (which were wrong!).

Listing 20.1 Implementation of the Example Function Using the FindFirstFile/FindNextFile API

```
1  void ClearDirectory(LPCTSTR lpszDir, LPCTSTR lpszFilePatterns)
2  {
3    TCHAR           szPath[1 + _MAX_PATH];
```

```
TCHAR           szFind[1 + _MAX_PATH];
WIN32_FIND_DATA find;
HANDLE          hFind;
size_t          cchDir;
LPTSTR          tokenBuff;
LPTSTR          tok;

pantheios::log_DEBUG(_T("ClearDirectory("), lpszDir, _T(", ")
                   , lpszFilePatterns, _T(")"));
::lstrcpy(szFind, lpszDir);
if(szFind[::lstrlen(lpszDir) - 1] != _T('\\'))
{
  ::lstrcat(szFind, _T("\\"));
}
cchDir    = ::lstrlen(szFind);
tokenBuff = ::_tcsdup(lpszFilePatterns); // strdup() or wcsdup()
if(NULL == tokenBuff)
{
  pantheios::log_ERROR(_T("Memory failure"));
  return;
}
else
{
  for(tok = ::_tcstok(tokenBuff, ";"); NULL != tok;
      tok = ::_tcstok(NULL, ";"))
  {
    ::lstrcpy(&szFind[cchDir], tok);
    hFind = ::FindFirstFile(szFind, &find);
    if(INVALID_HANDLE_VALUE != hFind)
    {
      do
      {
        if(find.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
        {
          continue;
        }
        ::lstrcpy(szPath, lpszDir);
        ::lstrcat(szPath, _T("\\"));
        ::lstrcat(szPath, find.cFileName);
        if(::DeleteFile(szPath))
        {
          pantheios::log_NOTICE( _T("Successfully deleted ")
                                , szPath);
          ::SHChangeNotify(SHCNE_DELETE, SHCNF_PATH, szPath, 0);
        }
        else
        {
            pantheios::log_ERROR(_T("Unable to delete "), szPath
                , _T(": "), winstl::error_desc(::GetLastError()));
          }
        }
        while(::FindNextFile(hFind, &find));
        ::FindClose(hFind);
      }
    }
    ::free(tokenBuff);
  }

  ::lstrcpy(szFind, lpszDir);
  ::lstrcat(szFind, _T("\\*.*"));
  hFind = ::FindFirstFile(szFind, &find);
  if(INVALID_HANDLE_VALUE != hFind)
  {
    do
    {
      if( (find.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) &&
          ::lstrcmp(find.cFileName, _T(".")) &&
          ::lstrcmp(find.cFileName, _T("..")))
      {
        ::lstrcpy(szPath, lpszDir);
        ::lstrcat(szPath, _T("\\"));
        ::lstrcat(szPath, find.cFileName);
        ClearDirectory(szPath, lpszFilePatterns); // Recurse
      }
    }
    while(::FindNextFile(hFind, &find));
    ::FindClose(hFind);
  }
}
```

This function is called with the directory and the file pattern and goes off and merrily removes all matching files. It's a lot of code, most of it tedious string manipulation and explicit resource handling. Now contrast this with the STL version, using the `winstl::basic_findfile_sequence` class template, shown in Listing 20.2.

Listing 20.2 Implementation of the Example Function Using `basic_findfile_sequence`

```
void ClearDirectory(LPCTSTR lpszDir, LPCTSTR lpszFilePatterns)
{
  typedef winstl::basic_findfile_sequence<TCHAR>  ffs_t;

  pantheios::log_DEBUG(_T("ClearDirectory("), lpszDir, _T(", ")
                     , lpszFilePatterns, _T(")"));
  ffs_t files(lpszDir, lpszFilePatterns, ';', ffs_t::files);
8    { for(ffs_t::const_iterator b = files.begin(); b != files.end();
9          ++b)
10    {
11     if(::DeleteFile((*b).c_str()))
12     {
13       pantheios::log_NOTICE(_T("Successfully deleted file ")
14                                , *b);
15       ::SHChangeNotify(SHCNE_DELETE, SHCNF_PATH, (*b).c_str(), 0);
16     }
17     else
18     {
19       pantheios::log_ERROR(_T("Unable to delete file "), *b
20                    , _T(": "), winstl::error_desc(::GetLastError()));
21     }
22   }}
23
24   ffs_t dirs(lpszDir, _T("*.*"), ffs_t::directories |
25                                  ffs_t::skipReparseDirs);
26   { for(ffs_t::const_iterator b = dirs.begin(); b != dirs.end();
27         ++b)
28   {
29     ClearDirectory((*b).c_str(), lpszFilePatterns);
30   }}
31 }
```

Note that the log statements in lines 13–14 and 19–20 do not need to invoke the `c_str()` method on the arguments because, as described in Section 9.3.1, **Pantheios** is implicitly compatible with any types for which the **c_str_data_a** and **c_str_len_a** string access shims are defined. Such overloads for `findfile_sequence::value_type` (which is actually a specialization of `basic_findfile_sequence_value_type`) are defined and exported to the `stlsoft` namespace (as we'll show later in this chapter). (The same thing applies to the `winstl::basic_error_desc` class template, such that the temporary instance of [the specialization] `winstl::error_desc` on line 20 can also be passed directly to the log call.)

Because the Windows API functions don't understand string access shims, we need to be explicit throughout the rest of `ClearDirectory()`. (In Volume 2 we'll see how standard C runtime library functions and those in operating system and third-party APIs can be integrated with shims.)

20.1.2 The FindFirstFile/FindNextFile API

The **FindFirstFile**/**FindNextFile** API consists of two structures and seven functions (as shown in Listing 20.3). Two of the functions are an optimization provided only on the NT family of operating systems; we'll cover these later in the chapter. The five main functions are two pairs of `FindFirstFileA/W()` and `FindNextFileA/W()` and `FindClose()`. The API function `FindFirstFile()` is actually a `#define` to either `FindFirstFileA()` (`char`) or `FindFirstFileW()` (`wchar_t`), depending on the ambient character encoding (itself dependent on the presence or absence of the `UNICODE` preprocessor symbol). Similarly, `FindNextFileA()` (`char`) and `FindNextFileW()` (`wchar_t`) underpin the API function `FindNextFile()`, and `WIN32_FIND_DATAA` (`char`) and `WIN32_FIND_DATAW` (`wchar_t`) underpin the API structure `WIN32_FIND_DATA`. Except where the character encoding is significant, from here on I will refer to them all without the `A`/`W` suffixes.

Listing 20.3 Types and Functions in the FindFirstFile/FindNextFile API

```
HANDLE FindFirstFileA(char const* searchSpec
                    , WIN32_FIND_DATAA* findData);
HANDLE FindFirstFileW(wchar_t const* searchSpec
                    , WIN32_FIND_DATAW* findData);
BOOL   FindNextFileA(HANDLE hSrch, WIN32_FIND_DATAA* findData);
BOOL   FindNextFileW(HANDLE hSrch, WIN32_FIND_DATAW* findData);
BOOL   FindClose(HANDLE hSrch);

struct WIN32_FIND_DATAA
{
  DWORD     dwFileAttributes;
  FILETIME  ftCreationTime;
  FILETIME  ftLastAccessTime;
  FILETIME  ftLastWriteTime;
  DWORD     nFileSizeHigh;
  DWORD     nFileSizeLow;
  CHAR      cFileName[MAX_PATH];
  CHAR      cAlternateFileName[14];
};
struct WIN32_FIND_DATAW
{
  . . . // Same as WIN32_FIND_DATAA, except:
  WCHAR     cFileName[MAX_PATH];
  WCHAR     cAlternateFileName[14];
};
```

`FindFirstFile()` returns a search handle and populates the caller-supplied instance of the `WIN32_FIND_DATA` structure if `searchSpec` matches one or more files and/or directories. Otherwise it returns *`INVALID_HANDLE_VALUE`* `(`*`-1`*`)`. If successful, the caller can use the handle to enumerate any further elements of the search by calling `FindNextFile()` repeatedly, until it returns *`false`*. Once the search is fully enumerated, or the caller is not interested in subsequent entries, `FindClose()` must be invoked to release any resources allocated to the search instance.

Note that the type of the search handle is the Windows opaque type `HANDLE`, which is usually closed by the `CloseHandle()` function (the one-stop shop for releasing kernel objects).

The **FindFirstFile**/**FindNextFile** API, like **glob**, supports wildcard searches. However, this facility is limited.

- The pattern matching understands only the `?` and `*` special characters.
- Only one pattern at a time is allowed. You cannot search for, say, *`"*.cpp;makefile.*"`*.

- Only the rightmost part of the putative path specified in searchSpec may contain wildcard characters: *"H:\publishing\books\XSTLv1\pre*.doc"* is okay, but *"H:\publishing\books\*\preface.doc"* is not.

In other respects, the API is much more similar to the **opendir**/**readdir** API in the way the functions are invoked by the caller, except that FindFirstFile() is the logical equivalent to the call to opendir() *plus* the first call to readdir(). In both cases, the caller makes further calls to elicit subsequent entries and then must close the search when no longer needed. Thus, we get a strong hint that the STL collection will support input iteration and will have much in common with readdir_sequence (Section 19.3).

However, these two APIs differ significantly in the way they return information and in what information is returned. The **FindFirstFile**/**FindNextFile** API writes all the available information for a search entry into a caller-supplied structure, rather than returning a pointer to a structure instance residing inside the API. Further, the WIN32_FIND_DATA structure contains not just the name of the entry (in the structure member cFileName, the analog of struct dirent::d_name), but also much of the information available from the struct stat structure used by the stat() call required by the implementations of glob_sequence (Section 17.3) and readdir_sequence. This is a significant point. Rather than having to make additional calls to ascertain attribute information about a file system entry in order to perform filtering, that information is already present. Indeed, the **FindFirstFile**/**FindNextFile** API is the only part of the publicly documented Windows API (available on all Windows platforms) with which you can ascertain all such information from a file name.

This difference is one of the main influences on the design of the basic_findfile_sequence class template and its worker classes, as we'll see throughout this chapter. Note that, unlike **glob** on UNIX, there is no API on Windows that provides multipart pattern searching. Again, this factors into the design of basic_findfile_sequence.

20.2 Decomposition of Examples

Before we dig in, let's get some ammunition by examining the different versions.

20.2.1 Longhand Version

Looking back at the code shown in Listing 20.1, we see it goes through the following steps.

Lines **13–18**: Manually form the first part of the file search pattern, by concatenating the search directory with a path name separator, if necessary.

Lines **19–28**: Make a mutable copy of the lpszFilePatterns parameter, and tokenize it using _tcstok(), which is a macro that resolves to strtok() or wcstok(), for multibyte and wide strings, respectively. (See Chapter 27 for a description of these and other tokenization functions and an STL-compliant sequence that provides much nicer tokenization functionality.)

Line **30**: Manually form the full search pattern for each pattern in the set specified, by concatenating tok to the directory (and path name separator) formed in lines 13–18.

Lines **31–32**: Open the search, testing for success or failure. Note that `FindFirstFile()` fails if there are no matching entries as well as if the search pattern or directory is invalid. This is in contrast to `opendir()`, which will always return a non-*NULL* `DIR*` for a valid directory even if it will yield only the dots directories or, on some systems, no entries at all.

Lines **36–39**: Filter out directories.

Lines **40–42**: Manually form the path of the file to be deleted.

Lines **43–53**: Attempt to delete the file, logging success or failure. If the file is successfully deleted, the invocation of `SHChangeNotify()` informs the Windows shell of the change so it can adjust any necessary displays that may include the deleted file.

Line **55**: Get the next entry matching the search pattern or terminate the loop.

Line **56**: Release the search handle.

Line **59**: Release the scratch buffer used by `_tcstok()`.

Lines **62–63**: Manually form the directory search pattern by concatenating the search directory, a path name separator, and the Windows "all entries" pattern `"*.*"`.

Lines **64–65**: Open the search, testing for success or failure.

Lines **69–71**: Filter out files and dots directories (to avoid infinite loops).

Lines **73–75**: Manually form the path of the subdirectory into which we will recurse.

Line **76**: Recurse.

Line **79**: Get the next entry matching the search pattern or terminate the loop.

Line **80**: Release the search handle.

Wow! That's a whole lot of code. As with `readdir()`, the API returns only file and directory names, so a lot of the effort here is simply building correctly formed full paths. Then there's filtering—*again with the dots directories!*—and ensuring that search resources are properly released. And, of course, because we've mixed in some C++ classes, which may throw exceptions, the whole thing lacks exception safety. This is not fun coding. Enter the STL extensions.

20.2.2 Shorthand Version

Looking back at Listing 20.2, we see a much more succinct alternative when using the `basic_findfile_sequence` component.

Line **3**: Define a suitable (i.e., short) local typedef for the specialization of the search sequence class template.

Line **7**: Declare an instance, `files`, passing the search directory, the search pattern(s), the delimiter (`';'`), and the filtering flag `ffs_t::files`. This instance will return all files matching the given patterns in the given directory.

Lines **8–22**: These are functionally identical to lines 13–60 of Listing 20.1.

Lines **24–25**: Declare an instance, `dirs`, passing the search directory, the "all entries" search pattern, and the filtering flags `ffs_t::directories` and `ffs_t::skipReparseDirs`. This instance will return only directories in the given directory. (We'll look at the `skipReparseDirs` flag in a moment.)

Lines **26–30**: These are functionally identical to lines 62–81 of Listing 20.1.

In addition to the succinctness, the reduction in effort, and the increase in discoverability, this variant is also exception safe, which is rather more important than usual when you are manipulating kernel objects. And I would challenge even the most ardent STL-phobic C-phile to claim that the longhand version is more transparent.

Note that on line 3 I've written out the specialization overtly. In fact, the *<winstl/filesystem/findfile_sequence.hpp>* header contains three full specializations of the class template, as follows:

```
typedef winstl::basic_findfile_sequence<char>     findfile_sequence_a;
typedef winstl::basic_findfile_sequence<wchar_t>  findfile_sequence_w;
typedef winstl::basic_findfile_sequence<TCHAR>    findfile_sequence;
```

The vast majority of Windows coding that is character-encoding-aware is done in a way that supports conditional compilation, rather than overtly aiming at the ANSI/multibyte or Unicode variants. Both versions of `ClearDirectory()` demonstrate this by following recommended practice in using the `_T()` literal macro and the `TCHAR`, `LPTSTR`, and `LPCTSTR` typedefs. By providing the sequence typedefs, a user merely has to use the common symbol `findfile_sequence` and can forget that it's a specialization of a class template.

20.2.3 Reparse Points and Infinite Recursion

Hopefully you've wondered what the `skipReparseDirs` flag is all about. (If you haven't: *Wake up!*) Over and above the other issues, there's a bug in the longhand version (Listing 20.1). Windows 2000 and later versions of the NT family of operating systems support the file system concept of reparse points, which allow you to load a drive into an empty directory. These are really useful for, say, adding space to an existing drive without repartitioning, or, as I like, for limiting the total size of a given directory into which downloads are retrieved automatically. Naturally, the intention is to load a *different* drive into the reparse directory, for example, loading *T:* into *H:\temp*. However, it's also possible to load a given drive into its own subdirectory, for example, to load *H:* into *H:\infinite*. This creates an infinite file system tree. Now, to be sure, such things are an abuse of the file system, but since it's possible, you must treat it as a legitimate runtime condition and test for it. The longhand version, therefore, is buggy. The shorthand version skips any directories that are reparse points and so is not.

20.3 Sequence Design

It's time to consider the features of the sequence that are implied by the **FindFirstFile**/**FindNextFile** API. First, we want to support both multibyte and wide string compilation, so the sequence will be a template, `winstl::basic_findfile_sequence`. The character-encoding-specific aspects will be abstracted into traits, just as is done in the standard library with class templates such as `basic_string`, `basic_ostream`, and so on. In this case, we'll be using `winstl::filesystem_traits` (Section 16.3). Hence the sequence template will be of the following form:

```
template< typename C
        , typename T = filesystem_traits<C>
        >
class basic_findfile_sequence;
```

As I mentioned earlier, the **FindFirstFile**/**FindNextFile** API is most like the **opendir**/**readdir** API in form and general semantics: It retrieves one element at a time from the underlying file system. This means we can expect the sequence to support input iteration, in the form of an iterator class. Because of inadequacies in some compilers that eventuated early in the life of this component, the iterator class template could not be implemented as a nested class, so it is a stand-alone class template, playfully entitled `basic_findfile_sequence_const_iterator`. It uses a reference-counted shared handle class similar to that described for `readdir_sequence::const_iterator` (Sections 19.3.5 and 19.3.7) for managing the underlying operating system search handle, to properly support the requirements of the input iterator.

The finer details of the two APIs differ, and this is reflected in the differences between the two sequences. The most obvious difference is that the `WIN32_FIND_DATA` contains additional characteristics of the enumerated entry. It would be crazy to throw this information away since it would only have to be retrieved by a subsequent call to `FindFirstFile()`. Coupled with the fact that the structure contains only the file name, rather than the full path, this implies that a custom value type is necessary. This is in the form of another stand-alone class template, its name another ode to succinctness: `basic_findfile_sequence_value_type`.

Since wildcards are supported, albeit in a limited form, we'd be foolish not to support them. Hence, the constructors are likely to be like those of `glob_sequence`, allowing users to specify pattern, search directory, and flags. But since there is no multipart pattern API in Windows, we need to add functionality to the sequence to provide this service. We'll follow the Windows shell standard convention of a semicolon as path separator, as in *`*.cpp;*h`*.

Other minor differences impact the implementation, rather than the design, such as the fact that Windows APIs can work with slashes and backslashes, or any mix of the two, for example, *`H:\Publishing/Books/`*.

20.4 `winstl::basic_findfile_sequence`

Let's begin by looking at the collection class template, `basic_findfile_sequence`.

20.4.1 Class Interface

The general form of `basic_findfile_sequence` (defined in the `winstl` namespace) follows that of `readdir_sequence`. Listing 20.4 shows the member types and member constants. Save for the use of `basic_findfile_sequence_value_type` as the `value_type`, it's as we would expect, given our previous experience.

Listing 20.4 Member Types and Constants

```
// In namespace winstl
template <typename C, typename T>
class basic_findfile_sequence_value_type;

template <typename C, typename T, typename V>
```

```
class basic_findfile_sequence_const_iterator;

template< typename C
        , typename T = filesystem_traits<C>
        >
class basic_findfile_sequence
{
public: // Member Types
  typedef C                                           char_type;
  typedef T                                           traits_type;
  typedef basic_findfile_sequence<C, T>               class_type;
  typedef basic_findfile_sequence_value_type<C, T>    value_type;
  typedef basic_findfile_sequence_const_iterator<C, T, value_type>
                                                      const_iterator;
  typedef value_type const                            reference;
  typedef value_type const                            const_reference;
  typedef typename traits_type::find_data_type        find_data_type;
  typedef ptrdiff_t                                   difference_type;
  typedef size_t                                      size_type;
  typedef int                                         flags_type;
public: // Member Constants
  enum search_flags
  {
      includeDots     =   0x0008
    , directories     =   0x0010
    , files           =   0x0020
    , skipReparseDirs =   0x0100
  };
  . . .
```

There are four constructors, shown in Listing 20.5, providing a range of initialization possibilities. They're more like `glob_sequence` than `readdir_sequence`.

Listing 20.5 Constructors and Destructor

```
public: // Construction
  explicit basic_findfile_sequence( char_type const* pattern
                                  , flags_type flags = directories | files);
  basic_findfile_sequence(  char_type const* patterns
                          , char_type        delim
                          , flags_type flags = directories | files);
  basic_findfile_sequence(  char_type const* directory
                          , char_type const* pattern
                          , flags_type flags = directories | files);
  basic_findfile_sequence(  char_type const* directory
                          , char_type const* patterns
                          , char_type        delim
                          , flags_type flags = directories | files);
  ~basic_findfile_sequence() throw();
```

The constructors support all of the following uses:

```
findfile_sequence   ffs1(_T("*.*"));
findfile_sequence_a ffs2("*.*", findfile_sequence_a::skipReparseDirs);
findfile_sequence_w ffs3(L"*.cpp|makefile.???", L'|');
findfile_sequence   ffs1( _T("h:/freelibs"), _T("*.h;*.hpp"), ';'
                        , findfile_sequence::files);
```

The other public methods are shown in Listing 20.6. Naturally, there's the `begin()/end()` pair. As with `readdir_sequence`, an `empty()` method is provided, but a `size()` method is not because it would require a full, and potentially expensive, iteration of the whole range of elements and may not be repeatable. The `get_directory()` method provides access to the directory specified to the constructor (or the current directory, at the time of construction, for those constructors not taking a `directory` parameter), after it has been validated, which we'll cover in a moment.

Listing 20.6 `basic_findfile_sequence` **Iteration, Attributes, and State Methods**

```
public: // Iteration
  const_iterator      begin() const;
  const_iterator      end() const;
public: // Attributes
  char_type const*    get_directory() const;
public: // State
  bool                empty() const;
```

The private worker methods (Listing 20.7) comprise the class invariant (Chapter 7) method `is_valid()` and the methods `validate_flags_()` and `validate_directory_()`. `validate_flags_()` performs the same functions as `glob_sequence` and `readdir_sequence`. The purpose of `validate_directory_()` is to ensure that the directory has a full path and has a trailing path name separator. It takes account of compilation absent exception support, so we'll look at its implementation in Section 20.4.4 where that issue is covered.

Listing 20.7 Invariant and Implementation Methods

```
private: // Invariant
  bool is_valid() const;
private: // Implementation
  static flags_type validate_flags_(flags_type flags);
  static void       validate_directory_(char_type const* directory
                                       , file_path_buffer_type_& dir);
```

The member variables are shown in Listing 20.8. The sequence holds the search directory, the patterns, the delimiter, and the flags. `m_directory` is a specialization of `file_path_buffer` (Section 16.4), since it is a path. `m_patterns` is a small-internal-buffer specialization of `auto_buffer`: It can be any length but is likely to be small in most cases.

Listing 20.8 Member Variables

```
private: // Member Variables
  typedef basic_file_path_buffer<char_type>   file_path_buffer_type_;
  typedef stlsoft::auto_buffer<char_type, 32> patterns_buffer_type_;
  const char_type          m_delim;
  const flags_type         m_flags;
  file_path_buffer_type_   m_directory;  // Directory used by ctor
  patterns_buffer_type_    m_patterns;   // Pattern(s) given to ctor
```

The class definition is rounded out by the usual proscription of copying methods (not shown; see Section 19.3).

20.4.2 Construction

All four constructors do pretty much the same thing. `m_flags` is initialized via `validate_flags_()`. The `pattern(s)` parameter is copied into `m_patterns`. The directory is calculated via `validate_directory_()`, which writes it into `m_directory`. Listing 20.9 shows only the four-parameter overload. Where the others do not specify a `directory` parameter, substitute *NULL*, and where they do not specify a `delim` parameter, substitute `char_type()` (or `'\0'`, in other words).

Listing 20.9 Four-Parameter Constructor of the Sequence

```
template <typename C, typename T>
basic_findfile_sequence<C, T>::basic_findfile_sequence(
                                              char_type const* directory
                                            , char_type const* patterns
                                            , char_type        delim
                                            , flags_type       flags)
  : m_delim(delim)
  , m_flags(validate_flags_(flags))
  , m_patterns(1 + traits_type::str_len(patterns))
{
  validate_directory_(directory, m_directory);
  traits_type::str_n_copy(&m_patterns[0], patterns
                                           , m_patterns.size());
  WINSTL_ASSERT(is_valid());
}
```

20.4.3 Iteration

The iteration methods are very straightforward, as shown in Listing 20.10.

Listing 20.10 Iteration Methods

```
template <typename C, typename T>
const_iterator basic_findfile_sequence<C, T>::begin() const
{
  WINSTL_ASSERT(is_valid());
  return const_iterator(*this, m_patterns.data(), m_delim, m_flags);
}
template <typename C, typename T>
const_iterator basic_findfile_sequence<C, T>::end() const
{
  WINSTL_ASSERT(is_valid());
  return const_iterator(*this);
}
```

begin() returns an iterator instance constructed from the sequence reference, the pattern(s), the delimiter, and the flags. Ostensibly, end() returns a default-constructed instance. For debugging purposes, however, the endpoint iterator takes a back reference to the sequence, in order to catch attempted comparisons between iterator instances retrieved from different sequence instances.

Tip: Consider maintaining back references to sequence instances for endpoint iterators to detect comparison of iterators elicited from different sequence instances, but take care to ensure that they do not break comparison semantics with default-constructed instances.

Note that this is another reason to prefer sequences over stand-alone iterators.

20.4.4 Exception Agnosticism

validate_directory_() must convert the given directory to a full path and must ensure that a trailing separator exists. To do the latter is simply a case of using filesystem_traits::ensure_dir_end(). The former is achieved via get_full_path_name(), and an exception is thrown if that fails. The implementation is shown in Listing 20.11.

Listing 20.11 Implementation of validate_directory_() Worker Method

```
template <typename C, typename T>
void basic_findfile_sequence<C, T>::validate_directory_(
                                        char_type const* directory
                                      , file_path_buffer_type_& dir)
{
  if( NULL == directory ||
      '\0' == *directory)
  {
    static const char_type    s_cwd[] = { '.', '\0' };
    directory = &s_cwd[0];
  }
  if(0 == traits_type::get_full_path_name(directory, dir.size()
                                                   , &dir[0]))
  {
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
    throw filesystem_exception(::GetLastError());
#else /* ? STLSOFT_CF_EXCEPTION_SUPPORT */
    dir[0] = '\0';
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
```

```
  }
  else
  {
    traits_type::ensure_dir_end(&dir[0]);
  }
}
```

Once again, a local static character string is used for the directory. In this case, however, we cannot initialize it with the literal string `"."` because that would be a compilation error when `char_type` is `wchar_t`. Instead, it is initialized as an array of two characters, `'.'` and `'\0'`. Because the nul character and the `'.'` character have the same code point in character-encoding schemes whether `char` or `wchar_t`, this is a straightforward and effective way to get string literals for simple strings such as `"."`.

Tip: Create character-encoding-independent string literals (containing code points within the 0x00-0x7F range) as arrays of `char_type`, initialized using array syntax.

For historical reasons, the `basic_findfile_sequence` component must work correctly absent exception support. In that context, if `get_full_path_name()` fails, it sets `dir[0]` to `'\0'`. This is then detected in `begin()` and returns the endpoint iterator as a consequence, as shown in Listing 20.12. Thus, enumeration of an invalid directory in compilations absent exception support is well founded.

Listing 20.12 `begin()` **Method Handling Compilation Absent Exception Support**

```
template <typename C, typename T>
const_iterator basic_findfile_sequence<C, T>::begin() const
{
  WINSTL_ASSERT(is_valid());
#ifndef STLSOFT_CF_EXCEPTION_SUPPORT
  if('\0' == m_directory[0])
  {
    ::SetLastError(ERROR_INVALID_NAME);
    return const_iterator(*this);
  }
#endif /* !STLSOFT_CF_EXCEPTION_SUPPORT */
  return const_iterator(*this, m_patterns.data(), m_delim, m_flags);
}
```

This is also reflected in the invariant, which is shown in Listing 20.13.

Listing 20.13 Invariant Method

```
template <typename C, typename T>
bool basic_findfile_sequence<C, T>::is_valid() const
{
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
  if('\0' == m_directory[0])
  {
# ifdef STLSOFT_UNITTEST
    unittest::fprintf(err, "dir empty when exception handling on\n");
# endif /* STLSOFT_UNITTEST */
    return false;
  }
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
  if( '\0' != m_directory[0] &&
    !traits_type::has_dir_end(m_directory.c_str()))
  {
#ifdef STLSOFT_UNITTEST
    unittest::fprintf(unittest::err, "m_directory not empty and does not
have a trailing path name separator; m_directory=%s\n", m_directory.c_
str());
#endif /* STLSOFT_UNITTEST */
    return false;
  }
  return true;
}
```

Tip: Try to ensure that your components have well-defined behavior when compiled absent support for exception handling. If this cannot be achieved, ensure that you use `#error` to prevent compilation, rather than silently generating unsafe code.

20.5 winstl::basic_findfile_sequence_const_iterator

Listing 20.14 shows the class definition of the `basic_findfile_sequence_const_iterator`. There are no surprises in its public interface. (They're waiting for us in the implementation.) The iterator category is input iterator. The element reference category is *by-value temporary*. The `shared_handle` nested class is virtually identical to that of `readdir_sequence`, other than that a "null" value is *INVALID_HANDLE_VALUE* rather than *NULL*, and the handle is released by `FindClose()`, so its definition is not shown.

Listing 20.14 Definition of `basic_findfile_sequence_const_iterator`

```
// In namespace winstl
template< typename C  // Character type
        , typename T  // Traits type
        , typename V  // Value type
        >
class basic_findfile_sequence_const_iterator
  : public std::iterator< std::input_iterator_tag
                        , V, ptrdiff_t
                        , void, V     // BVT
                        >
{
private: // Member Types
  typedef basic_findfile_sequence<C, T>               sequence_type;
public:
  typedef C                                              char_type;
  typedef T                                              traits_type;
  typedef V                                              value_type;
  typedef basic_findfile_sequence_const_iterator<C, T, V> class_type;
  typedef typename traits_type::find_data_type           find_data_type;
  typedef typename sequence_type::size_type              size_type;
private:
  typedef typename sequence_type::flags_type             flags_type;
private: // Construction
  basic_findfile_sequence_const_iterator( sequence_type const&  seq
                                    , char_type const*      patterns
                                    , char_type             delim
                                    , flags_type            flags);
  basic_findfile_sequence_const_iterator(sequence_type const& seq);
public:
  basic_findfile_sequence_const_iterator();
  basic_findfile_sequence_const_iterator(class_type const& rhs);
  ~basic_findfile_sequence_const_iterator() throw();
  class_type& operator =(class_type const& rhs);
public: // Input Iterator Methods
  class_type&       operator ++();
  class_type        operator ++(int); // Canonical implementation
  const value_type  operator *() const;
  bool              equal(class_type const& rhs) const;
private: // Implementation
  static HANDLE find_first_file_( char_type const*  spec
                               , flags_type        flags
                               , find_data_type*   findData);
private: // Utility Classes
  struct shared_handle
  {
    . . .
  };
private: // Member Variables
  friend class basic_findfile_sequence<C, T>;
  typedef basic_file_path_buffer<char_type>   file_path_buffer_type_;
  sequence_type const*                   m_sequence;
  shared_handle*                         m_handle;
  typename traits_type::find_data_type  m_data;
  file_path_buffer_type_                 m_subPath;
  size_type                              m_subPathLen;
  char_type const*                       m_pattern0;
  char_type const*                       m_pattern1;
  char_type                              m_delim;
  flags_type                             m_flags;
};
```

m_sequence is the back pointer to the sequence instance. (It has to be a pointer because references cannot be reassigned.) m_handle is a pointer to the shared_handle shared context instance. m_data is the instance of WIN32_FIND_DATA, which stores the information for the currently iterated entry. m_delim and m_flags are the values held in the sequence instance. The remaining four members, m_subPath, m_subPathLen, m_pattern0, and m_pattern1 are used in the processing of the pattern(s) and enumerated entries in the preincrement operator (Section 20.5.3).

20.5.1 Construction

As shown in Listing 20.14, there are four constructors of the iterator class, in addition to the public copy assignment operator and the destructor. The implementation of the four-parameter conversion constructor is shown in Listing 20.15. It calls operator ++() to move the iterator instance to the first available entry (or to the endpoint). Note that m_pattern0 and m_pattern1 are both initialized to the patterns parameter, which is actually the m_patterns member of the sequence class; this will be explained in Section 20.5.3.

Listing 20.15 Four-Parameter Constructor of the Iterator

```
template <typename C, typename T, typename V>
basic_findfile_sequence_const_iterator<C, T, V>::
  basic_findfile_sequence_const_iterator( sequence_type const&  seq
                                        , char_type const*      patterns
                                        , char_type             delim
                                        , flags_type            flags)
  : m_sequence(&seq)
  , m_handle(NULL)
  , m_subPath()
  , m_subPathLen(0)
  , m_pattern0(patterns)
  , m_pattern1(patterns)
  , m_delim(delim)
  , m_flags(flags)
{
  m_subPath[0] = '\0';
  operator ++();
}
```

The copy assignment operator is shown in Listing 20.16. Note the use of the local variable prev_handle and its use in delayed release of the original shared handle context. This avoids the problem whereby an iterator is self-assigned when its reference count is *1*. The natural tendency is to release first, but in that specific case the shared context object would be destroyed before the second reference was added.

Listing 20.16 Copy Assignment Operator

```
template <typename C, typename T, typename V>
class_type&
 basic_findfile_sequence_const_iterator<C, T, V>::
  operator =(class_type const& rhs)
{
  WINSTL_MESSAGE_ASSERT("Assigning iterators from different sequences"
    , m_sequence == NULL || rhs.m_sequence == NULL || rhs.m_sequence);
  shared_handle*  prev_handle   = m_handle;
                  m_handle      = rhs.m_handle;
                  m_data        = rhs.m_data;
                  m_subPath     = rhs.m_subPath;
                  m_subPathLen  = rhs.m_subPathLen;
                  m_pattern0    = rhs.m_pattern0;
                  m_pattern1    = rhs.m_pattern1;
                  m_delim       = rhs.m_delim;
                  m_flags       = rhs.m_flags;
  if(NULL != m_handle)
  {
    m_handle->AddRef();
  }
  if(NULL != prev_handle)
  {
    prev_handle->Release();
  }
  return *this;
}
```

Such cases are obviated by the use of reference-counting smart pointers, such as the **STLSoft** class template `ref_ptr`. The reason I've not done so in these implementations is that many of the sequence classes—`readdir_sequence`, `basic_findfile_sequence`, `basic_findvolume_sequence`, and so on—were written before the `ref_ptr` was migrated into **STLSoft** from my company's proprietary library (where it was known less succinctly as `ReleaseInterface`). If I were writing the collection class from scratch now, I'd likely use `ref_ptr`. As it is, it draws the reference-counting issues to your attention.

The implementations of the other construction methods are equivalent to those of `readdir_sequence` and for brevity are not shown here.

20.5.2 `find_first_file_()`

Before we get into the preincrement operator, which is the biggest of this class by some margin, I want to discuss one of the worker functions that it uses, `find_first_file_()` (Listing 20.17).

Listing 20.17 Implementation of the `find_first_file_()` Worker Method

```
template <typename C, typename T, typename V>
HANDLE
 basic_findfile_sequence_const_iterator<C, T, V>::find_first_file_(
                              char_type const*  searchSpec
                            , flags_type        flags
                            , find_data_type*   findData)
{
  HANDLE  hSrch = INVALID_HANDLE_VALUE;
  enum
  {
#ifdef FILE_ATTRIBUTE_REPARSE_POINT
    reparsePointConstant    =   FILE_ATTRIBUTE_REPARSE_POINT
#else /* ? FILE_ATTRIBUTE_REPARSE_POINT */
    reparsePointConstant    =   0x00000400
#endif /* FILE_ATTRIBUTE_REPARSE_POINT */
  };
#if defined(_WIN32_WINNT) && \
    _WIN32_WINNT >= 0x0400
  if( (directories == (flags & (directories | files))) &&
      system_version::winnt() &&
      system_version::major() >= 4)
  {
    hSrch = traits_type::find_first_file_ex(searchSpec
                        , FindExSearchLimitToDirectories, findData);
  }
  else
#endif /* _WIN32_WINNT >= 0x0400 */
  if(INVALID_HANDLE_VALUE == hSrch)
  {
    hSrch = traits_type::find_first_file(searchSpec, findData);
  }
  for(; INVALID_HANDLE_VALUE != hSrch; )
  {
    if(traits_type::is_file(findData))
    {
      if(flags & sequence_type::files)
      {
        break;
      }
    }
    else
    {
      if(traits_type::is_dots(findData->cFileName))
      {
        if(flags & sequence_type::includeDots)
        {
          break;
        }
      }
      else if(flags & sequence_type::directories)
      {
        if( 0 == (flags & sequence_type::skipReparseDirs) ||
            0 == (findData->dwFileAttributes & reparsePointConstant))
        {
          break; // Not skipping reparse pts, or not reparse pt
        }
      }
    }
    if(!traits_type::find_next_file(hSrch, findData))
    {
      ::FindClose(hSrch);
      hSrch = INVALID_HANDLE_VALUE;
      break;
    }
  }
  return hSrch;
}
```

The purpose of this function is to call the appropriate `FindFirstFile()` function and have the first returned item correspond to the given flags. In order to apply the filtering, it must synthesize a local constant to be used for testing reparse points. This is because the constant *`FILE_ATTRIBUTE_REPARSE_POINT`* is not present in all compilers' Windows header files. (It's done as an enumerator, rather than a constant integer, to put off a number of compilers' whinges about unused variables, linker errors, and other meaningless gripes.)

The other notable feature is the conditional testing of flags and invocation of the `FindFirstFileEx()` function (via `traits_type::find_first_file_ex()`). This function is provided on all operating systems from the NT family from NT 4 onward but is not available on any Windows 9x system. It is in the form of a pair of `A`/`W` functions, whose ostensible character-encoding-independent signature is as follows:

```
HANDLE FindFirstFileEx( LPCTSTR             fileName
                      , FINDEX_INFO_LEVELS  infoLevelId */
                      , void*               findFileData */
                      , FINDEX_SEARCH_OPS   searchOp
                      , void*               searchFilter
                      , DWORD               additionalFlags);
```

The `searchOp` parameter determines the kind of searching that will be carried out. The `FindExSearchLimitToDirectories` flag instructs the function to return only directories, equivalent to the **glob** API's *`GLOB_ONLYDIR`*. However, like *`GLOB_ONLYDIR`*, this is only an advisory; not all file systems can comply. But it is included in the hope that those file systems that do support it can facilitate a reduction in the number of system calls required to search for a set of directories.

When you've read later chapters and seen the dynamic library function invocation library `dl_call()` used to invoke functionality that may not exist on a given system, you might wonder why it is not called in this case. The reason is that absence of the function is not terminal to the functioning of the sequence, and the repeated costs of loading the library or the catching of load-failure exceptions would have a negative impact on performance. Alternatively, the function could be loaded and stored by the sequence instance, but that's adding too much complexity; I prefer to just go with static linking in the case where the user has defined `_WIN32_WINNT` appropriately. This symbol is used throughout the Windows headers to indicate specific NT-only compilation, so I think that piggybacking on such user-directed choices is a reasonable and practicable option.

Tip: Consider using specific, and not universally available, API functions in conditions where availability has already been selected by explicit user choice.

20.5.3 `operator ++()`

I can't hide it from you: This method is a titan. There are several reasons for this.

- The multipart patterns must be parsed and processed serially.
- Windows accepts slashes as path name separators, as well as its native backslashes.
- Special case treatment of patterns of dots directories is required.
- Entries must be filtered.
- The search directory must be joined with each search pattern before passing to `find_first_file()`.
- The subpath for the iterator must be formed.

Together these actions constitute a nontrivial block of code, so I will show them in separate segments.

Before I describe the implementation, I need to introduce you to a utility function template, `stlsoft::find_next_token()`, which implements restartable parsing semantics. It has two overloads, with the following signatures:

```
template <typename C>
C const* find_next_token( C const*&      p0
                        , C const*&      p1
                        , C const* const end
                        , C              delim);

template <typename C>
bool find_next_token(C const*& p0, C const*& p1, C delim);
```

The first parses a string, with the given delimiter, up to the `end` point. The second parses until the nul-terminating character is found. Both take two references to caller-supplied pointers, in which the parsing state is maintained. Parsing begins by setting the two pointers to the same point (the start) in the string and then calling `find_next_token()` until they indicate the parsing is ended, by returning *end* or *false*, respectively. Each parsed token is represented by the string slice defined as {`p1 - p0`, `p0`}, as shown in the following example, which prints out *[][*.zip][*.html][][*.exe][][*.pdf]*.

```
static const char patterns[] = "||*.zip|*.html||*.exe||*.pdf|";
char const* p0  = &patterns[0];
char const* p1  = &patterns[0];
while(stlsoft::find_next_token(p0, p1, '|'))
{
  ::printf("[%.*s]", p1 - p0, p0);
}
```

Because it does not need to perform any allocation or copying, it is exceedingly fast (see Section 27.9, where we look at string tokenization in a lot more detail). The downside is that it works only with single-character delimiters, and since the returned tokens are not nul-terminated strings, they can be difficult to work with. Note that it does not elide blanks, but that is easy to do in the client code, as in the following:

```
. . .
while(stlsoft::find_next_token(p0, p1, '|'))
{
  if(p1 != p0)
  {
    ::printf("[%.*s]", p1 - p0, p0);
  }
}
```

This now prints *[*.zip][*.html][*.exe][*.pdf]*.

Now that that's settled, let's get on with the main course. I'm going to first show the structure of the method, in Listing 20.18, and then show the definition of each part.

Listing 20.18 Preincrement Method

```
template <typename C, typename T, typename V>
class_type&
 basic_findfile_sequence_const_iterator<C, T, V>::operator ++()
{
  WINSTL_MESSAGE_ASSERT("Attempting to increment invalid iterator!"
                       , '\0' != *m_pattern0);
  WINSTL_ASSERT(NULL != m_pattern0);
  WINSTL_ASSERT(NULL != m_pattern1);
  enum
  {
#ifdef FILE_ATTRIBUTE_REPARSE_POINT
    reparsePointConstant    =   FILE_ATTRIBUTE_REPARSE_POINT
#else /* ? FILE_ATTRIBUTE_REPARSE_POINT */
    reparsePointConstant    =   0x00000400
#endif /* FILE_ATTRIBUTE_REPARSE_POINT */
  };

  for(; '\0' != *m_pattern0 || '\0' != *m_pattern1; )
  {
    if(NULL == m_handle)
    {
      while(stlsoft::find_next_token(m_pattern0,m_pattern1, m_delim))
      {
        WINSTL_ASSERT(m_pattern0 <= m_pattern1);
        if(m_pattern1 != m_pattern0)
        {
          . . . // 1. Form search pattern from search dir & token.
          . . . // 2. Determine subpath of current enumeration.
          . . . // 3. Invoke find_first_file_().
          . . . // 4. Handle special case of explicit dots dirs.
        }
      }
    }
    if(NULL != m_handle)
    {
      for(; INVALID_HANDLE_VALUE != m_handle->hSrch; )
      {
        . . . // 5. Filter entries, according to flags.
      }
    }
  }
  return *this;
}
```

Hopefully it's reasonably clear what's going on here. The outermost `for` loop ensures processing until the pattern(s) have been fully exhausted. Inside this loop there is a test of `m_handle`. If it's *NULL*, it means that there's no currently active pattern, that is, the iterator is not currently enumerating file system contents based on a parsed pattern. This can be because it's the first time `operator ++()` has been called, or it might be that all elements matching the previous pattern have been found, and the next one needs to be processed. Whatever the reason, the next token is retrieved inside the `while` loop, with blanks elided by the `if` statement. (Now it's clear why the default value of `m_delim` is *'\0'* since that is the nul-terminator that `find_next_token()` already looks for to terminate processing. Since it is ignored, there is effectively no parsing, thus no multipattern support when those constructors are used. *QED*.)

Once a nonblank pattern has been found, the search pattern is formed from the current pattern token and the search directory, as shown in Listing 20.19.

Listing 20.19 Search Pattern Formation

```
. . .
// 1. Form search pattern from search dir & token
file_path_buffer_type search; // Buff to prepare srch ptrn
size_type             cch;    // Speeds up str_??() ops
if(traits_type::is_path_rooted(m_pattern0))
{
  search[0] = '\0';
  cch       = 0;
}
else
{
  traits_type::str_copy(&search[0], m_sequence->get_directory());
  cch = traits_type::str_len(&search[0]);
  --cch; // Directory is always trailing a path name separator
  traits_type::ensure_dir_end(&search[(cch > 1) ? (cch - 2) : 0]);
}
traits_type::str_n_cat( &search[0] + cch, m_pattern0
                      , m_pattern1 - m_pattern0);
. . .
```

The test for `is_path_rooted()` is for the case where the user might construct a sequence instance as follows:

```
findfile_sequence  files("D:\\Dev", "*.cpp|h:/abs/olute.txt", '|');
```

In such a case, we would not want the second search pattern formed as "*D:\Dev\h:/abs/olute.txt*". In other cases, the directory is retrieved from the sequence class via the `m_sequence` back pointer and copied into `search`. The length is taken here and used throughout the remainder of the block, to avoid having to effect all string operations from the start of the string—this may be a modest efficiency, but I think it's worth doing if we can avoid otherwise completely pointless processing. It's decremented, so that `ensure_dir_end()` does the correct thing in all cases, and finally the current pattern token is appended.

Tip: Consider caching a reliable minimum length to avoid unnecessary costs when manipulating nul-terminated strings using standard C library functions.

The next section of the method calculates the search subpath, taking into account the issue of the schizophrenic support of the Windows API file system functions for both slashes and backslashes. It is shown in Listing 20.20.

Listing 20.20 Slash and Backslash Handling

```
  . . .
  // 2. Determine subpath of current enumeration
  char_type const* slash;  // Have to declare these to . . .
  char_type const* bslash; // . . . fit in book. ;-)
  slash = traits_type::str_rchr(&search[0] + cch, '/');
  bslash = traits_type::str_rchr(&search[0] + cch, '\\');
  WINSTL_ASSERT(!traits_type::is_path_rooted(m_pattern0) || ((NULL !=
slash) || (NULL != bslash)));
  if( NULL != slash &&
      slash >= m_pattern1)
  {
    slash = NULL;
  }
  if( NULL != bslash &&
      bslash >= m_pattern1)
  {
    bslash = NULL;
  }
  if( NULL == slash &&
      NULL == bslash)
  {
    m_subPath[0] = '\0';
    m_subPathLen = 0;
  }
  else
  {
    if(NULL == slash)
    {
      slash = bslash;
    }
    else if(NULL != bslash &&
            slash < bslash)
    {
      slash = bslash;
    }
    const size_t n = static_cast<size_t>(slash - &search[0]);
    traits_type::str_n_copy(&m_subPath[0], &search[0], n);
    m_subPathLen = n;
    m_subPath[n] = '\0';
  }
  . . .
```

This code finds the last slash or backslash, which is used to calculate how much of the full search pattern is to be copied into the `m_subPath` member. Note the continued use of `cch` in the calls to `traits_type::str_rchr()`, to minimize the amount of the string searched. The reason we need to calculate the subpath, which will later be combined with the

`WIN32_FIND_DATA::cFileName` member to form the entry's full path, is to handle search criteria such as the following:

```
findfile_sequence  files( "H:/freelibs/shwild/current"
                        , "include\shwild\*.h*;src\*.h*", ';');
```

The search patterns here include subdirectories. Without specific subpath calculation, the returned entry paths would be wrong. Imagine that the directory *H:\freelibs\shwild\current\include\shwild* contains the file *shwild.hpp*. If subpath calculation was not performed, the returned path for this entry could be erroneously calculated as *H:\freelibs\shwild\current\shwild.hpp*. (This is a lesson learned from real experience, an embarrassing length of time after I considered `findfile_sequence` to be well tested and fully functional!)

The next step is to invoke `find_first_file_()`, followed by special case handling of explicit dots directories and creation of the shared handle instance, as shown in Listing 20.21.

Listing 20.21 Retrieval of Entries and Creation of Shared Context

```
. . .
// 3. & 4. Invoke find_first_file_(), and dots dirs
HANDLE hSrch = find_first_file_(search.c_str(), m_flags, &m_data);
if(INVALID_HANDLE_VALUE != hSrch)
{
  stlsoft::scoped_handle<HANDLE>  cleanup( hSrch, ::FindClose
                                          , INVALID_HANDLE_VALUE);
  if( '.' == m_pattern0[0] &&
      ( m_pattern1 == m_pattern0 + 1 ||
        ( '.' == m_pattern0[1] &&
          m_pattern1 == m_pattern0 + 2)))
  {
    const size_t n = static_cast<size_t>(m_pattern1 - m_pattern0);
    traits_type::str_n_copy(&m_data.cFileName[0], m_pattern0, n);
    m_data.cFileName[n] = '\0';
  }
  m_handle = new shared_handle(hSrch);
  if(NULL != m_handle)
  {
    cleanup.detach();
  }
  return *this;
}
. . .
```

The search handle is given to an instance of `scoped_handle` (Section 16.5), which will automatically call `FindClose()` on it if any exceptions are thrown.

The special case testing is to ensure that when the user specifies `"."` or `".."` as a search pattern, the corresponding entry has that as its name, rather than the actual name of the corresponding directory. This is no right or wrong issue; I just prefer to use this way. You might choose to do otherwise.

All that remains is to create a `shared_handle` instance, ensuring that the resource is released if that fails. The test against *NULL* is to ensure correct behavior regardless of whether or not exception support is enabled. If all is well, `scoped_handle::detach()` is invoked, to release the search handle into the care of the `shared_handle` instance.

The final part of this method is the filtering, as shown in Listing 20.22. It is logically the same as that performed in `find_first_file_()`, the difference being that the `shared_handle` is released and set to *NULL* when the (current) search is exhausted.

Listing 20.22 Filtering of Elements

```
. . .
// 5. Filter entries, according to flags.
if(NULL != m_handle)
{
  for(; INVALID_HANDLE_VALUE != m_handle->hSrch; )
  {
    if(!traits_type::find_next_file(m_handle->hSrch, &m_data))
    {
      m_handle->Release();
      m_handle = NULL;
      break;
    }
    else
    {
      if(traits_type::is_file(&m_data))
      {
        if(m_flags & sequence_type::files)
        {
          return *this;
        }
      }
      else
      {
        if(traits_type::is_dots(m_data.cFileName))
        {
          if(m_flags & sequence_type::includeDots)
          {
            return *this;
          }
        }
        else if(m_flags & sequence_type::directories)
        {
          if( 0 == (m_flags & sequence_type::skipReparseDirs) ||
            0 == (m_data.dwFileAttributes & reparsePointConstant))
          {
            return *this;
          }
        }
      }
    }
  }
}
```

And that's pretty much it for the iterator class. The dereference operator just passes the subpath and the `m_data` member to the value type constructor, and the `equal()` method simply determines whether the two instances have the same `m_handle` member.

One feature to note in the implementation is that a failed call to `find_first_file_()` is treated as "no elements found." There is no additional testing to determine whether this is genuinely the case or whether some operating-system-specific condition has prevented the search. I just didn't feel that it was worth the extra load on the user of the sequence to take into account the various conditions that do so, and no user has ever complained about this aspect. Again, this is something you might wish to do differently. Indeed, the `inetstl::basic_findfile_sequence` class template, which shares much with the **WinSTL** version, does make one such test, which we'll discuss in the next intermezzo, Chapter 21.

20.6 winstl::basic_findfile_sequence_value_type

In essence, the value type is a full path and a `WIN32_FIND_DATA` instance, with a friendly interface, in the form of the `basic_findfile_sequence_value_type` class template. The class definition is shown in Listing 20.23, along with the implementations of four methods.

Listing 20.23 Definition of `basic_findfile_sequence_value_type`

```
template<typename C, typename T>
class basic_findfile_sequence_value_type
{
private: // Member Types
  typedef basic_findfile_sequence<C, T>           sequence_type;
  typedef typename sequence_type::flags_type      flags_type;
public:
  typedef C                                       char_type;
  typedef T                                       traits_type;
  typedef basic_findfile_sequence_value_type<C, T> class_type;
  typedef typename traits_type::find_data_type    find_data_type;
  typedef typename sequence_type::size_type       size_type;
private: // Construction
  basic_findfile_sequence_value_type( find_data_type const& data
                                    , char_type const*  directory
                                    , size_type         cchDirectory)
    : m_data(data)
  {
    traits_type::str_n_copy(&m_path[0], directory, cchDirectory);
    traits_type::ensure_dir_end(&m_path[0]);
    traits_type::str_cat(&m_path[0] + cchDirectory, data.cFileName);
  }
public:
  basic_findfile_sequence_value_type();
  class_type& operator =(class_type const& rhs);
public: // Attributes
  find_data_type const& get_find_data() const;
  char_type const*      get_filename() const;
  char_type const*      get_short_filename() const
  {
    return '\0' != m_data.cAlternateFileName[0]
              ? m_data.cAlternateFileName : m_data.cFileName;
  }
  char_type const*      get_path() const;
  char_type const*      c_str() const;
  operator char_type const * () const;
  bool                  is_directory() const
  {
    return traits_type::is_directory(&m_data);
  }
  bool                  is_file() const;
  bool                  is_compressed() const;
#ifdef FILE_ATTRIBUTE_REPARSE_POINT
  bool                  is_reparse_point() const;
#endif /* FILE_ATTRIBUTE_REPARSE_POINT */
  bool                  is_read_only() const;
  bool                  is_system() const;
  bool                  is_hidden() const;
  bool                  equal(char_type const* rhs) const
  {
    return 0 == traits_type::str_compare_no_case( this->get_path()
                                                , rhs);
  }
  bool                  equal(class_type const& rhs) const;
private: // Member Variables
  friend class
    basic_findfile_sequence_const_iterator<C, T, class_type>;
  typedef basic_file_path_buffer<char_type> file_path_buffer_type_;
  find_data_type          m_data;
  file_path_buffer_type_  m_path;
};
```

The constructor copies the `WIN32_FIND_DATA` information into `m_data` and then builds the `m_path` member from the subpath and the `cFileName` structure member. All the `is_??()`

methods are implemented in terms of the corresponding methods of `filesystem_traits`, which use the `dwFileAttributes` member of `WIN32_FIND_DATA`.

Comparison is done via `equal()` methods. The one taking another class instance simply calls the `char_type const*` overload, which is where the work is done. The comparison is carried out in a case-independent manner since the Windows file systems do not take account of case.

Finally, the `get_short_filename()` method returns either the actual short file name, if any, or the "normal" name if not. Windows does a bit of a backward-compatibility shuffle with file names, giving ones that have names that do not correspond with the DOS naming convention an additional name. For example, the file name of my home directory is *`MATTY.SYNESIS`*, and the short name for this is *`MATTY~1.SYN`*. When the name corresponds to the conventions, the `cAlternateFileName` member is empty. This behavior can be a pain to work with, hence the logic in `get_short_filename()`.

Although we saw how reparse points are handled regardless of the specific headers used to compile the file, because that's runtime behavior, we go for simplicity in this class in not allowing `is_reparse_point()` to be seen if *`FILE_ATTRIBUTE_REPARSE_POINT`* is not defined. As well as being a sensible (and effort-saving) default action, this can also remind users that they may need to upgrade their header files.

There's an implicit conversion operator. This is a backward-compatible vestige of a former time when I was not embarrassed by writing such things. In this case, it hasn't been deprecated because it's harmless—the class does not have a corresponding implicit conversion constructor. However, such things are jejune and should be avoided in new designs, given that we know better alternatives exist. One such alternative is string access shims, and these are defined for the value type, as described in the next section.

20.7 Shims

In the motivating example (Section 20.2), I relied on string access shims for the sequence's value type. Listing 20.24 shows their definitions.

Listing 20.24 String Access Shims for `basic_findfile_sequence_value_type`

```
namespace stlsoft
{
  template <typename C, typename T>
  C const* c_str_data(
           winstl::basic_findfile_sequence_value_type<C, T> const& v)
  {
    return v.get_path();
  }
  template <typename C, typename T>
  size_t c_str_len(
           winstl::basic_findfile_sequence_value_type<C, T> const& v)
  {
    return ::stlsoft::c_str_len(v.get_path());
  }
  template <typename C, typename T>
  C const* c_str_ptr(
           winstl::basic_findfile_sequence_value_type<C, T> const& v)
  {
    return v.get_path();
  }
} // namespace stlsoft
```

String access shim aficionados know that there are also `_a` and `_w` variants of these three functions. Consult the implementation of `winstl::basic_findfile_sequence` in the **STLSoft** distribution (included on the CD) for the full skinny.

20.8 What, No Shims and Constructor Templates?

From the previous chapters (and much of the rest of the book), you'd be forgiven for thinking I'm shim-crazy, because I am. So it seems an obvious omission that `basic_findfile_sequence` does not use string access shims in its constructors. After all, the same benefits would apply to users of this class as to the others where shims are used. The reason they're not used is, alas, nothing more than oversight and poor planning on my part. We've already seen the kind of hurdles that we'd need to jump in order to solve the ambiguities of enumerators and constructor templates for `glob_sequence` in Chapter 18. The constructors of `basic_findfile_sequence` have a greater variety as they stand, and I've not (yet) devised a workable, unambiguous, backward-compatible upgrade of the class interface. So users must manually apply the tiresome `c_str()` and its ilk in order to use the sequence with types other than C-style strings. If any reader wishes to take this as a challenge and can work out a successful mechanism of applying string access shims to the sequence, I'd be very happy to learn about it.

20.9 Summary

Fundamentally, adapting the **FindFirstFile**/**FindNextFile** API to STL presents similar problems and has a solution similar to that for the **opendir**/**readdir** API.

- The iterator class must share state to fulfill single-pass semantics.
- There is no `size()` method. (This supports the *Principle of Least Surprise* and the *Principle of Most Surprise*.)
- Elision of dots directories and file/directory filtering is carried out by the component (the *Principle of Economy*).
- Encapsulation and exception safety are appropriately employed for robust interface and semantics.

However, the differences between the two APIs means there are specific differences in the design and implementation of `basic_findfile_sequence`.

- There is substantial demand on Windows for programming to the ANSI/multibyte *and* the Unicode/wide string variants of their APIs. Consequently, the STL collection (and its related types) must support both forms and is therefore a class template, supported by the `winstl::filesystem_traits` traits class (the *Principle of Diversity*).

- `WIN32_FIND_DATA` contains valuable `stat()` information, which we don't want to waste. In order to let user code have access to this *and* to allow full/absolute paths, a value type class is required.
- Because there's no Windows equivalent to the **glob** API, and `FindFirstFile()` takes a single-pattern wildcard search specification, we need to layer multipart pattern handling over the API. This requires use of pattern parsing and unavoidably complicates the definition of `operator ++()` (the *Principle of Economy*).
- Windows APIs can (mostly) handle both forward slashes and backslashes as path name separators, which complicates the manipulation of patterns (the *Principle of Economy*).
- Optional file system features, such as reparse points and support for directory-only searches (via `FindFirstFileEx(FindExSearchLimitToDirectories)`), must be catered for, without restricting the portability of the component (the *Principle of Economy*).

We have one further, small, file-system-related component to discuss, in the intermezzo following this chapter. Then we will branch out into issues of mathematical sequences, process management, string tokenization, windowing, and scatter/gather I/O over the remaining chapters in Part II to expand our coverage of the range of issues regarding STL collection adaptation.

20.10 File System Enumeration with recls: Coda

Just out of interest, I'd like to show you an alternative implementation of `ClearDirectory()` using **recls**, the recursive file system search library, which is implemented in terms of `unixstl::glob_sequence`, `unixstl::readdir_sequence`, and `winstl::basic_findfile_sequence`. With logging elided for brevity, the function reduces to the code shown in Listing 20.25.

Listing 20.25 Implementation of the Example Function Using the recls Library

```
void ClearDirectory(LPCTSTR lpszDir, LPCTSTR lpszFilePatterns)
{
  typedef recls::stl::basic_search_sequence<TCHAR>  ffs_t;
  ffs_t files(lpszDir, lpszFilePatterns, recls::FILES);
  { for(ffs_t::const_iterator b = files.begin(); b != files.end();
        ++b)
  {
    if(::DeleteFile((*b).c_str()))
    {
      ::SHChangeNotify(SHCNE_DELETE, SHCNF_PATH, (*b).c_str(), 0);
    }
  }}
}
```

CHAPTER 21

Intermezzo: When the Efficiency/Usability Balance Is Tipped: Enumerating FTP Server Directories

Giving money and power to government is like giving whiskey and car keys to teenage boys.

—P. J. O'Rourke

The Windows **WinInet** API presents an abstraction over Internet protocols, providing a suite of functions that simplify Internet programming of the HTTP, FTP, and Gopher protocols. Of particular interest to us, at this point in the book, are the `FtpFindFirstFile()`, `InternetFindNextFile()`, and `InternetClose()` functions, which provide a means of enumerating the contents of a directory on a remote FTP server. Conveniently, for Windows programmers at least, `FtpFindFirstFile()` and `InternetFindNextFile()` return information to the caller using the `WIN32_FIND_DATA` structure. Hence, we can write code such as that shown in Listing 21.1.

Listing 21.1 Enumeration of FTP Server Files Using WinInet API Functions

```
HINTERNET       hConnection = . . .
WIN32_FIND_DATA data;
HINTERNET       hFind = ::FtpFindFirstFile(hConnection, "*.*", &data
                                                           , 0, 0);
if(NULL != hFind)
{
  do
  {
    . . . // Do stuff with data
  } while(::InternetFindNextFile(hFind, &data));
  ::InternetCloseHandle(hFind);
}
```

Naturally, this means that code that can work with the Windows **FindFirstFile**/**FindNextFile** API for file system searches can, in large part, be reused or adapted for searching remote FTP systems. The major difference is that a search is initiated relative to a connection, represented in the form of the **WinInet** opaque handle type `HINTERNET`. Given this, you won't be too surprised to learn that there's a `basic_findfile_sequence` component in the **InetSTL** subproject that is a close relative of the one in the **WinSTL** subproject.

21.1 inetstl::basic_findfile_sequence

In this case, I am not ashamed to say that this definition of `basic_findfile_sequence` was largely a copy-and-paste exercise because the differences are almost entirely handled by the respective `filesystem_traits` classes from the two subprojects. The only difference in the public interface, shown in Listing 21.2, is that the constructors require a handle to an open connection.

Listing 21.2 Definition of the `basic_findfile_sequence` Constructors

```
// In namespace inetstl
template< typename C
        , typename X = throw_internet_exception_policy
        , typename T = filesystem_traits<C>
        >
class basic_findfile_sequence
{
  . . .
public: // Construction
  basic_findfile_sequence(HINTERNET          hconn
                  , char_type const*  pattern
                  , flags_type        flags = directories | files);
  basic_findfile_sequence(HINTERNET          hconn
                  , char_type const*  directory
                  , char_type const*  pattern
                  , flags_type        flags = directories | files);
  basic_findfile_sequence(HINTERNET          hconn
                  , char_type const*  directory
                  , char_type const*  patterns
                  , char_type         delim
                  , flags_type        flags = directories | files);
```

Despite the high conformance between the two sets of file search functions (and the two `basic_findfile_sequence` class templates), there is a significant difference in their semantics: Features of the FTP protocol mean that only one active file system enumeration can be supported for a given connection. Attempts to call `FtpFindFirstFile()` a second time will return *`NULL`*, and `GetLastError()` will give *`ERROR_FTP_TRANSFER_IN_PROGRESS`*, until the first search handle is closed. Thus, the code shown in Listing 21.3 will never enter the second loop.

Listing 21.3 Enumeration of FTP Server Files Using `inetstl::findfile_sequence`

```
using inetstl::findfile_sequence;

inetstl::session      sess;
inetstl::connection   conn( sess.get(), "ftp.some-host-or-other.com"
                          , INTERNET_INVALID_PORT_NUMBER, "anonymous"
                          , NULL, INTERNET_SERVICE_FTP
                          , INTERNET_FLAG_PASSIVE);
findfile_sequence     ffs(conn.get(), "/", "*.zip|*.bz", '|');
findfile_sequence::const_iterator b  = ffs.begin();
findfile_sequence::const_iterator b2 = ffs.begin(); // Always fails

for(; b != ffs.end(); ++b)
{}
for(; b2 != ffs.end(); ++b2)
{
  . . . // Never get here
}
```

This is handled by throwing an exception, in the `inetstl::basic_findfile_sequence::find_first_file_()` private static worker function, as shown in Listing 21.4.

Listing 21.4 Implementation of `inetstl::findfile_sequence ::find_first_ file()`

```
template <typename C, typename X, typename T>
HINTERNET basic_findfile_sequence<C, X, T>::
    find_first_file_( INTERNET          hconn
                    , char_type const*  spec
                    , flags_type     /* flags */
                    , find_data_type*   findData)
{
  HINTERNET hSrch = traits_type::find_first_file( hconn, spec
                                                 , findData);
  if(NULL == hSrch)
  {
    DWORD err = ::GetLastError();
    if(ERROR_FTP_TRANSFER_IN_PROGRESS == err)
    {
      exception_policy_type()("Already enumerating connection", err);
    }
    else
    {
      exception_policy_type()("Search failed", err);
    }
  }
  return hSrch;
}
```

If the class is used in compilations with exception handling turned off, or if the user chooses to parameterize with a null exception policy, the `begin()` method will return a correctly formed iterator instance equivalent to the `end()` iterator.

21.2 inetstl::basic_ftpdir_sequence

Although `inetstl::basic_findfile_sequence` has well-defined semantics and has been used successfully for a number of years (including in the FTP support in the **recls** library), it provides a restriction that in some cases is too onerous. Used with whole collection algorithms (those that operate on a sequence, rather than an iterator pair; covered in Volume 2), the *single active enumeration* issue has caused problems on a couple of occasions. Also of significance in this case, the latencies in setting up an FTP connection to, and in retrieving information from, an FTP server far outweigh the potential inefficiency of copying entries into a container.

The balance between efficiency and usability is a leading factor in the design of STL collections. In many cases, efficiency rightly inclines us to opt for a more constrained interface, where we have to work with restricted iterator and element reference categories. In this case, however, efficiency on the client side of the FTP connection is virtually irrelevant. Furthermore, usability strongly advises the use of caching to avoid the restrictive and occasionally hard-to-predict semantics of the single active enumeration. Hence, the **InetSTL** subproject now contains the collection `basic_ftpdir_sequence`, which is recommended for application programming in preference to `basic_findfile_sequence`. It maintains a list of entries in an internal `vector`, which is populated by a local instance of `basic_findfile_sequence` in the constructors. Its full definition is shown in Listing 21.5 (except for the implementations of the second and third constructors, which have implementations similar to the first).

Listing 21.5 Definition of `basic_ftpdir_sequence`

```
// In namespace inetstl
template< typename C
        , typename X = throw_internet_exception_policy
        , typename T = filesystem_traits<C>
        >
class basic_ftpdir_sequence
{
private: // Member Types
  typedef basic_findfile_sequence<C, X, T>      sequence_type_;
public:
  typedef typename sequence_type_::char_type    char_type;
  typedef typename sequence_type_::value_type   value_type;
  typedef typename sequence_type_::size_type    size_type;
  typedef int                                   flags_type;
private:
  typedef std::vector<value_type>               values_type_;
public:
  typedef typename values_type_::const_iterator const_iterator;
  typedef typename values_type_::const_reverse_iterator
                                               const_reverse_iterator;
public: // Member Constants
  enum search_flags
  {
      includeDots = sequence_type_::includeDots
    , directories = sequence_type_::directories
    , files       = sequence_type_::files
  };
public: // Construction
  basic_ftpdir_sequence(HINTERNET         hconn
                      , char_type const*  pattern
                      , flags_type        flags = directories | files)
  {
    sequence_type_  ffs(hconn, pattern, flags);
    std::copy(ffs.begin(), ffs.end(), std::back_inserter(m_values));
  }
  basic_ftpdir_sequence(HINTERNET         hconn
                      , char_type const*  directory
                      , char_type const*  pattern
                      , flags_type        flags = directories | files);
  basic_ftpdir_sequence(HINTERNET         hconn
                      , char_type const*  directory
                      , char_type const*  patterns
                      , char_type         delim
                      , flags_type        flags = directories | files);
public: // Iteration
  const_iterator         begin() const  { return m_values.begin();  }
  const_iterator         end() const    { return m_values.end();    }
  const_reverse_iterator rbegin() const { return m_values.rbegin(); }
  const_reverse_iterator rend() const   { return m_values.rend();   }
public: // Size
  size_type   size() const              { return m_values.size();   }
  bool        empty() const             { return m_values.empty();  }
private: // Member Variables
  values_type_  m_values;
};
typedef basic_ftpdir_sequence<char>     ftpdir_sequence_a;
typedef basic_ftpdir_sequence<wchar_t>  ftpdir_sequence_w;
typedef basic_ftpdir_sequence<TCHAR>    ftpdir_sequence;
```

The class has `size()`, `empty()`, and forward and reverse iterators. Because it uses a vector to store its values, its iterator category is *contiguous,* and because it is nonmutable, its element reference category is *fixed.* Such a class is like a tank: It's not very fast but is exceedingly hard to break.

CHAPTER 22

Enumerating Processes and Modules

Never let your sense of morals get in the way of doing what's right.

—Isaac Asimov

This is my shiny thing. And if you try and take it off me I may have to eat you.

—Cat, *Red Dwarf*

The Windows **Process Status** API (**PSAPI**) provides several methods for accessing system state information. For processes, the simplest and most common method is via the EnumProcesses() function, defined as follows:

```
BOOL EnumProcesses( DWORD*  pProcessIds
                  , DWORD   cb
                  , DWORD*  pBytesReturned);
```

pProcessIds points to an array into which the process identifiers of process objects in the system are returned. cb is the size of this array, measured in bytes. pBytesReturned points to a single variable that receives the number of bytes written to the array. The semantics of this function are that fewer process identifiers than exist may be reported if the array is not of sufficient size, so users are advised to call again with a larger array in the case where *pBytesReturned == cb. Other than that, the function is pretty straightforward. We'll wrap this with the pid_sequence STL extension collection.

PSAPI also provides a function EnumProcessModules(), for enumerating the modules of a given process:

```
BOOL EnumProcessModules(HANDLE    hProcess
                      , HMODULE*  lphModule
                      , DWORD     cb
                      , DWORD     *lpcbNeeded);
```

The semantics are directly analogous to EnumProcesses(). We'll wrap this with the process_module_sequence STL extension collection.

22.1 Collection Characteristics

Let's consider what characteristics for our collection are implied by the semantics of these functions.

- There's really no sense in the notion of a mutable process sequence, and the API does not facilitate this, so we can discount this now. The collection will be immutable.
- A process can complete and be removed from the active list at any point, quite independent of any client code of an `EnumProcesses()` call. Indeed, the fact that the entire process identifier list is returned en masse emphasizes the fact that a client must always treat the list as a snapshot of the system state at a given time.
- A collection wrapping `EnumProcesses()` would naturally own the array of process identifiers.
- Since the collection is immutable, the iterators are nonmutating. The underlying API fills a caller-supplied array of variables, so the containing class can simply maintain an internal buffer. Hence, the iterators can be nonmutating pointers into this array, so the category is *contiguous* (Section 2.3.6).
- The iterator is nonmutating contiguous; so as long as the collection employs *immutable RAII* (Section 11.1), the references can be nonmutating fixed (Section 3.3.2).

22.2 `winstl::pid_sequence`

This is probably the simplest fully functional STL extension collection class featured in this book. The full class definition is shown in Listing 22.1.

Listing 22.1 Declaration of the `pid_sequence` Class

```
// In namespace winstl
class pid_sequence
{
public: // Member Types
  typedef DWORD                                value_type;
  typedef processheap_allocator<value_type> allocator_type;
  typedef pid_sequence                         class_type;
  typedef value_type const*                    const_pointer;
  typedef value_type const*                    const_iterator;
  typedef value_type const&                    const_reference;
  typedef size_t                               size_type;
  typedef ptrdiff_t                            difference_type;
  typedef std::reverse_iterator<const_iterator>
                                               const_reverse_iterator;
public: // Construction
  pid_sequence();
  pid_sequence(class_type const& rhs);
  ~pid_sequence() throw();
public: // Iteration
  const_iterator          begin() const;
  const_iterator          end() const;
  const_reverse_iterator  rbegin() const;
  const_reverse_iterator  rend() const;
public: // Element Access
  const_reference operator [](size_type index) const;
public: // Size
  bool            empty() const;
  size_type       size() const;
private: // Member Variables
  typedef stlsoft::auto_buffer< value_type
                              , 64, allocator_type
                              >           buffer_type_;
  buffer_type_  m_pids;
private: // Not to be implemented
  class_type& operator =(class_type const&);
};
```

It's almost entirely boilerplate stuff. The notable features are the following.

- The use of `processheap_allocator<value_type>` as the allocator, rather than `std::allocator`. This facilitates the use of the class in small Windows components that don't link to the standard library.
- The provision of a copy constructor, which allows snapshots to be easily stored.
- The use of `stlsoft::auto_buffer` (Section 16.2), to handle buffer allocation and resizing for us. A conservative internal size of 64 elements (256 bytes) is specified, which means that in many cases no heap allocation will be required.

22.2.1 Simple Compositional Implementations

Although `auto_buffer` is *not* an STL container (Section 2.2), it nevertheless provides many container methods for the convenience of users implementing collections. Hence, the bulk of the `pid_sequence` methods can be implemented directly in terms of same-named methods of `auto_buffer`, which contributes to `pid_sequence`'s simplicity. For example, `empty()` returns `m_pids.empty()`:

```
bool pid_sequence::empty() const
{
  return m_pids.empty();
}
```

The same goes for `size()`, `begin()`, `end()`, `rbegin()`, and `rend()`. Although the subscript operator could be done in the same way, I elected to implement a precondition within `pid_sequence::operator []()`, rather than relying on the same precondition check provided within `auto_buffer::operator []()`. It's nice to know as soon as possible where precondition violations happen. Hence:

```
const_reference pid_sequence::operator [](size_type index) const
{
  WINSTL_MESSAGE_ASSERT("Index out of range", index < size());
  return m_pids[index];
}
```

Tip: Place precondition enforcements as close as possible to the public interface of the class.

22.2.2 Acquiring the Process Identifiers

That leaves only the default and copy constructors. Simplest first, the copy constructor is defined as follows:

```
pid_sequence::pid_sequence(pid_sequence const& rhs)
  : m_pids(rhs.m_pids.size())
{
  std::copy(rhs.m_pids.begin(), rhs.m_pids.end(), m_pids.begin());
}
```

Note that `m_pids` of the constructing instance is initialized to the size of the source instance's `m_pids`. Because `auto_buffer` does not (by design) define a copy constructor, the elements must be explicitly copied between the two instances. (At this point, one of my most sagacious reviewers expressed not a little exasperation with this [lack of] functionality, noting that "if `auto_buffer` had a copy constructor, `pid_sequence` wouldn't even need one written!" While I respect his point of view, I don't share it. For me, the importance of `auto_buffer` not even hinting at semantics it does not provide is paramount.)

That leaves us with only the meat of this particular class, the default constructor, as shown in Listing 22.2.

Listing 22.2 Default Constructor

```
pid_sequence::pid_sequence()
  : m_pids(buffer_type_::internal_size())
{
  DWORD   cbReturned;
  for(;;)
  {
    if(!::EnumProcesses(&m_pids[0], sizeof(value_type) * m_pids.size()
                       , &cbReturned))
    {
      throw system_exception( "Failed to enumerate processes"
                            , ::GetLastError());
    }
    else
    {
      const size_t n = cbReturned / sizeof(value_type);
```

```
        if(n < m_pids.size())
        {
          m_pids.resize(n);
          break;
        }
        else
        {
          const size_type size = m_pids.size();
          m_pids.resize(1);
          m_pids.resize(2 * size);
        }
      }
    }
  }
```

m_pids is initialized to have the actual size equivalent to its internal size, the maximum available without going to the heap. The call to EnumProcesses() is contained within a loop that doubles the size of m_pids with each iteration. If the call to EnumProcesses() fails, an exception is thrown. If the call succeeds, the number returned by EnumProcesses() is tested to determine whether all the available space of m_pids is used. If so, this can indicate that there are more process identifiers to be retrieved, so the execution loops and the call is repeated with twice as much space. If not, we are certain the call returned all possible identifiers and can break the loop.

Note that an API such as this clearly exhibits the preference for a collection rather than an iterator since the potentially costly call to enumerate the process identities is done once, whereas enumeration of the resultant sequence may be done as many times as required, with nothing more than the cost of incrementing a pointer.

You might be wondering about the curious call to m_pids.resize(1). This was added to prevent a slight inefficiency pointed out by keen-eyed Adi Shavit during an early review of this book. Can you spot the problem, and how the solution works? Answers on a postcard, please.

22.2.3 Working without Exception Support

For compatibility with contexts that do not use and/or link to standard library implementations, such as lightweight COM servers that link only to the Windows system libraries, the class is implemented to work when exceptions are not supported. Thus, the actual form is shown in Listing 22.3.

Listing 22.3 Revised Implementation of the Default Constructor

```
pid_sequence::pid_sequence()
  : m_pids(buffer_type_::internal_size())
{
  . . .
    if(!::EnumProcesses(&m_pids[0], sizeof(value_type) * m_pids.size()
                        , &cbReturned))
    {
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
```

```
      throw system_exception( "Failed to enumerate processes"
                            , ::GetLastError());
#else /* ? STLSOFT_CF_EXCEPTION_SUPPORT */
      m_pids.resize(0);
      break;
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */    }
    else
    {
        . . .
        m_pids.resize(1);
        if(!m_pids.resize(2 * size))
        {
          m_pids.resize(0);
          break;
        }
        . . .
```

When either `EnumProcesses()` or memory allocation fails, the size of `m_pids` is set to zero. `pid_sequence::size()` will return 0, and client code will be able to elicit a representative error code from the thread-specific Windows last-error function `GetLastError()`. Naturally, such uses are not the norm, but they're frequent enough to warrant this modest additional effort.

22.3 `winstl::process_module_sequence`

The implementation of `process_module_sequence` is virtually identical to `pid_sequence`, save for the fact that the value type is `HMODULE` rather than `DWORD` and that there is no default constructor. Instead, the constructor takes a process `HANDLE`, to be passed to `EnumProcessModules()`, as shown in Listing 22.4.

Listing 22.4 `process_module_sequence` **Value Type and Conversion Constructor**

```
// In namespace winstl
class process_module_sequence
{
public: // Member Types
  . . .
  typedef HMODULE                    value_type;
  . . .
public: // Construction
  explicit process_module_sequence(HANDLE hProcess);
  . . .
```

22.4 Enumerating All Modules on a System

With these two classes, it is a trivial matter to enumerate all the modules of all the processes within the system. To find out the path name of a module within a particular process, we use the **PSAPI** function `GetModuleFileNameEx()`, via a little helper function, `get_module_path()`, as shown in Listing 22.5. Since Windows module paths are length limited to *_MAX_PATH*, we can use the `stlsoft::basic_static_string` class template that we met in Section 19.3.11.

Listing 22.5 `get_module_path()` **Helper Function**

```
stlsoft::basic_static_string<char, _MAX_PATH>
    get_module_path(HANDLE hProcess, HMODULE hModule)
{
  stlsoft::basic_static_string<char, _MAX_PATH> s("", _MAX_PATH);
  DWORD cch = ::GetModuleFileNameEx(hProcess, hModule
                                          , &s[0], _MAX_PATH);
  if(0 == cch)
  {
    throw winstl::system_exception( "Could not elicit path"
                                  , ::GetLastError());
  }
  s.resize(cch);
  return s;
}
```

Assuming the user running the program has the appropriate privileges, the program shown in Listing 22.6 will display the path of each running process and the paths of its constituent modules. (For brevity, error handling and closing of the process handle, `hProcess`, is elided.)

Listing 22.6 Example Program Enumerating Processes and Modules

```
#include <winstl/system/pid_sequence.hpp>
#include <winstl/system/process_module_sequence.hpp>

int main()
{
  using winstl::pid_sequence;
  using winstl::process_module_sequence;

  pid_sequence                  pids;
  pid_sequence::const_iterator  b = pids.begin();

  for(; b != pids.end(); ++b)
  {
    HANDLE  hProcess = ::OpenProcess( PROCESS_QUERY_INFORMATION
                                    | PROCESS_VM_READ
                                    , false, *b);
    std::cout << get_module_path(hProcess, NULL) << ": " << std::endl;

    process_module_sequence                modules(hProcess);
    process_module_sequence::const_iterator b2 = modules.begin();

    for(; b2 != modules.end(); ++b2)
    {
      std::cout << "  " << get_module_path(hProcess, *b2)
                        << std::endl;
    }
  }
}
```

The following is a fragment of the output on my system as I'm writing this chapter:

```
\SystemRoot\System32\smss.exe:
  \SystemRoot\System32\smss.exe
  D:\WIN2K\system32\ntdll.dll
\??\D:\WIN2K\system32\csrss.exe:
  \??\D:\WIN2K\system32\csrss.exe
  D:\WIN2K\system32\ntdll.dll
. . .
H:\Publishing\Books\XSTL\test\Pt2_Collections\process_module_sequence_
test\vc6\Debug\Process_module_sequence_test.debug.exe:
  H:\Publishing\Books\. . .\Debug\Process_module_sequence_test.debug.exe
  D:\WIN2K\system32\ntdll.dll
  D:\WIN2K\system32\PSAPI.DLL
  D:\WIN2K\system32\KERNEL32.DLL
  D:\WIN2K\system32\MMSecBsc.dll
  D:\WIN2K\system32\MMCmnBas.dll
  D:\WIN2K\system32\NETAPI32.dll
. . .
```

22.5 Avoiding the System Pseudo Processes

One slight rub remains. Windows defines two processes, the Idle process and the System pseudo process, whose details may not be elicited. The Idle process is invoked when nothing else is running, in order to have *something* executing and idle time accounted for. The System process is where most of the kernel-mode system threads live. Neither of these are full processes. Naturally enough, messing around with the internals of either of these would be a bad idea.

If we run the code in Listing 22.6, the call to `OpenProcess()` for the Idle process (whose process identifier is *0*) fails, and `GetLastError()` returns *ERROR_INVALID_PARAMETER*. If we cause the code to execute for the System process (whose process identifier is a small, nonzero, operating-system-dependent value), the call to `GetModuleFileNameEx()` fails, and `GetLastError()` returns *ERROR_PARTIAL_COPY*.

Since there's precious little you can do with them, the appropriate thing to do here is to elide the Idle and System process identifiers from the list. Naturally, we can add tests to our outer loop to effect this:

```
  . . .
  for(; b != pids.end(); ++b)
  {
    if( 0 == *b ||                              // Idle process?
        AcmeLib_GetSystemProcessId() == *b) // System process?
    {
      continue;
    }
    HANDLE  hProcess = ::OpenProcess( PROCESS_QUERY_INFORMATION
                                    | PROCESS_VM_READ
                                  , false, *b);
  . . .
```

However, there are two reasons not to do so. First, an STL extension, like any other class/library, should endeavor to simplify the life of the user, both by assisting with onerous and/or obvious tasks and, where possible, by lifting the level of abstraction of the underlying service(s); the *Principle of Least Surprise* applies. Since, by the criteria of the Windows operating system, neither the Idle nor System processes are full processes, it is appropriate that the `pid_sequence` provides neither for us. Simply put, such client code is at once easier to write and more transparent once written if `pid_sequence` provides such a filtering service (as long as it does so in a manner that is itself clear and discoverable). A library that provides such partial or feature-incomplete abstractions will not prosper.

Tip: Shield users from inconvenient or unusable aspects of underlying APIs in STL extensions. (An awkward or vexing abstraction is self-defeating.)

Second, performing the elision of the two process identifiers in client code increases the testing obligations of all users and violates *DRY SPOT* (Chapter 5) for any that need to perform process identifier enumeration in multiple places. This argument is particularly compelling in this case, given the fact that the actual value of the System process's process identifier has wandered somewhat with different releases of the NT operating system family.

Hence, the actual implementation of `pid_sequence` defines three member values in an anonymous enumeration. The constructor takes a single `flags` parameter, defaulted appropriately, and elides special process identifiers accordingly (Listing 22.7).

Listing 22.7 Definition of Constants, Constructor, and Feature Methods

```
class pid_sequence
{
  . . .
public: // Member Constants
  enum
  {
      elideIdle   = 0x0001
    , elideSystem = 0x0002
    , sort        = 0x0004
  };
public: // Construction
  explicit pid_sequence(uint32_t flags = elideIdle | elideSystem);
  . . .
```

```
public: // System Features
  static value_type idleProcessId();   // 0
  static value_type systemProcessId(); // 2, 4, or 8, depending on OS
  . . .
```

The constructor body encapsulates the necessary process identifier elicitation logic, removing the Idle and/or System identifiers (determined by the two static utility methods) from `m_pids` if `elideIdle` and/or `elideSystem` flags are specified. It also calls `std::sort()` on `m_pids` if the `sort` flag is specified.

22.6 Handling Optional API Headers

The three Windows API functions `EnumProcesses()`, `EnumProcessModules()`, and `GetModuleFileNameEx()` are located, along with the other members of the **Process Status** API, within the *PSAPI.DLL* dynamic library. Their C/C++ definitions are to be found in the Windows header *<psapi.h>*, and you link to the dynamic library via the import library *psapi.lib* (or *libpsapi.a* for GCC on Windows).

PSAPI is supported only on the Windows NT family of operating systems, and the header files and import libraries are therefore not provided with all Windows compilers. For example, the Digital Mars C/C++ compiler (as of version 8.45) does not provide them.

Such situations are a bit of a thorn in the foot of authors who wish to write portable libraries with that “works out of the box” feel. Thankfully, there’s a perfect implement in the toolbox to assist us in just such a situation: `dl_call()` (Section 16.6). In each place in the components described where the **PSAPI** functions are called, there’s actually a conditional test for the well-known include guard symbols of the *<psapi.h>* header(s), and in the case where that’s not found, the call is made via `dl_call()`. Listing 22.8 shows the actual code from the `pid_sequence` constructor.

Listing 22.8 Optional Runtime Invocation of the System Library Function

```
inline pid_sequence::pid_sequence(ws_uint32_t flags)
    : m_pids(buffer_type_::internal_size())
{
  DWORD   cbReturned;
  for(;;)
  {
#if defined(_PSAPI_H_) || \
    defined(_PSAPI_H)
    if(!::EnumProcesses(&m_pids[0], sizeof(value_type) * m_pids.size()
                      , &cbReturned))
#else /* ? psapi */
    if(!dl_call<BOOL>("PSAPI.DLL", "S:EnumProcesses", &m_pids[0]
                    , sizeof(value_type) * m_pids.size(), &cbReturned))
#endif /* psapi */
    {
    . . .
```

With this form, the code compiles and executes without issue, irrespective of the presence or absence of *<psapi.h>*. It does, of course, require that *PSAPI.DLL* is present on the system to be loaded, but that's so in either case. The only complication is that the invocation of `GetFileNameEx()` has to give the name as *"GetFileNameExA"* since Windows API functions that manipulate characters have ANSI/multibyte (*'A'* suffix) and Unicode/wide string (*'W'* suffix) variants.

22.7 Summary

The **PSAPI** functions `EnumProcesses()` and `EnumProcessModules()` are easy to understand, but they are not easy (or pleasant) to use directly. The `pid_sequence` and `process_module_sequence` classes provide convenient ***Façades*** that have the following advantages.

- They hide the inconvenient repeat-until-no-longer-unsure nature of the functions (the *Principle of Simplicity*).
- They elide unwanted and unusable process identifiers (the *Principle of Simplicity*).
- They offer a snapshot of the system state at the time of call, which can be manipulated efficiently (via contiguous iteration).
- They handle the case where some compilers' Windows header files are missing **PSAPI** elements, using `dl_call()` (the *Principle of Least Surprise*).
- They offer simple implementation, relying on composition in terms of `auto_buffer` (the *Principle of Composition*).
- They work correctly with or without exception support (the *Principle of Diversity*).
- They work nicely together, as shown in the example program (the *Principle of Modularity*).
- And finally, I inadvertently incurred the *Principle of Optimization* in leaving the implementations open for Adi's efficient `resize(1)` trick.

And that's it for operating specifics for a while. Next we're going to go for a ride on the hypothetical plane of an infinite arithmetic sequence.

CHAPTER 23

The Fibonacci Sequence

If liberty means anything at all, it means the right to tell people what they do not want to hear.

—George Orwell

Subtlety chases the obvious in a never-ending spiral and never quite catches it.

—Nero Wolfe

23.1 Introduction

Those who enjoy mathematical elegance may share my appreciation of the Fibonacci sequence and its associated relationship, the Golden Ratio. In this chapter we're going to look at how we might represent this mathematical sequence as a collection, in the form of an STL sequence class, and then consider whether it might be better represented as an iterator, before finally coming back to seeing how a range-limited sequence is the most discoverable representation.

Unlike the other STL extensions described in this book, this one does not derive from any libraries. It is entirely pedagogical. As such, I trust you'll bear with me in some of the less practicable fancies used to illuminate the STL extension issues covered. For those who prefer real examples, worry not, this is the only such fanciful example in the whole book.

23.2 The Fibonacci Sequence

The Fibonacci sequence is a series of numbers, starting with the pair 0 and 1, where the value of each element is calculated as the sum of the two preceding it. Hence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1,597, 2,584, 4,181, 6,765, and so on, ad infinitum.

The ratio of each entry in the series to its next tends toward an irrational constant, known as the Golden Ratio, whose value is approximately 1.61803398875. The Golden Ratio appears to crop up in all kinds of places in the universe, from the ratio of aesthetically pleasing picture frames to the twirls of conch shells to the dimensions of the Parthenon. If you've not come across it before, I recommend you check it out.

23.3 Fibonacci as an STL Sequence

My first instinct when thinking about how to represent a mathematical sequence was to use an STL-compliant sequence, as shown in Listing 23.1. As we'll see, however, this is not as nice a fit as we might think. Since this is a notional collection—there are no elements in existence anywhere—the enumeration of the values in the sequence is carried out in the iterator, an instance of the member class `const_iterator`, whose element reference category is *by-value temporary* (Section 3.3.5).

Listing 23.1 `Fibonacci_sequence` **Version 1 and Its Iterator Class**

```
class Fibonacci_sequence
{
public: // Member Types
  typedef uint32_t  value_type;
  class             const_iterator;
  . . .
public: // Iteration
  const_iterator  begin() const
  {
    return const_iterator(0, 1);
  }
  const_iterator  end() const;
  . . .
};

class Fibonacci_sequence::const_iterator
  : public std::iterator< std::forward_iterator_tag
                        , Fibonacci_sequence::value_type, ptrdiff_t
                        , void, Fibonacci_sequence::value_type // BVT
                        >
{
public: // Member Types
  typedef const_iterator                  class_type;
  typedef Fibonacci_sequence::value_type  value_type;
public: // Construction
  const_iterator(value_type i0, value_type i1);
public: // Iteration
  class_type& operator ++();
  class_type operator ++(int);
  value_type operator *() const;
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    return m_i0 == rhs.m_i0 && m_i1 == rhs.m_i1;
  }
  . . .
private: // Member Variables
  value_type  m_i0;
  value_type  m_i1;
};

inline bool operator ==(Fibonacci_sequence::const_iterator const& lhs
                      , Fibonacci_sequence::const_iterator const& rhs)
```

```
{
  return lhs.equal(rhs);
}
inline bool operator !=(Fibonacci_sequence::const_iterator const& lhs
                      , Fibonacci_sequence::const_iterator const& rhs)
{
  return !lhs.equal(rhs);
}
```

Listing 23.2 shows the implementations of the only two nonboilerplate methods of `const_iterator`.

Listing 23.2 Version 1: Preincrement and Dereference Operators

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  value_type  res   = m_i0 + m_i1;
              m_i0  = m_i1;
              m_i1  = res;
  return *this;
}
value_type Fibonacci_sequence::const_iterator::operator *() const
{
  return m_i0;
}
```

Each time the preincrement operator is called, the next result is calculated and moved into `m_i1`, after `m_i1` is first moved into `m_i0`. The current result is held in `m_i0`. Note that the `const_iterator` could just as easily support the *bidirectional* iterator category, wherein the predecrement operator would subtract `m_i0` from `m_i1` to get the previous value in the sequence. I've not done so simply because the Fibonacci is a forward sequence.

Because the sequence is infinite, `end()` is defined to return an instance of `const_iterator` whose value is such that it will never compare `equal()` to a valid iterator. (The implementation shown in Listing 23.3 corresponds to *`Fibonacci_sequence_1.hpp`* on the CD.)

Listing 23.3 Version 1: `end()` Method

```
class Fibonacci_sequence
{
  . . .
  const_iterator  end() const
  {
    return const_iterator(0, 0);
  }
  . . .
```

Let's now use this definition of the sequence:

```
Fibonacci_sequence                  fs;
Fibonacci_sequence::const_iterator  b = fs.begin();

for(size_t i = 0; i < 10; ++i, ++b)
{
  std::cout << i << ": " << *b << std::endl;
}
```

This works a treat, giving the first ten elements in the Fibonacci sequence: 0–34. However, as we well know, iterators like to work with algorithms and usually take them in pairs, for example:

```
std::copy(fs.begin(), fs.end()
    , std::ostream_iterator<Fibonacci_sequence::value_type>(std::cout
                                                            , " "));
```

Unfortunately, there are two problems with this statement. First, it runs forever, which represents somewhat of an inconvenience when you want to use your computer for something worthwhile, such as updating it with the latest virus definitions and operating system patches to fill up that last 12GB of disk you were saving for your database of fine European chocolatiers. You might wonder whether we will be saved when the overflowed arithmetic happens on a result whose value modulo 0x10000000 is 0. Although this does eventually occur—after 3,221,225,426 iterations, as it happens—the iterator still does not compare equal to the `end()` iterator because its `m_i1` member is nonzero. Since it is not possible for both members to be 0 at one time, the code will loop forever.

Second, after the forty-seventh iteration, the results returned are no longer members of the Fibonacci sequence but pseudo junk values as a consequence of overflow of our 32-bit value type. As we know, computers don't generally like to live in the infinite, and integral types are particularly antipathetic to unconstrained ranges.

23.3.1 Interface of an Infinite Sequence

We'll deal with the first problem first. Since the Fibonacci sequence is infinite, one option would be to make the `Fibonacci_sequence` infinite also. This is easily effected by removing the `end()` method. The sequence is now quite literally one without end. Now users of the class cannot make the mistake, shown earlier, of passing an ostensibly bounded [`begin()`, `end()`) range to an algorithm since there is no end.

In my opinion, this is the most appealing form from a conceptual point of view because the public interface of the sequence is representing its semantics most clearly. However, it's not terribly practical because, as we've already seen, overflow occurs after a soberingly finite number of steps. For infinite sequences whose values are bound within a representable range, this would be a good candidate approach, but it's not suitable for the Fibonacci sequence.

Note that this reasoning also rules out the possible alternative implementations of Fibonacci sequences as independent iterator classes or as generator functions.

23.3.2 Put a Contract on It

Let's now take the sensible step of putting some contract programming protection into the preincrement operator before we attempt to use the sequence. (The implementation shown in Listing 23.4 corresponds to *Fibonacci_sequence_2.hpp* on the CD.)

Listing 23.4 Version 2: Preincrement Operator

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  STLSOFT_MESSAGE_ASSERT("Exhausted integral type", m_i0 <= m_i1);
  value_type  res   = m_i0 + m_i1;
              m_i0  = m_i1;
              m_i1  = res;
  return *this;
}
```

In executing the `std::copy` statement shown previously, we find that the assertion is fired on the increment after output of the value 2,971,215,073. At this point, the previous value was 1,836,311,903, so we would expect `m_i1` to be 4,807,526,976. However, that exceeds the maximum value representable in a 32-bit unsigned integer (4,294,967,295), so the result is truncated (to 512,559,680), and the assertion fires. Hence, although we've managed to iterate 48 items, the last increment left the iterator in an invalid state, an unincrementable state, so there are only actually 47 viable enumerable values from a 32-bit representation.

I want to stress the distinction between providing a usable interface and guarding against misuse, well exemplified in this case. Thus far, our Fibonacci sequence does not have a usable interface—since its failure is a matter of surprise—but now, with the introduction of the assertion, it does have protection against its misuse.

23.3.3 Changing Value Type?

Perhaps a solution lies in using a different value type. Obviously, using `uint64_t` is only going to be a small bandage over the problem, allowing us to enumerate 93 steps and get to 7,540,113,804,746,346,429. And once we're there, we still precipitate a contract violation, indicating abuse of the sequence.

Maybe floating point is the way to go? (This implementation corresponds to *Fibonacci_sequence_3.hpp* on the CD.) Alas, no—32-bit `float` enters INF territory at 187 entries, 64-bit `double` at 1,478. Furthermore, since the entries in the sequence are not nicely rounded 10^N values, rounding errors creep in as soon as the exponent value reaches the extent of the mantissa.

Conceivably, a `BigInt` type using coded decimal evaluation would be able to go infinite, but it would have correspondingly poor performance. (Readers are invited to submit such a solution. In reward I can promise the unquantifiable fame that will come from having your name on the book's Web site.)

23.3.4 Constraining Type

To avoid floating-point inaccuracies, we would like to constrain the value type to be integral. To avail ourselves of the maximum range of the type and to catch overflow, we would like to constrain the value type to be unsigned. These constraints are achieved by providing a destructor for the sequence for this very purpose, as shown in Listing 23.5.

Listing 23.5 Constraints Enforced in the Destructor

```
Fibonacci_sequence::~Fibonacci_sequence() throw()
{
  using stlsoft::is_integral_type; // Using using declarations . . .
  using stlsoft::is_signed_type;   // . . . to fit in book. ;-)
  STLSOFT_STATIC_ASSERT(0 != is_integral_type<value_type>::value);
  STLSOFT_STATIC_ASSERT(0 == is_signed_type<value_type>::value);
}
```

You might think it strange to put in such constraints in a non-template class. The reason is simple: Maintenance programmers (including those who maintain their own code, hint, hint) are wont to change things without putting in all the big-picture research (i.e., reading all documentation). By putting in constraints, you are literally constraining any future changes from violating the design assumptions, or at least from doing so without extra thought.

Tip: Use constraints even in non-template classes to restrict and inform future maintenance activities.

I prefer to place constraints in the destructor of template classes because it's the method we can most rely on being instantiated. In non-template classes, I continue to use it for consistency.

23.3.5 Throw `std::overflow_error`?

One possible approach is to change the precondition enforcement assertion to be a legitimate runtime condition and to throw an exception. (The implementation shown in Listing 23.6 corresponds to *Fibonacci_sequence_4.hpp* on the CD.)

Listing 23.6 Version 4: Preincrement Operator

```
class_type& Fibonacci_sequence::const_iterator::operator ++()
{
  if(m_i1 < m_i0)
  {
    throw std::overflow_error("Exhausted integral type");
  }
  value_type  res   = m_i0 + m_i1;
  . . . // Same as Version 2
```

Although, in strict terms, this is a legitimate approach, it really doesn't appeal. The so-called exceptional condition is not an unpredictable emergent characteristic of the system at a particular state and time but an entirely predictable and logical consequence of the relationship between the modeled concept and the type used to hold its values. Using an exception in this case just smacks of Java hackery.

I think it's clear at this point that we should either decide to represent the Fibonacci sequence as something that is genuinely infinite, with suitable indicators, or provide a mechanism for providing finite endpoints.

23.4 Discoverability Failure

Although the limit of a Fibonacci sequence for a given unsigned integral type is predictable and constant, requiring users of a type to know this either a priori or a posteriori is a bit rich, to say the least. Quite simply, people would not use such a component.

Our three current candidate implementations present unappealing alternatives.

1. Define the sequence without `end()`. This precludes any use of (`begin()`, `end()`) arguments to algorithms, but it does not preclude two iterators derived from `begin()` being used with algorithms. Further, there's nothing stopping users from gaily advancing their `begin()`-derived iterator past the point of overflow, and nothing to guide them in avoiding this.
2. Define the sequence with `end()` and rely on users' common sense not to use `end()` for anything at all. If they go into overflow, their program will die in a contract violation.
3. Throw an exception when overflow occurs. Despite this giving a tepid feeling of robustness, it's just as much a discoverability transgression as the other two options, and it also encourages a style of programming that is rightly confined to the world of virtual machines and seven-figure installation and deployment consultancy contracts.

Tip: Avoid using exceptions for failures that are a predictable result of the normal use of a component.

So, either the Fibonacci sequence is not something we should attempt to play with in an STL kind of way, or we need to apply some "finity" to it.

23.5 Defining Finite Bounds

There are two clear and related solutions to this problem.

1. Have `end()` return an iterator in the range [`begin()`, ∞) whose value does not overflow the value type.
2. Allow the user to specify an upper limit for the effective range provided by the sequence, represented in the value returned by `end()`. This value would have to be within the valid range.

A good implementation would provide both, where solution 1 is merely the default form of solution 2. We'll examine this by looking at the user-specified limits first.

23.5.1 Iterators Rule After All?

Before we proceed, I must cover an issue that some readers may now be considering. Earlier I ruled out the representation of Fibonacci sequences as independent iterators. The cunning linguists among you may be considering a form that does exactly that, as in:

```
std::copy(Fibonacci_iterator(), Fibonacci_iterator() + 40
    , std::ostream_iterator<Fibonacci_sequence::value_type>(std::cout
                                                           , " "));
```

In this case, the putative `Fibonacci_iterator` would implement the addition operator, such that the expression `Fibonacci_iterator() + 40` would evaluate to an instance that would terminate the iteration of a default-constructed iterator on its fortieth increment. At first blush this seems like an adequate solution to the problem.

However, the problem is that use of the addition operator on an iterator indicates that the iterator type is a random access iterator. Further, random access iterators have constant time complexity. To be sure, we're perforce violating pure STL requirements here and there in STL extension. But such violations are never done without due care and particular attention to the effects on discoverability and the *Principle of Least Surprise*. For example, it's hard to imagine that users of the **InetSTL** `findfile_sequence` class (Section 21.1), an STL collection that provides iteration of remote FTP host directory contents, will assume any particular complexity guarantees, given the vagaries of Internet retrieval. However, I suggest that it's far more likely that someone would assume constant time seeing pointer arithmetic syntax on an iterator.

Further, since a user will reasonably expect to be able to type `*(Fibonacci_iterator() + 40)` if he or she can type `Fibonacci_iterator() + 40`, we'd have to implement full random access semantics. But, as far as I know, there's no constant-time function integral formula with which you can determine the *N*th value of the Fibonacci sequence. (There are a couple of formulas that may be used, but they rely on the square root of five, which would rely on floating-point calculation. One of them is `((1 + sqrt(5)) / 2) - ((1 - sqrt(5)) / 2) ^ n`. I'm just enough of a computer numerist to know that I know far too little about floating point to be confident of writing a 100% correct sequence using floating-point calculations.)

Thus we would have to perform a number of forward or backward calculations to arrive at the required value, which is a linear-time operation. This would be a very unobvious violation of a user's expectations and is, in my opinion, unacceptable.

Tip: Beware of changing the complexity of built-in operators, particularly for random access iterators.

(Of course, we could provide amortized constant time by storing the calculated values in an array. We could go further and provide a static member array with precalculated values. We could even use template metaprogramming and effect compile-time calculation. But the purpose of this chapter is pedagogical. Feel free to do any of these, and let me know how it goes. I'll gladly post interesting solutions on the book's Web site.)

23.5.2 Constructor-Bound Range

One use case of a sequence might be to retrieve the first N elements in the sequence. It would be straightforward to implement the sequence and iterator classes such that you would specify the number of elements in the sequence constructor, which would then return a bounding iterator instance via its `end()` method, as shown in Listing 23.7. (This corresponds to *`Fibonacci_sequence_5.hpp`* on the CD.)

Listing 23.7 Version 5: Constructor and `end()` Method

```
public: // Construction
  explicit Fibonacci_sequence(size_t n) // Max # entries to enumerate
    : m_numEntries(n)
  {}
  . . .
public: // Iteration
  const_iterator begin();
  const_iterator end()
  {
    return const_iterator(m_numEntries); // Define end of sequence
  }
  . . .
```

You could use this as follows:

```
Fibonacci_sequence fs(25);

std::copy(fs.begin(), fs.end()
  , std::ostream_iterator<Fibonacci_sequence::value_type>(std::cout));
```

This could be implemented by adding an additional `m_stepIndex` member to `const_iterator`, which would be incremented each time `operator ++()` is called, and by evaluating equality (in `equal()`) by comparing the `m_stepIndex` members of the comparands (Listing 23.8).

Listing 23.8 Version 5: `const_iterator`

```
class Fibonacci_sequence::const_iterator
  : . . . // As shown previously
{
public: // Construction
  const_iterator(value_type i0, value_type i1)
    : m_i0(i0)
    , m_i1(i1)
    , m_stepIndex(0)
  {}
  const_iterator(size_t stepIndex)
    : m_i0(0)
    , m_i1(0)
    , m_stepIndex(stepIndex)
  {}
public: // Iteration
  class_type& operator ++()
  {
    . . . // Perform the advancement summations as before
    ++m_stepIndex;
    return *this;
  }
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    return m_stepIndex == rhs.m_stepIndex;
  }
  . . .
```

This is a nice solution, and it also allows us to meaningfully support the `empty()` method. However, there's an equally valid use case, that of constraining the enumeration within a given integral range, for example, to enumerate all entries less than the value 1,000,000,000. The sequence might look like that shown in Listing 23.9. (This implementation corresponds to *Fibonacci_sequence_6.hpp* on the CD.)

Listing 23.9 Version 6: Constructor and `end()` Method

```
class Fibonacci_sequence
{
  . . .
public: // Construction
  explicit Fibonacci_sequence(value_type limit); // Value ceiling
    : m_limit(limit)
  {}
  . . .
public: // Iteration
  const_iterator begin();
  const_iterator end()
  {
    return const_iterator(m_limit); // Define sequence ceiling
  }
  . . .
```

Comparison would be conducted by the somewhat abstruse implementation of `equal()` shown in Listing 23.10. (There's an overflow bug in here, which I've left since this is a pedagogical class. Try setting the limit to 1,836,311,904 for a 32-bit unsigned value type. If readers want to implement the full testing for overflow, I'll be happy to post any correct solutions on the book's Web site.)

Listing 23.10 Version 6: `const_iterator::equal()` **Method**

```
bool
 Fibonacci_sequence::const_iterator::equal(class_type const& rhs)
  const
{
  if( 0 != m_i1 &&
      0 != rhs.m_i1)
  {
    // Both definitely normal iterable instances
    return m_i0 == rhs.m_i0 && m_i1 == rhs.m_i1;
  }
  else if(0 != m_threshold &&
          0 != rhs.m_threshold)
  {
    // Both definitely threshold sentinel instances
    return m_threshold == rhs.m_threshold;
  }
  else
  {
    // Heterogeneous mix of the two types
    if(0 == m_threshold)
    {
      return m_i0 >= rhs.m_threshold;
    }
    else
    {
      return rhs.m_i0 >= m_threshold;
    }
  }
}
```

A more flexible class would accommodate both these usage models. But doing so presents the sticky problem of how to unambiguously construct an instance of the sequence for either use case. One solution would be to use a two-parameter constructor, as follows:

```
  . . .
public: // Construction
  Fibonacci_sequence(size_t n, value_type limit);
  . . .
```

The parameter for the end-marker type not used would be given a stock value, for example:

```
Fibonacci_sequence(0, 10000); // This uses a limit of 10,000
Fibonacci_sequence(20, 0);    // This gives a sequence of 20 entries
```

Obviously, this is an inelegant and highly error-prone approach. A slightly less revolting alternative would be to use an enumeration to indicate the type of end marker required and use a `value_type` parameter for both the threshold and the number of entries:

```
  . . .
public: // Member Constants
  enum LimitType { thresholdLimit, countLimit };
public: // Construction
  Fibonacci_sequence(value_type limit, LimitType type);
  . . .
```

23.5.3 True Typedefs

The best solution uses true typedefs (Section 12.3), which facilitate the unambiguous overloading of essentially similar or even identical types. The final implementation of the `Fibonacci_sequence` does this, as shown in Listing 23.11. (This corresponds to *`Fibonacci_sequence_7.hpp`* on the CD.) Note the use of precondition enforcements in both constructors. A valid design alternative would be to throw `std::out_of_range` (since the user's value is not predictable).

Listing 23.11 Version 7: Class Declaration and Traits Class

```
template <typename T>
struct Fibonacci_traits;

template <>
struct Fibonacci_traits<uint32_t>
{
  static const uint32_t maxThreshold  = 2971215073;
  static const size_t   maxLimit      = 47;
};
template <>
struct Fibonacci_traits<uint64_t>
{
  static const uint64_t maxThreshold  = 12200160415121876738;
  static const size_t   maxLimit      = 93;
};

class Fibonacci_sequence
{
public: // Member Types
  typedef ?? uint32_t or uint64_t ??        value_type;
  typedef Fibonacci_traits<value_type>       traits_type;
  typedef true_typedef<size_t, unsigned>     limit;
  typedef true_typedef<value_type, signed>   threshold;
  class                                      const_iterator;
public: // Construction
  explicit Fibonacci_sequence(limit l = limit(traits_type::maxLimit))
    : m_limit(l.base_type_value())
    , m_threshold(0)
  {
    STLSOFT_MESSAGE_ASSERT( "Sequence limit exceeded"
                          , l <= traits_type::maxLimit());
  }
  explicit Fibonacci_sequence(threshold t)
    : m_limit(0)
    , m_threshold(t.base_type_value())
  {
    STLSOFT_MESSAGE_ASSERT( "Sequence threshold exceeded"
                          , t <= traits_type::maxThreshold());
  }
public: // Iteration
  const_iterator  begin() const;
  const_iterator  end() const
  {
    return (0 == m_limit)
              ? const_iterator(m_threshold)
              : const_iterator(m_limit, 0);
  }
public: // Size
  bool   empty() const
  {
    return 0 == m_limit && 0 == m_threshold;
  }
  size_t max_size() const
  {
    return traits_type::maxLimit;
  }
private: // Member Variables
  const size_t      m_limit;
  const value_type  m_threshold;
};
```

Note the use of the traits. Although they're not required by the definition of the sequence as it stands, they serve two important purposes. First, they provide a clear and obvious place for the limit and threshold magic numbers to reside, as well as making them largely self-documenting. Second, should you choose to use a 32- or 64-bit value type, the change involves just a single line.

The iterator class can now be defined as shown in Listing 23.12.

Listing 23.12 Version 7: `const_iterator`

```
class Fibonacci_sequence::const_iterator
  : . . . // As shown previously
{
public: // Member Types
  typedef Fibonacci_sequence::value_type      value_type;
  typedef Fibonacci_sequence::const_iterator  class_type;
private: // Construction
  friend class Fibonacci_sequence;
  const_iterator();
  const_iterator(Fibonacci_sequence::limit lim);
  const_iterator(Fibonacci_sequence::threshold t);
  . . . // Iteration and Comparison methods as before
};
```

With this definition, all the following are well defined (and thereby value constrained):

```
typedef Fibonacci_sequence  fibseq_t;
fibseq_t  fs(fibseq_t::limit(0));              // Empty sequence
fibseq_t  fs(fibseq_t::limit(1));              // 1 value
fibseq_t  fs(fibseq_t::limit(10));             // 10 values
fibseq_t  fs(fibseq_t::limit(47));             // 47 values
fibseq_t  fs;                                  // 47 values
fibseq_t  fs(fibseq_t::threshold(0));          // Empty sequence
fibseq_t  fs(fibseq_t::threshold(1));          // 1 value
fibseq_t  fs(fibseq_t::threshold(2));          // 3 values
fibseq_t  fs(fibseq_t::threshold(47));         // 10 values
fibseq_t  fs(fibseq_t::threshold(100));        // 12 values
fibseq_t  fs(fibseq_t::threshold(1000000000)); // 45 values
```

Equally important, the following are not well defined, and the user knows this because he or she can evaluate them against the member constants `Fibonacci_sequence::traits_type::maxLimit` and `Fibonacci_sequence::traits_type::maxThreshold`. Furthermore, because of the enforcements placed in the constructor bodies, the user finds out immediately when something is wrong, rather than at a later point during enumeration when the values overflow.

```
fibseq_t  fs(fibseq_t::limit(50));             // Breaks ctor precond
fibseq_t  fs(fibseq_t::threshold(2971215075)); // Breaks ctor precond
```

23.6 Summary

This chapter has covered the issues related to implementing an unbounded (infinite) notional collection. It has highlighted the imperfect fit between such collections and the strict finitude of C++'s integral types and the assumed boundedness of STL iterator pairs. Primarily, we've encountered, and eventually avoided, violations of the Goose Rule (Section 10.1.3).

We’ve seen that contracts—a mechanism for ensuring program adherence to design—are a thoroughly inappropriate mechanism for dealing with the conflict between conceptual infinity and the finitude of the language’s types. We considered exceptions, but this was discounted because the exception point not only was not exceptional but also was entirely predictable. The requirement that users provide the limit to the range was unavoidable.

We determined two equally valid and desirable ways of limiting the range. Each was unambiguous, and providing a discoverable implementation was straightforward. But supporting both led to ambiguous and undiscoverable syntax, with unattractive compromises. Applying true typedefs saved the day, providing a class interface that is clear and discoverable, with the positive side effect that client code is itself more transparent.

Chapter 24

Adapting MFC's `CArray` Container Family

It has become appallingly obvious that our technology has exceeded our humanity.

—Albert Einstein

My God, you've gotten fat!

—Edna Mole, *The Incredibles*

24.1 Introduction

In this chapter we look at the adaptation of collections that are fully fledged containers (Section 2.2)—they own their elements, manage their memory, control their lifetimes, and provide operations by which they may be manipulated. For subject matter we'll be using the array containers from the **Microsoft Foundation Classes** (**MFC**) library. Those of you gentle readers who may wish to run away at the mere mention of **MFC**, please staunch your understandable sentiment. It is the very anachronism of these containers that makes them suitable for our pedagogical purposes because they bring many important nuances of the STL into sharp focus. In any case, there are plenty of people who are still using **MFC** in the real world and whose lives we should seek to simplify.

The two components we will feature are the **MFCSTL** class templates `CArray_cadaptor` and `CArray_iadaptor`. The former is a ***Class Adaptor***—it adapts **MFC** `CArray` container family types to a `std::vector`-like interface. The latter is an ***Instance Adaptor***—it is used to temporarily adapt *instances* of `CArray` family containers to a `std::vector`-like interface so that they can be used in the style of *STL collections* (Section 2.2).

24.2 Motivation

Working with components from different libraries whose idioms don't mix is an arduous task, particularly so when mixing **MFC** and **STL**. Providing a bridging technology could afford significant benefits to the many projects that still use **MFC** but whose authors and maintainers wish to avail themselves of the modern way of doing things that **STL** provides. Suppose we have a function `readLines()` that loads the contents of a text file into an array of strings, in the form of the MFC container class `CStringArray`:

```
void readLines(TCHAR const *fileName, CStringArray &arr);
```

Further suppose that we need to have all the file's lines read in, sorted, and presented in a list. We might add a function `readOrderedLines()` to do this. Doing it in straight **MFC** might look like the version shown in Listing 24.1, which also contains a supporting comparison function (to be passed to `::qsort()`) and some client code to display the results. (Remember, from Section 5.2.2, that `TCHAR` and `LPCTSTR` are typedefs for `char` and `char const*`, or `wchar_t` and `wchar_t const*` for Unicode builds.)

Listing 24.1 MFC Version of `readOrderedLines()`

```
int compare_CStrings(void const* pv1, void const* pv2)
{
  ASSERT(NULL != pv1 && NULL != pv2);
  CString const *str = static_cast<CString const*>(pv1);
  return str->Compare(static_cast<TCHAR const*>(pv2));
};
void readOrderedLines(char const *fileName, CStringList &lst)
{
  CStringArray  arr;
  // 1. Read in lines into array
  readLines(fileName, arr);
  // 2. Sort lines
  ::qsort(arr.GetData(), static_cast<size_t>(arr.GetSize())
        , sizeof(CString), compare_CStrings);
  // 3. Copy lines into list
  lst.RemoveAll();
  { for(int i = 0; i < arr.GetSize(); ++i)
  {
    lst.AddTail(arr[i]);
  }}
}

char const  *fileName = . . .
CStringList lines;
readOrderedLines(fileName, lines);
{ for(POSITION pos = lines.GetHeadPosition(); NULL != pos; )
{
  std::cout << "\t" << lines.GetNext(pos) << std::endl;
}}
```

Not exactly pretty, is it? And did you notice the deliberate mistake? The second cast in `compare_CStrings` is invalid, even though it compiles and runs without crashing on several systems. It works only because of a particular characteristic of the layout of `CString`, one that may not be present in `CString` in future versions of **MFC** and one that certainly does not apply to other types. If you did a copy/paste on this code for a different element type, you'd have an unhappy *runtime* experience; the compiler does not help you out here. The code should actually be as shown in Listing 24.2.

Listing 24.2 Correct Implementation of the `compare_CStrings()` Comparator Function

```
int compare_CStrings(void const* pv1, void const* pv2)
{
  CString const *str = static_cast<CString const*>(pv1);
  return str->Compare(*static_cast<CString const*>(pv2));
};
```

With that settled, let's look now at the alternative implementation in **STL** style (Listing 24.3), using the adaptor components described in this chapter and some of their peers for list containers from the **MFCSTL** library. (To make it extra-specially nice, I've also used the enhanced `ostream_iterator` described in Chapter 34.)

Listing 24.3 MFCSTL Version of `readOrderedLines()`

```
void readOrderedLines(char const *fileName, CStringList &lst)
{
  CArray_cadaptor<CStringArray> arr;
  // 1. Read lines into array
  readLines(fileName, arr);
  // 2. Sort lines
  std::sort(arr.begin(), arr.end());
  // 3. Copy lines into list
  CList_iadaptor<CStringList>   la(lst);
  la.assign(arr.begin(), arr.end());
}

char const                  *fileName = . . .
CList_cadaptor<CStringList> lines;
readOrderedLines(fileName, lines);
std::copy(lines.begin(), lines.end(),
        , stlsoft::ostream_iterator<LPCTSTR>(std::cout, "\t", "\n"));
```

I trust you'll agree, that's pretty conclusive. Clearly, there's less code, but there are most significant gains. For one thing, we didn't need to define a custom sort functionality. We just used `std::sort()`. This algorithm requires that the iterators (obtained by `begin()` and `end()`) are *random access* (or *contiguous*), and that the value type of the iterators passed to `std::sort()` is *LessThanComparable*. Since `CArray_cadaptor` preserves the contiguous nature of the `CArray` container family, and the **MFC** header files define a number of overloads of `operator <()` for various combinations of `CString` and `LPCTSTR`, it all works nicely. And the algorithm is standard (and therefore can be trusted), so there's no need to worry about mistakes like that in the first version of `compare_CStrings()`.

Furthermore, we didn't need to explicitly empty the list instance and then manually enumerate the contents of the array (`arr`) and add to the list (`lst`). We just used the `assign()` method. Similarly, in the client code, we're not exposed to **MFC**'s nonstandard idiom for enumerating elements in lists, via `GetHeadPosition()`/`GetNext()`. We just used `std::copy()` with the `lines` instance.

This version demonstrates use of both ***Class Adaptor*** (`arr` and `lines`) and ***Instance Adaptor*** patterns (`la`) . The runtime costs are zero for use of the former and merely the cost of taking a pointer to the adapted instance in the latter.

24.3 Emulating `std::vector`

Since we're adapting array classes, it's natural for us to seek to emulate `std::vector`, which we mostly succeed in achieving. Let's look at the `std::vector` interface (Listing 24.4) to guide us in the development of the adaptors.

Listing 24.4 Definition of the `std::vector` Class Template

```
// In namespace std
template< class T // The value type
        , class A = allocator<T>
        >
class vector
{
public: // Member Types
  . . . // iterator, const_reference, size_type, etc.
public: // Construction
  vector();
  explicit vector(allocator_type const& ator);
  explicit vector(size_type numInitial);
  vector(size_type numInitial, value_type const& v);
  vector(size_type numInitial, value_type const& v
                             , allocator_type const& ator);
  vector(vector<T, A> const& rhs);
  template <typename I>
  vector(I first, I last);
  template <typename I>
  vector(I first, I last, allocator_type const& ator);
  ~vector();
  allocator_type get_allocator() const;
public: // Assignment
  vector<T, A>& operator =(vector<T, A> const& rhs);
  void        assign(size_type n, value_type const &value);
  template <typename I>
  void        assign(I first, I last);
public: // Iteration
  iterator               begin();
  iterator               end();
  const_iterator         begin() const;
  const_iterator         end() const;
  reverse_iterator       rbegin();
  reverse_iterator       rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
public: // Size
  size_type size() const;
  size_type max_size() const;
  bool      empty() const;
  void      resize(size_type n);
  void      resize(size_type n, value_type value);
public: // Capacity
  size_type capacity() const;
  void      reserve(size_type numSpaces);
public: // Element Access
  reference       operator [](size_type n);
  const_reference operator [](size_type n) const;
  reference       at(size_type n);
  const_reference at(size_type n) const;
  reference       front();
  const_reference front() const;
  reference       back();
  const_reference back() const;
public: // Modifiers
  void     push_back(value_type const& value);
  void     pop_back();
  iterator insert(iterator pos, value_type const& value);
  void     insert(iterator pos, size_type n, value_type const& value);
  template <typename I>
  void     insert(iterator pos, I first, I last);
  iterator erase(iterator pos);
  template <typename I>
  iterator erase(I first, I last);
  void     swap(vector<T, A>& rhs);
  void     clear();
private: // Implementation
  . . .
private: // Member Variables
  . . .
};

// Comparison Operators
template <class T, class A>
bool operator ==(vector<T, A> const& lhs, vector<T, A> const& rhs);
. . . // and operators !=, <, <=, >, >=

// swap()
template <class T, class A>
void swap(vector<T, A>& lhs, vector<T, A>& rhs);
```

The standard stipulates that the constructor overloads are implemented via default parameters. Some implementations instead elect to define them as overloads. I've done so here so we can clearly identify which are allocator-related, for reasons that will become clear shortly.

24.4 Design Considerations

When implementing adaptors for containers, we have several factors to consider. Must the component(s) adapt one container type or a family of container types? If a family, do the containers exhibit structural conformance? Do they have (public) member types that may be used to assist in defining traits class(es)? Do they have construction/assignment/reallocation semantics that impact the use of common implementation idioms? What memory allocation scheme do they use? How much of the adaptors' interfaces are implemented in terms of that of the adapted type; how much in terms of themselves?

Let's start our search for answers to these questions with the container family's history and characteristics.

24.4.1 MFC's Array Container Family

MFC has been around since the early 1990s, so a fair amount of code predates C++ templates (and the 1998 standardization of C++), including many of the container classes. Consequently, the **MFC Array** family of containers comprises the non-template containers `CByteArray`, `CDWordArray`, `CPtrArray`, `CStringArray`, `CUIntArray`, and `CWordArray`, in addition to the later class template container `CArray`. There are similar types and a class template in each of the **List** and **Map** container families. Thus, as well as providing both ***Class Adaptor*** and ***Instance Adaptor*** functionality, our adaptor class(es) must work with non-template and template array classes alike. For brevity, I will use the identifier `CArray` to refer to all these classes, except when I say "`CArray` class template."

Thankfully, all the array classes present structurally conformant (Section 10.1) interfaces. Consider Listing 24.5, which shows the public interface of the `CArray` class template. `TYPE` is the element type, and `ARG_TYPE` is the type used by the methods when manipulating the elements. The non-template classes all follow the same pattern; for example, substituting `CString` for `TYPE` and `LPCTSTR` for `ARG_TYPE` would yield an interface virtually identical to that of the non-template `CStringArray` class. (The few slight differences are noted in the extra material for this chapter on the CD.)

Listing 24.5 Definition of the MFC `CArray` Class Template

```
template<class TYPE, class ARG_TYPE>
class CArray
  : public CObject
{
public:
  CArray();

  int GetSize() const;
  int GetUpperBound() const;
  void SetSize(int newSize, int growBy = -1) throw (CMemoryException*)
  void FreeExtra();
  void RemoveAll();

  TYPE GetAt(int nIndex) const;
  void SetAt(int nIndex, ARG_TYPE newElement);
  TYPE& ElementAt(int nIndex);

  const TYPE* GetData() const;
  TYPE* GetData();

  void SetAtGrow( int      nIndex
              , ARG_TYPE newElement) throw (CMemoryException*);
  int Add(ARG_TYPE newElement) throw (CMemoryException*);
  int Append(const CArray& src);
  void Copy(const CArray& src);

  void InsertAt(int nIndex, ARG_TYPE newElement
             , int nCount = 1) throw (CMemoryException*);
  void InsertAt(int     nStartIndex
             , CArray* pNewArray) throw (CMemoryException*);
  void RemoveAt(int nIndex, int nCount = 1);

  TYPE operator[](int nIndex) const;
  TYPE& operator[](int nIndex);
};
```

Note that the GetData() methods provide a pointer to the entire array of elements. Thus, all of the containers in the CArray family are contiguous containers (Section 2.2). Obviously, we will ensure that the adaptors are, too.

24.4.2 CArray_traits

Since we're dealing with a container family, the appropriate technique is to abstract away the array container type through the use of traits (Section 12.1), specifically, CArray_traits (Listing 24.6). Thankfully, despite a lack of suitably defined public member types, their close structural conformance saves us from what might otherwise have been a considerable headache.

Listing 24.6 Primary Template of the CArray_traits **Traits Class**

```
// In namespace mfcstl
template <class C>
struct CArray_traits;
```

Unfortunately, the **MFC** container classes do not define any member types that would make the definition of the primary template of such a traits class a straightforward matter. The only recourse is to leave the primary traits template as a declaration and to specialize and partially specialize for the various array classes, as shown in Listing 24.7 for CByteArray and CPtrArray. The specializations for CDWordArray, CUIntArray, CWordArray, and CObArray take the same form.

Listing 24.7 Specializations of `CArray_traits`

```
template <>
struct CArray_traits<CByteArray>
{
  typedef BYTE            value_type;
  typedef BYTE            arg_type;
  typedef CByteArray      array_type;
};
template <>
struct CArray_traits<CPtrArray>
{
  typedef void*           value_type;
  typedef void*           arg_type;
  typedef CPtrArray       array_type;
};
```

For `CStringArray`, I have broken with the `CStringArray` convention of using `LPCTSTR` as the argument type. Since **MFC**'s `CString` type uses reference-counted shared buffers and copy-on-write (and has done so since version 4), it is more efficient to use `CString const&` as the traits' argument type, `arg_type`, as shown in Listing 24.8.

Listing 24.8 Specialization of `CArray_traits` **for** `CStringArray`

```
template <>
struct CArray_traits<CStringArray>
{
  typedef CString         value_type;
  typedef CString const&  arg_type;
  typedef CStringArray    array_type;
};
```

The specializations are completed with the partial specialization for the `CArray` class template (Listing 24.9), which infers the member types from the `CArray` specialization itself.

Listing 24.9 Specialization of `CArray_traits` **for the MFC** `CArray` **Class Template**

```
template <class V, class A>
struct CArray_traits<CArray<V, A> >
{
  typedef V               value_type;
  typedef A               arg_type;
  typedef CArray<V, A>    array_type;
};
```

24.4.3 Design of the Array Adaptor Classes

In all the other cases of collection adaptation in the book, we're dealing with ***Façades*** over collection APIs. As such, the issue of identity of the collection hasn't arisen: Naturally enough, we expect the ***Façade*** to hide from us the (bulk of the) underlying collection (API). In the case of adapting container classes, however, this is not the case. We must consider whether we wish to provide a ***Class Adaptor***, an ***Instance Adaptor***, or both.

In my opinion, when writing general-purpose components we cannot predict (and should not prescribe) which is the dominant use form and hence should try to accommodate both. That is the situation in this case. This is a good thing for you, gentle reader, because you'll see a technique for doing so with high levels of reuse. (A little less good for me, as it's a trifle tricky to do and even more tricky to present in a clear narrative.) To achieve this reuse and consequent increased code coverage, robustness, and maintainability, we will define a base class adaptor, `CArray_adaptor_base`, from which the two adaptor templates will derive.

24.4.4 Abstract Manipulation of State

`CArray_adaptor_base` must serve the same semantics for two completely different structures, so it must not have any state of its own. Rather, we will have to provide it with a mechanism of communication with its derived types in order to get hold of the state (the underlying `CArray` instance) to manipulate. The choices are to use runtime polymorphism or to apply the ***Curiously Recurring Template Pattern*** (CRTP) for compile-time polymorphism. Since we're using a base class to share implementation rather than to provide customization of behavior, we will use the latter approach. (The fact that it has zero runtime cost has got nothing to do with it. Honest, guvna!)

24.4.5 Copy-and-Swap Idiom

One useful technique for implementing assignment is to use a combination of construction followed by swapping state, as shown in Listing 24.10 for the notional class `CopyNSwap`. It is advantageous for a number of reasons. First, it's idiomatic, so the discoverability/transparency burden on people using or maintaining your code is low. Second, it follows good *DRY SPOT* (Chapter 5) practice in reusing existing code (the constructor). Third, it removes the need for self-assignment detection—`if(&rhs != this)`—in the assignment operator. Fourth, it is strongly exception safe: If construction fails at any point and emits an exception, the instance being assigned to is unchanged. As long as the given type can provide a nonthrowing `swap()` method—almost always a simple swap of the internal member variables between the two instances concerned—copy-and-swap is a dream. The only major downside is when copying is too expensive; in such cases it may be preferable to use a less costly technique and only provide weak exception safety. No resources will be lost, but it cannot be guaranteed that the (logical) state of the instance will be unchanged if an exception is thrown.

Listing 24.10 Class Illustrating the Copy-and-Swap Idiom for Assignment

```
class CopyNSwap
{
public: // Member Types
  typedef CopyNSwap  class_type;
public: // Construction
  CopyNSwap(class_type const& rhs);
public: // Assignment
  class_type& operator =(class_type const& rhs)
  {
    class_type  r(rhs);      // Copy . . .
    r.swap(*this);           // . . . 'n' swap
    return *this;
  }
public: // Operations
  void swap(class_type& rhs) throw()
  {
    std::swap(m_buff, rhs.m_buff);
    std::swap(m_size, rhs.m_size);
  }
  . . .
private: // Member Variables
  int*    m_buff;
  size_t  m_size;
};
```

Unfortunately, the **MFC** containers make life more than a little difficult in this regard. Many of their functions are not actually exception safe. (To be fair, exception safety was not fully understood until the late 1990s.) And they do not provide any operations that directly support swap semantics. (See the extra material on the CD.)

There's another issue at play here too. By definition, an ***Instance Adaptor*** only has one kind of constructor: the one in which it acquires (the pointer or reference to) the instance being adapted. Conversely, a ***Class Adaptor*** will likely provide most/all of the constructors available in the adapted type, and perhaps add a few of its own. Each constructor will pass off most/all its functionality to its base class, the adapted type. In either case, the constructor of the adaptor is not invoked until *after* the adapted instance is fully constructed (albeit that in the latter case, the adapted instance is actually the same object, via `static_cast<adapted_type&>(*this)`). The significance of this is that copy-and-swap cannot be (directly) employed in defining assignment operations. Indeed, we'll see that it is much more straightforward to implement constructors (for the ***Class Adaptor***) in terms of the assignment operator.

24.4.6 Collection-Interface Composition

When writing ***Adaptors*** and ***Façades*** you typically have some latitude in determining which methods of the wrapper class are implemented in terms of which methods of the underlying interface. For example, we might postulate that `clear()` can be implemented in terms of `CArray::RemoveAll()`. Conversely, we might also use the canonical implementation, as described in the standard (C++-03: 23.1.1;4) of implementing it as `erase(begin(), end())`.

Wherever it is possible and does not detract from performance or exception-safety guarantees, I prefer implementing the wrapper in terms of its own interface, for a number of reasons. First, it is unit testing for free. Second, it reduces the requirements on the adapted type, making the adaptor widely applicable. Third, it makes copy/paste reuse more survivable; not that any of you gentle readers would employ copy/paste reuse, of course. Finally, it is easier to refactor separate adaptors into more general class templates and/or to isolate features that may be abstracted into policies.

Tip: Prefer implementing ***Adaptors*** and ***Façades*** in terms of their own interface, rather than that of the adapted type.

The one drawback is that it's not all that hard to find yourself in a logical cycle (and a runtime endless loop). You must not implement `assign()` in terms of `insert()` if you've already implemented `insert()` in terms of `assign()`!

24.4.7 Pedagogical Approach

Because this chapter discusses a large number of issues, some of which are quite abstruse, I'm going to take a stepwise approach based on difficulty—for both reader and author. First I'll present the `CArray_adaptor_base` class template interface and then immediately go on to define the two derived adaptor class templates `CArray_cadaptor` and `CArray_iadaptor`, so that you can understand how they are tied together. The rest of the chapter will then be devoted to examining the implementation of `CArray_adaptor_base`, starting with the straightforward and readily comprehensible element access and iteration methods, through the assignment methods, the issue of allocation optimization, and the complexities of exception safety in the mutating methods, to the downright baffling mechanism for eliciting copy-and-swap from the recalcitrant **MFC** container classes, covering issues of conformance (to `std::vector`), exception translation, and memory allocation efficiency along the way.

24.5 `mfcstl::CArray_adaptor_base` Interface

The interface of the base class template, `CArray_adaptor_base`, is shown in Listing 24.11. It takes three template parameters: `A` is the underlying array class, for example, `CObArray`; `D` is the derived class type, which will be either `CArray_cadaptor` or `CArray_iadaptor`; and `T` is the traits class type, for example, `CArray_traits<CObArray>`.

Listing 24.11 Public Interface of the `CArray_adaptor_base` Class Template

```
// In namespace mfcstl
template< typename A // Adapted MFC array class
        , typename D // Derived class, e.g., CArray_cadaptor<. . .>
        , typename T // Traits class, e.g., CArray_traits<A>
        >
class CArray_adaptor_base
{
public: // Member Types
  typedef A                                             array_type;
private:
  typedef D                                             derived_class_type;
  typedef T                                             array_traits_type;
```

```
public:
  typedef typename array_traits_type::value_type  value_type;
  typedef afx_allocator<value_type>               allocator_type;
  typedef typename allocator_type::reference      reference;
  typedef typename allocator_type::pointer        pointer;
  . . . // And size_type, difference_type, and so on.
protected: // Member Constants
  enum { growthGranularity = 16 };
public: // Underlying Container Access
  array_type&       get_CArray();
  array_type const& get_CArray() const;
protected: // Construction
  CArray_adaptor_base();
  ~CArray_adaptor_base() throw();
  allocator_type get_allocator() const;
public: // Assignment
  void assign(size_type n, value_type const& value);
  template <typename I>
  void assign(I first, I last);
public: // Size
  size_type size() const;
  bool      empty() const;
  void      resize(size_type n);
  void      resize(size_type n, value_type value);
public: // Element Access
  reference       operator [](size_type n);
  const_reference operator [](size_type n) const;
  reference       at(size_type n);
  const_reference at(size_type n) const;
  reference       front();
  reference       back();
  const_reference front() const;
  const_reference back() const;
public: // Iteration
  iterator               begin();
  iterator               end();
  const_iterator         begin() const;
  const_iterator         end() const;
  reverse_iterator       rbegin();
  reverse_iterator       rend();
  const_reverse_iterator rbegin() const;
  const_reverse_iterator rend() const;
public: // Comparison
  bool                   equal(???? const& rhs) const;
public: // Modifiers
  void      push_back(value_type const& value);
  void      pop_back() throw();
  iterator  insert(iterator pos, value_type const& val);
  void      insert(iterator pos, size_type n, value_type const& val);
  template <typename I>
  void      insert(iterator pos, I first, I last);
  iterator  erase(iterator pos) throw();
  iterator  erase(iterator first, iterator last) throw();
  void      clear() throw();
  void      swap(class_type& rhs) throw();
private: // Not to be implemented
  CArray_adaptor_base(class_type const&);
  class_type& operator =(class_type const&);
};
```

As intended, the methods and their groupings are very similar to those of `std::vector`, with just a few necessary differences. The major difference is the presence of a pair of `get_CArray()` overloads, which return references to the underlying array type. `CArray_adaptor_base` handles the differences between class and interface adaptation by abstracting away the true location of the underling `CArray` instance in the form of these helper method overloads: All the other methods of `CArray_adaptor_base` are implemented, either directly or indirectly, in terms of `get_CArray()`. This allows us to avoid using any special cases or requiring the derived classes to provide duplicated functionality that is the rightful task of `CArray_adaptor_base`.

24.6 `mfcstl::CArray_cadaptor`

Class adaptation of the `CArray` containers is provided by the `CArray_cadaptor` class template, shown in Listing 24.12. Note the *DRY SPOT* violation here in having an additional specification of the specialization of `CArray_adaptor_base` in order to declare it a friend. Alas, you can't specify `friend class parent_class_type` because some compilers can't handle it. (Yes, I have been bitten by the copy/paste demon in this circumstance in the past. Be on your guard with code like this!)

Listing 24.12 Definition of the `CArray_cadaptor` Class Template

```
// In namespace mfcstl
template< typename A // Adapted MFC array class
        , typename T = CArray_traits<A>
        >
class CArray_cadaptor
  : public A
  , public CArray_adaptor_base<A, CArray_cadaptor<A, T>, T>
{
private: // Member Types
  typedef CArray_adaptor_base<A, CArray_cadaptor<A, T>, T>
                                                 parent_class_type;
public:
  typedef typename parent_class_type::array_type  array_type;
  . . . // and value_type, allocator_type, reference, and so on.
  typedef CArray_cadaptor<A, T>                  class_type;
private: // Identity
  friend class CArray_adaptor_base<A, CArray_cadaptor<A, T>, T>;
  array_type&       get_actual_array();
  array_type const& get_actual_array() const;
public: // Construction
  CArray_cadaptor();
  explicit CArray_cadaptor(size_type n);
  CArray_cadaptor(size_type n, value_type const& value);
  CArray_cadaptor(class_type const& rhs);
  CArray_cadaptor(array_type const& rhs);
  template <typename I>
  CArray_cadaptor(I first, I last);
  ~CArray_cadaptor() throw();
public: // Assignment
  class_type& operator =(class_type const& rhs);
public: // Element Access
  reference       operator [](size_type index);
  const_reference operator [](size_type index) const;
};
```

Not much needs to be said about the member types, as it's pretty obvious that everything is coming from the base class.

24.6.1 Template Declaration and Inheritance

The `CArray_cadaptor` class template takes two template parameters. The first, `A`, is the array class that is to be adapted. The second, defaulted to `CArray_traits<A>`, is the traits type that provides the all-important member type `value_type` to the base class specialization of `CArray_adaptor_base`. So far, so good.

When we look at the inheritance relationship, things become a little less clear. The `CArray_cadaptor` class template inheritance is an example of CRTP, whereby a class derives from a template specialized with the deriving class. The behavior of the base class template depends on that of its derived class, in a manner loosely analogous to classic runtime polymorphism. The crucial difference is that the function binding is done at compile time, and there is no runtime indirection. The application of CRTP is a powerful way to combine the benefits of templates and runtime polymorphism, as we will see.

It's important to note that `CArray_cadaptor<A, . . .>` derives publicly from `A`: It *is* an array. The second parent class, `CArray_adaptor_base<A, CArray_cadaptor<A, T>, T>`, brings in all the adaptive functionality. Recall from Section 24.5 that `CArray_adaptor_base` takes three parameters: the underlying array type (`A`), the derived class type (`D`), and the traits type (`T`). In the specialization for the base class, the underlying array type is `A`, the first template parameter to `CArray_cadaptor`. The traits type is `T`, the second template parameter. The derived class type, `D`, is `CArray_adaptor<A, T>`. It looks odd to see a type being used to define itself, since at the point of its parameterization of `CArray_adaptor_base` it is an incomplete type. Here's where the magic of C++ templates demonstrates its worth: It is only when this type, in the guise of the member type `derived_class_type` of `CArray_` adaptor_base, is *used* that it must be complete. Which brings us to the mechanism of CRTP in this case.

24.6.2 Applying CRTP

Let's look at how the base and derived classes communicate. When CArray_adaptor_base was introduced, I showed the declaration for the get_CArray() overloads, in terms of which all CArray_adaptor_base's functionality is implemented. These methods are defined as shown in Listing 24.13.

Listing 24.13 Identity Methods for CArray_adaptor_base

```
public: // Identity
  A&        get_CArray()
  {
    return static_cast<derived_class_type*>(this)->get_actual_array();
  }
  A const&  get_CArray() const
  {
    return static_cast<derived_class_type const*>(this)->get_actual_
array();
  }
```

Each is implemented in terms of (the appropriate overload of) the get_actual_array() method of the *derived* class. The static_cast instructs the compiler to provide the this pointer of the derived class, which it knows how to do because, by the time these functions are (directly or indirectly) invoked, it has the complete definition of the specialization. As long as the derived class type supplies these methods, and they return references to the underlying array instance, it all works peachy. If they do not, a compile error will be emitted. This is the only place where any such C++ magic is used for the array adaptors, so the derived classes can have clear and transparent methods.

Because CArray_cadaptor *is* an array, it implements these methods simply by returning a reference to itself, as shown in Listing 24.14. Thus, all the code carried out in the base class operates on the instance itself. Although it's not strictly necessary, I've declared the methods private and made the base class a friend to avoid cluttering up the public interface.

Listing 24.14 Identity Methods for CArray_cadaptor

```
private: // Identity
  friend class CArray_adaptor_base<A, CArray_cadaptor<A, T>, T>;
  array_type&       get_actual_array()
  {
    return *this;
  }
  array_type const& get_actual_array() const
  {
    return *this;
  }
```

24.6.3 Construction

There are five non-template constructors and one constructor template, as shown in Listing 24.15. Because **MFC** does not facilitate the use of custom allocation schemes, the usual (defaulted) allocator arguments found in `std::vector` (and other standard containers) are missing from all but the first constructor.

Listing 24.15 `CArray_cadaptor` **Constructors**

```
public: // Construction
  explicit CArray_cadaptor(allocator_type const& = allocator_type())
  {
    parent_class_type::resize(0); // Sets grow increment; see below
  }
  explicit CArray_cadaptor(size_type n)
  {
    parent_class_type::resize(n);
  }
  CArray_cadaptor(size_type n, value_type const& value)
  {
    parent_class_type::assign(n, value);
  }
  CArray_cadaptor(class_type const& rhs)
  {
    parent_class_type::assign(rhs.begin(), rhs.end());
  }
  CArray_cadaptor(array_type const& rhs)
  {
    parent_class_type::assign(rhs.GetData()
                            , rhs.GetData() + rhs.GetSize());
  }
  template <typename I>
  CArray_cadaptor(I first, I last)
  {
    parent_class_type::assign(first, last);
  }
  ~CArray_cadaptor()
  {
    STLSOFT_STATIC_ASSERT(sizeof(A)
                                  == sizeof(typename T::array_type));
  }
```

Note the seemingly strange business of using public methods inside constructor bodies. This should ring alarm bells in normal circumstances because it leaves you open to using partially constructed objects to construct themselves. However, no state is maintained in the `CArray_cadaptor`; it is all in the `CArray` instance, and that is guaranteed by the language rules on construction to be fully constructed before the `CArray_cadaptor` constructor bodies are entered.

The destructor has nothing to do at runtime. This correlates directly with the fact that `CArray_cadaptor` maintains no state in and of itself. However, we use it to define a constraint that asserts that the size of the parameterizing type is the same size as the `array_type` member type of the `CArray_traits` specialization for the parameterizing type of the adaptor. (I'll explain why this restriction is important when we discuss the perverse handling of `swap()` in Section 24.16; for now, just trust me that it is.) The constraint ensures that users do not use the adaptors on types derived from one of the **MFC** array types, unless those types are the same size as their parents. In other words, this prevents us from using any derivations that provide their own additional state, in the form of member variables (which would show up in the memory footprint of the array type). Hence, this:

```
class CStringArraySameSize
  : public CStringArray
{};
mfcstl::CArray_cadaptor<CStringArraySameSize
                       ,   mfcstl::CArray_traits<CStringArray>
                       >   ar;
```

compiles okay, but this:

```
class CStringArrayBigger
  : public CStringArray
{
  int i;
};
mfcstl::CArray_cadaptor<CStringArrayBigger
                       ,   mfcstl::CArray_traits<CStringArray>
                       >   ar;
```

does not, experiencing a compile-time error in the destructor's static assertion.

24.6.4 operator []()

Because `CArray_cadaptor` derives from the array type (`A`) *and* from `CArray_adaptor_base<A, . . .>`, any methods whose names are in common between the array and the adaptor cannot be referenced unambiguously by client code. Specifically, there is a conflict between the subscript operators of the two types. Resolving this is achieved by using a *using-declaration*, as shown in Listing 24.16.

Listing 24.16 `CArray_cadaptor`**'s Subscript Operators**

```
template <. . .>
class CArray_cadaptor
  : public A                         // Defines operator []()
  , public CArray_adaptor_base<. . .> // Also defines operator []()!
{
  . . .
public: // Element Access
  using parent_class_type::operator [];
  . . .
```

24.7 `mfcstl::CArray_iadaptor`

The definition of the instance adaptor class template, `CArray_iadaptor` (Listing 24.17), is much simpler than its sibling. It has a single member variable, `m_array`, which is a reference to the underlying array instance and is used to implement the `get_actual_array()` overloads required by `CArray_adaptor_base`.

Listing 24.17 Definition of the `CArray_iadaptor` **Class Template**

```
// In namespace mfcstl
template< typename A // Adapted MFC array class
        , typename T = CArray_traits<A>
        >
class CArray_iadaptor
    : public CArray_adaptor_base<A, CArray_iadaptor<A, T>, T>
{
private: // Member Types
  typedef CArray_adaptor_base<A, CArray_iadaptor<A, T>, T>
                                                   parent_class_type;
public:
  typedef typename parent_class_type::array_type  array_type;
  . . . // And value_type, allocator_type, reference, and so on
  typedef CArray_iadaptor<A, T>                    class_type;
private: // Identity
  friend class CArray_adaptor_base<A, CArray_iadaptor<A, T>, T>;
  array_type&         get_actual_array()
  {
    return m_array;
  }
  array_type const&   get_actual_array() const
  {
    return m_array;
  }
public: // Construction
  template <typename A2>
  CArray_iadaptor(A2& array)
    : m_array(&array)
  {
    STLSOFT_STATIC_ASSERT((stlsoft::is_same_type<A, A2>::value));
  }
private: // Member Variables
  array_type& m_array;
private: // Not to be implemented
  CArray_iadaptor(class_type const&);
  class_type& operator =(class_type const&);
};
```

Both `get_actual_array()` overloads return `m_array`, a reference to the array *instance* that they're adapting.

The constraint in the constructor ensures that the type of the instance passed in is actually the same type as the specializing type and not a derived type. This is to ensure that the actions taken to ensure efficient `swap()`-ing (which we'll see shortly in Section 24.16) are well founded. It can't be placed in the destructor, which is my usual way to ensure that a constraint is applied to all instantiations, because it operates on the type `A2`, which is not available in the destructor. (Note the double round braces; without these, the `STLSOFT_STATIC_ASSERT` macro will think it's being given two parameters, `stlsoft::is_same_type<A` and `A2>::value`.)

And that's it for the derived classes. All the rest of the chapter is spent looking into the implementation of `CArray_adaptor_base`, in order to support its designated semantics.

24.8 Construction

Because `CArray_adaptor_base` has no state of its own (see Listing 24.11), and therefore no member variables, there is no direct reason to provide any construction methods. However, since it's only for use as a base class, a `protected` default constructor and a `protected` destructor are defined. The former ensures that users cannot attempt to create an instance of the adaptor, and the latter ensures that users cannot attempt to delete an adaptor instance via the specialized `CArray_adaptor_base` base type. Note that the destructor is not `virtual` so that the instance adaptor class template has a minimum footprint.

Since the class adaptor must be copyable, and the instance adaptor must not, the copy constructor and copy assignment operator of the base adaptor are proscribed in the usual manner: in a private **`Not to be implemented`** section. This forces the derived class to take the appropriate action: `CArray_cadaptor` defines these operations publicly; `CArray_iadaptor` declares them private so that error messages don't refer to the base class.

24.9 Allocator

Since **MFC** does not allow customization of the memory allocation schemes used by individual containers, providing constructor and assignment methods with an allocator parameter would be violating the Goose Rule (Section 10.1.3). Consequently, all methods of the adaptors (and the base class template) eschew any facility for the user to specify a custom allocator instance. The sole exception to this is one constructor of `CArray_cadaptor` that accepts (and then ignores) a single parameter of type allocator_type (afx_allocator), to enable it to be compatible with std::stack. (See the extra material on the CD.)

24.10 Element Access Methods

This is the low-hanging fruit of our adaptation since none of these methods affect the arrangement of the container (Section 2.2.1). All of the element access functions can be easily implemented in terms of the subscript (operator []()) overloads, which may themselves be implemented using the subscript operator overloads of the CArray interface, as shown in Listing 24.18. (For brevity, the nonmutating forms are not shown because they have syntactically identical forms.)

Listing 24.18 CArray_adaptor_base **Element Access Methods**

```
public: // Element Access
  reference operator [](size_type n)
  {
    MFCSTL_MESSAGE_ASSERT("index out of bounds", n < size());
    return get_CArray()[n];
  }
  reference at(size_type n)
  {
    if(n >= size())
    {
      throw std::out_of_range("Invalid index specified");
    }
    return (*this)[n];
  }
  reference front()
  {
    MFCSTL_MESSAGE_ASSERT("front() called on empty array", !empty());
    return (*this)[0];
  }
  reference back()
  {
    MFCSTL_MESSAGE_ASSERT("back() called on empty array", !empty());
    return (*this)[size() - 1];
  }
```

24.11 Iteration

Next we consider the implementation of the iterators. Since elements are stored contiguously, the iterator model is contiguous (Section 2.3.6), and the iterator and const_iterator types are defined as mutating and nonmutating pointers.

24.11.1 begin() and end()

As a consequence of supporting contiguous iterators, the implementations of the mutating overloads of the begin() and end() methods are very simple, as shown in Listing 24.19. Note that `GetData()` is used, rather than taking the address of the zeroth element, because `CArray<>::operator []()` asserts that the given index is less than the number of elements, which *0* clearly is not for an empty array.

Listing 24.19 `CArray_adaptor_base` **Iteration Type and Methods**

```
public: // Member Types
  . . .
  typedef value_type*                                   iterator;
  typedef value_type const*                             const_iterator;
  . . .
public: // Iteration
  iterator        begin()
  {
    return get_CArray().GetData();
  }
  iterator        end()
  {
    return begin() + size();
  }
  const_iterator  begin() const
  {
    value_type const* p1  = get_CArray().GetData();
    value_type* const p2  = const_cast<value_type* const>(p1);
    return p2;
  }
  const_iterator  end() const
  {
    return begin() + size();
  }
```

Things are not quite so clear with the nonmutating form of `begin()`, however, as a consequence of how `const` is propagated in **MFC** arrays and in standard containers. Consider the `CPtrArray` class. Its nonmutating `GetData()` method returns `void const**`—a pointer to a nonmutable pointer to `void`. Conversely, the `const_pointer` type of `std::vector <void*>` is `void* const*`—a nonmutating pointer to a pointer to `void`. The plain fact of the matter is that `std::vector` is right, and **MFC** is wrong. To provide the appropriate return type, `CArray_adaptor_base`'s nonmutating `begin()` method must use a `const_cast` to get to the correct type.

24.11.2 `rbegin()` and `rend()`

Listing 24.20 shows the reverse iterator types `reverse_iterator` and `const_reverse_iterator`, as well as the implementation of the reverse iteration method pairs `rbegin()` and `rend()`. The reverse iterator types are specializations of the `std::reverse_iterator` template. The reverse iteration methods `rbegin()` and `rend()` apply these types to adapt the return values from `end()` and `begin()`, respectively, to derive the appropriate reverse iterator instances. I've shown all definitions in full here because this is the canonical form common to almost all reversed iterators you'll ever meet. All implementations for subsequent sequences in this book (and in Volume 2) will refer back to this definition, except for special cases that don't use this form. (The only special case in this volume is the self-reversing iterator class template used for Z-order iteration in Section 26.8.)

Listing 24.20 `CArray_adaptor_base` **Reverse Iteration Type and Methods**

```
public: // Member Types
  . . .
  typedef value_type*                         iterator;
  typedef value_type const*                   const_iterator;
  typedef std::reverse_iterator<iterator>     reverse_iterator;
  typedef std::reverse_iterator<const_iterator>
                                              const_reverse_iterator;
  . . .
public: // Iteration
  reverse_iterator       rbegin() { return reverse_iterator(end()); }
  reverse_iterator       rend() { return reverse_iterator(begin()); }
  const_reverse_iterator rbegin() const
                            { return const_reverse_iterator(end()); }
  const_reverse_iterator rend() const
                           { return const_reverse_iterator(begin()); }
  . . .
```

Rule: Implement reverse iteration support for collections in terms of `std::reverse_iterator` and `begin()`/`end()`, except in very rare cases where special reverse iterator semantics are required.

24.12 Size

Listing 24.21 shows the size methods. `size()` is simply implemented in terms of the `GetSize()` method of the underlying `CArray` instance. All that's required is a `static_cast` to get from `int` to `size_type`. `empty()` is equally straightforward.

Listing 24.21 `CArray_adaptor_base` **Size Methods**

```
protected: // Member Constants
  enum { growthGranularity = 16 };
private: // Size
  static size_type calc_increment_(size_type n)
  {
    return (1 + (n / growthGranularity)) * growthGranularity;
  }
public:
  size_type size() const
  {
    return static_cast<size_type>(get_CArray().GetSize());
  }
```

```
bool      empty() const
{
  return 0 == size();
}
void      resize(size_type n)
{
  try
  {
    get_CArray().SetSize(n, calc_increment_(n));
  }
  catch(CMemoryException *px)
  {
    throw std::bad_alloc();
  }
}
void      resize(size_type n, value_type value)
{
  const size_type oldSize = size();
  resize(n);
  if(oldSize < n)
  {
    try
    {
      std::fill_n(begin() + oldSize, n - oldSize, value);
    }
    catch(...)
    {
      resize(oldSize);
      throw;
    }
  }
  MFCSTL_ASSERT(size() == n);
}
```

The first overload of `resize()` is implemented in terms of `SetSize()`. The second argument, the result of `calc_increment_()`, is used for memory optimization, described in the next subsection. Note the code that catches allocation failure of `SetSize()`, indicated by a throw of `CMemoryException*`, and translates it to a standard-conformant `std::bad_alloc`. Because we are applying an **STL** adaptation to the **MFC Array** containers, we must indicate failure in the standard way. (There's a longer discussion about this issue in the extra material for this chapter on the CD.) Throughout the rest of the material in this chapter, the comment **`// Translate Memory Exception`** means that the code is surrounded by the same `try`-`catch` logic shown in Listing 24.21.

Rule: STL adaptations must intercept nonstandard exceptions and throw the appropriate standard-conformant exception types.

The standard stipulates (C++-03: 23.2.4.2;6) that the effects of `resize()` in the case that it increases the size are equivalent to a call to `insert()`. Since the standard also states (C++-03: 23.2.4.3;1) that calls to `insert()` that involve multiple elements can change the state of the container (i.e., basic exception safety), all that's required of the second overload is the call to `std::fill_n()`. However, I prefer to be as strong as possible where it's practicable, so the `try`-`catch` ensures that the array maintains its size if any of the copy assignments of any new elements fail. Note that this is still not strongly exception safe because the call to `SetSize()` may have reallocated the underlying memory block, so any references, pointers, or iterators will be invalidated.

24.12.1 Memory Allocation Optimization

Container implementations often use flexible memory allocation schemes that attempt to optimize performance by allocating extra space in their internal data structures, to enable (some) new elements that may be added by subsequent modifying operations to be accommodated without needing to reallocate new memory. Since allocating a larger block is almost always less expensive than performing multiple allocations, this can bring significant performance savings. The `CArray` family of classes implements such a scheme and offers the user a *limited* degree of customization of the allocation strategy. Each instance has a growth increment member, `m_nGrowBy`, which is used to determine the amount to grow the *next* allocation by. The initial allocation will always be precisely the size requested.

By default, the growth increment is 0, which causes the implementation to calculate the growth increment by the relation *`min(1024, max(4, size / 8))`*; in other words, current size divided by 8, but not less than 4 and not more than 1,024. Hence, if you add one element at a time to an array whose `m_nGrowBy` is 0, you will allocate the following internal buffer sizes: 1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 46, 51, 57, 64, 72, 81, 91, 102, and so on. Now, I'm far from an expert in memory allocation, but I recognize an inefficient series of numbers when I see one. It's mitigated somewhat by the fact that **MFC** containers do not copy construct when resizing, they simply do `memmove()`, but even so this seems like far too many allocations.

The user can customize this scheme by specifying a growth increment, passed as a second argument to `SetSize()`. If we specify, say, 8 as the increment, we'll see a series of internal buffer sizes of 1, 9, 17, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 105, and so on. Clearly, this is going to be an advantage over the default scheme for small array sizes but a disadvantage at larger ones. We can move out the point at which the fixed increment size becomes a disadvantage by passing a larger growth increment. For example, specifying 16 gives the series of internal buffer sizes of 1, 17, 33, 49, 65, 81, 97, 113, and so on. But the disadvantage is that memory is wasted with small array sizes when using large increments, and too many allocations are required with large array sizes when using small or modest increments. In my opinion, using the fixed increment is a bad idea unless you have a firm notion of the maximum number of elements that a given container instance may contain in its lifetime.

Common container implementations use algorithms that increase the required size by a factor when (re)allocation is needed: 1.5 and 2 are both used. (Obviously, if the starting size is 0, the algorithms don't multiply that, since they'd get 0. So there's a minimum block size, say, 16, above which the algorithm comes into effect.) Since I'm dissatisfied with any of the options provided by `CArray`, I decided to see if I could address them in the adaptors.

There's no way to alter the internal mechanisms of the `CArray` classes, but we don't need to. We can adjust the increment as we operate on the underlying `CArray` instance through the interface of the adaptor. This is what the implementation of `resize()` does in passing the result of `calc_increment_()` to `SetSize()`. (`calc_increment_()` uses a simple integer arithmetic mechanism to ensure that the given size is always rounded up to the next increment of the constant `growthGranularity`, which is 16.) By incorporating this into `resize()`, we oblige the array instances to favor speed over space. Now the series of internal buffer sizes is 1, 17, 49, 113, and so on. The effects on the number of allocations and the consequent effects on speed are shown in Table 24.1, which contains results of a performance test that exercised the different schemes. Insertion of elements into an empty instance to a total of 10, 20, 100, 200, and 1,000 elements was repeated 100,000 times. The table shows the number of (re)allocations incurred and the time (in milliseconds) required. The compiler was Visual C++ in release mode, optimized for speed. Virtually identical results were achieved with other compilers that support **MFC**.

Table 24.1 Comparison of the Performance of Allocation Schemes for Elements Inserted One at a Time into an Empty Instance

Scheme	10		20		100		200		1,000	
	#	Time	#	Time	#	Time	#	Time	#	Time
0	4	279	6	448	19	1,730	25	2,780	39	8,838
1	10	591	20	1,185	100	6,683	200	14,560	1,000	100,553
10	2	175	3	276	11	1,253	21	2,595	101	15,518
20	2	165	2	219	6	933	11	1,914	51	10,790
`resize()`	2	193	3	334	4	1,077	5	1,985	7	8,805

You might wonder why the test focuses on the case of adding one element at a time. There are many other use cases for containers, which may well highlight more favorable characteristics for the schemes provided by **MFC**. The reason is simple: The combination of `std::copy()` and `std::back_inserter()` is a very useful and widely used one. Indeed, as we'll see later, it is a necessity for implementing some of the methods of the adaptor class; so good performance of this particular mechanism is a requirement of good performance for the adaptors as a whole.

In order to keep the increment up to date with respect to the size, the statement `resize(size())` is called before operations that add elements and after operations that remove them. It's a trifle hacky, but it works. If users manipulate the array instance outside the purview of the adaptor, for example, via `get_CArray()`, that's up to them. The worst that can happen is a little suboptimal reallocation.

It's important to note here, as in other methods that cause new elements to be added or created, that the setting of the additional values does not involve the use of copy construction, as required of implementers of `std::vector` (and other standard containers). This is because `SetSize()`, like the other additive mutating methods of the `CArray` containers, default constructs them for us.

For the majority of value types, that's likely to be of little practical consequence because any type used with a `std::vector` must be *Assignable*. Nonetheless, it is a break from the prescribed behavior and must be clearly communicated to potential users as a semantic difference. Further, when the argument type specified to the `CArray` class template is `T const&`, rather than `T`, then `T` does not have to be *CopyConstructible*.

Note: `CArray` adaptors require value types to be *DefaultConstructible* and *Assignable*, as opposed to the requirements that standard containers be *CopyConstructible* and *Assignable*.

24.13 Capacity

Now let's consider the `std::vector` methods that pertain to capacity: `capacity()` and `reserve()` (refer back to Listing 24.4). Section 23.2.4.2;5 of the standard states, "Reallocation invalidates all the references, pointers and iterators referring to the elements in the sequence. It is guaranteed that no reallocation takes place during insertions that happen after a call to `reserve()` until the time when an insertion would make the size of the vector greater than the size specified in the most recent call to `reserve()`." Uh oh!

Consider what this means for our adaptor. Client code (of the adapted specialization) may call `reserve()` with a given extent and proceed to manipulate the contents of the container instance within the bounds of that extent, certain of the promise that reallocation will not occur and references, pointers, and iterators will not be invalidated. Unfortunately, `CArray` provides us with no mechanism to achieve this end.

The `SetSize()` method looks promising since its second, defaulted parameter specifies a size to grow by, which is stored in the `m_nGrowBy` member variable. Unfortunately, two aspects of the semantics of this function make it unsuitable for supporting vector-like capacity. First, calling `SetSize(0, 100)` does not, as we might hope, result in an array of capacity 100 and length 0. The implementation of `CArray<>::SetSize()` makes a special case of `newSize == 0` to destroy the contained elements, delete the memory, and set all member variables to 0. This rules out capacity support proper, but there would be a possibility of having a rather hacky version were it not for the fact that the growth parameter is not even used when setting the size of a currently empty array; it is just remembered for later when the array allocation needs to grow. Thus, there's no way to provide implementations of `capacity()` and `reserve()` with the required semantics for an adaptation of **MFC** array classes.

If we faked `capacity()` by returning `size()`, that could lead to subtle runtime failures. Consider the case where some function `f()` is written to copy ten elements into a container, sort them, and remove the highest five. `f()` then passes off the container to another function `g()`, whose published semantics are to `push_back()` five elements onto the instance if it has sufficient `capacity()`, or to create a new instance with ten spaces, copy in the old and new five, and then `swap()` with the original instance. Unfortunately, `f()`, knowing that `g()` will not need to swap, maintains iterators into the container for some of the first five elements. `g()`, seeing that the container does not contain enough space, does the copy and swap, and returns to `f()`, which promptly crashes the process. Neither `f()` nor `g()` is to blame: It is the `CArray` adaptor.

Similar conjectures can be made for a faked `reserve()`, and with far greater ease. The only sensible approach here is to not define these two methods. This disqualifies `CArray_adaptor_base` from being compliant with `std::vector`, but does not disqualify it from being an STL container (C++-03: 23.1). (Of course, that particular issue is already moot since the value types must be *DefaultConstructible*, something not required of STL containers by clause 23.1; in fact, the standard doesn't even mention the *DefaultConstructible* concept.) We'll see later how we can permit ourselves a few expeditious transgressions of encapsulation, in order to effect a noncopying `swap()`. We *could* choose to employ a similar tactic to implement `capacity()` and `reserve()`, but I have not done so, based on the relative risks and benefits of the two cases. You might well see things differently.

24.14 Comparison

STL containers are required (C++-03: 23.1;5) to support all of the following comparisons: equality (==), inequality (!=), less than (<), less than or equal (<=), greater than (>), and greater than or equal (>=). As is my wont (Chapter 15), these will take the form of nonmember free functions that are implemented in terms of public methods. The minimum comparability for an instance of the adaptor would be with an instance of its own type, as in the following:

```
CArray_cadaptor<CStringArray> ca;

ca == ca;
ca != ca;
ca < ca;
. . . // And so on
```

This could be implemented by the use of nonmember operator functions implemented in terms of the comparison methods shown in Listing 24.22.

Listing 24.22 `CArray_adaptor_base` **Comparison Methods**

```
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    return size() == rhs.size()
                    && std::equal(begin(), end(), rhs.begin());
  }
  bool less_than(array_type const& rhs) const
  {
    return std::lexicographical_compare(begin(), end(), rhs.GetData()
                                      , rhs.GetData() + rhs.GetSize());
  }
```

`equal()` evaluates equality by a full element-by-element comparison, after checking, as an optimization, that both containers have the same number of elements. `less_than()` simply uses `std::lexicographical_compare()` to do a comparison of the two ranges. The logic's spot on, but we may be missing something. After all, consider the following situation:

```
CStringArray                    sa;
CArray_iadaptor<CStringArray> ia(sa);

ia == sa; // Compile error!
```

Since the adaptors do not hold any additional state over the adapted types, it is, in my opinion, valid and desirable to allow them to be compared with instances of the adapted type (*Principle of Least Surprise*). We might therefore add an overload for `equal()`, as shown in Listing 24.23. Note that, to avoid a lot of `static_casts`, it's simpler to work in terms of the underlying array types for both instances. (From this point on, for brevity, we're going to concentrate only on equality. You can check out the code, included on the CD, to see how the other comparisons are implemented to work between heterogeneous types.)

Listing 24.23 `CArray_adaptor_base` **Comparison Methods, with Added Overload for** `equal()`

```
public: // Comparison
  bool equal(class_type const& rhs) const;
  bool equal(array_type const& rhs) const
  {
    array_type const& lhs = this->get_CArray();
    return lhs.GetSize() == rhs.GetSize()
                        && std::equal(begin(), end(), rhs.GetData());
  }
```

Now the comparison between `CStringArray` and `CArray_iadaptor<CStringArray>` is acceptable to the compiler, and the programmer's life is a bit more straightforward. But we must go further. Consider the following code:

```
CStringArray                    sa;
CArray_iadaptor<CStringArray> ia(sa);
CArray_cadaptor<CStringArray> ca;

ia == sa; // OK
sa == ca; // OK
ia == ca; // Compile error!
```

We can compare `ia` with `sa`, and `sa` with `ca`, but not `ia` with `ca`. The current situation is perverse, to say the least; it's not likely to win friends among users. To make the comparison, the user would have to write:

```
ia == ca.get_CArray();
```

or

```
ia.get_CArray() == ca;
```

Nasty, filthy, leaky abstraction (Chapter 6). Naturally, we can do better. We simply need to redefine the first overload of `equal()` as a function template, as shown in Listing 24.24.

Listing 24.24 `CArray_adaptor_base` **Comparison Function Template Method**

```
public: // Comparison
  template <typename A2, typename I2, typename T2>
  bool equal(CArray_adaptor_base<A2, I2, T2> const& rhs) const
  {
    return size() == rhs.size()
                        && std::equal(begin(), end(), rhs.begin());
  }
```

Note the absence of overt constraints in this method. `std::equal()` acts as a moderately effective constraint in this case, ensuring that the value types of the receiving instance and `rhs` are comparable. However, this may not be tight enough for all tastes, including mine, since it facilitates comparisons between collections whose value types are implicitly comparable. For example, we would be allowed to compare an instance of `CArray_cadaptor<CUIntArray>` with an instance of `CArray_cadaptor<CArray<double, double> >`. I humbly suggest that to allow such a comparison would be a bad idea.

You might think that this is easily solved by a compile-time constraint on the size of the value types, as shown in Listing 24.25.

Listing 24.25 Possible Version of the Heterogeneous `equal()` **Method**

```
  template <. . .>
  bool equal(CArray_adaptor_base<A2, I2, T2> const& rhs) const
  {
    typedef
     typename CArray_adaptor_base<A2, I2, T2>::value_type rhs_val_t;
    STLSOFT_STATIC_ASSERT(sizeof(value_type) == sizeof(rhs_val_t));
    return size() == rhs.size()
                        && std::equal(begin(), end(), rhs.begin());
  }
```

While this is helpful—it would catch the `double`/`DWORD` case—it would still allow comparisons between, say, a (32-bit) `float` and a `DWORD`. So we also use the `is_same_type` template (Section 12.1.7), as shown in Listing 24.26. Because `is_same_type` relies on support for partial template specialization, and the `CArray` adaptors are still used with compilers that do not support it, we keep the `sizeof()` constraint and conditionally add the new one.

Listing 24.26 Actual Version of the Heterogeneous `equal()` **Method**

```
  template <. . .>
  bool equal(CArray_adaptor_base<A2, I2, T2> const& rhs) const
  {
    typedef
     typename CArray_adaptor_base<A2, I2, T2>::value_type rhs_val_t;
    STLSOFT_STATIC_ASSERT(sizeof(value_type) == sizeof(rhs_val_t));
    STLSOFT_STATIC_ASSERT(( stlsoft::is_same_type<value_type
                          , rhs_val_t>::value));
    return size() == rhs.size()
                        && std::equal(begin(), end(), rhs.begin());
  }
```

24.15 Modifiers

Let's now look at the modifiers: functions that add or remove elements from the container, thereby affecting the structure or arrangement of the container, rather than just the elements. These include `push_back()`, `pop_back()`, `clear()`, and overloads of `assign()`, `insert()`, and `erase()`.

The issue of exception safety will be to the fore for all of the additive methods. Because the **MFC** containers offer little in the way of support for exception safety themselves, we have to step in here and there to lend a helping hand. I'll show some before-and-after implementations in these cases, to highlight the measures you need to take with such adaptations.

24.15.1 push_back()

The `CArray` method `Add()` is an obvious candidate for the implementation of `push_back()`, which we'd expect to take the form shown in Listing 24.27.

Listing 24.27 Initial Version of `push_back()`

```
public: // Modifiers
  void push_back(value_type const& value)
  {
    // Translate Memory Exception . . .
    resize(size());
    get_CArray().Add(value);
  }
  . . .
```

However, the `CArray` classes default construct new element instances before they assign the new value. Thus, it's possible that the reallocation and default constructor succeed but the subsequent assignment fails, leaving the container holding an empty element it wasn't supposed to have. In order to handle this, the call to `Add()` is wrapped in a `try` block, and a `catch(...)` handler ensures that the original size is restored if it's changed, as shown in Listing 24.28.

Listing 24.28 Revised Version of `push_back()`

```
public: // Modifiers
  void push_back(value_type const& value)
  {
    const size_type oldSize = size();
    resize(size());
    try
    {
      // Translate Memory Exception . . .
      get_CArray().Add(value);
    }
    catch(...)
    {
      if(size() != oldSize)
      {
        MFCSTL_ASSERT(size() == oldSize + 1);
        resize(oldSize);
      }
      throw;
    }
  }
  . . .
```

24.15.2 assign()

As I mentioned earlier, the `assign()` overloads are full-featured methods. Consequently, they need to empty their current contents and then copy in the new ones, as shown in Listing 24.29.

Listing 24.29 `CArray_adaptor_base` **Assignment Methods**

```
public: // Assignment
  void assign(size_type n, value_type const& value)
  {
    resize(n);
    std::fill_n(begin(), n, value);
    MFCSTL_ASSERT(size() == n);
  }
  template <typename I>
  void assign(I first, I last)
  {
    clear_and_assign_(first, last);
  }
  . . .
private: // Implementation
  template <typename I>
  void clear_and_assign_(I first, I last);
```

The implementations are pretty straightforward. The non-template overload changes the size to that required and assigns the given value en bloc, using `std::fill_n()`. The template overload invokes the private helper member function template `clear_and_assign_()`. It's important to bear in mind that range methods such as this can be given iterators of different categories. `I` might be an input iterator or it might be a forward iterator or "higher," handled by selecting the appropriate worker function based on iterator category, as shown in Listing 24.30.

Listing 24.30 `CArray_adaptor_base` **Assignment Worker Methods**

```
private: // Implementation
  template <typename I>
  void clear_and_assign_(I first, I last)
  {
    clear_and_assign_(first, last
            , typename std::iterator_traits<I>::iterator_category());
  }
  template <typename I>
  void clear_and_assign_(I first, I last, std::input_iterator_tag)
  {
    clear();
    std::copy(first, last, std::back_inserter<class_type>(*this));
  }
  template <typename I>
  void clear_and_assign_(I first, I last, std::forward_iterator_tag)
  {
    resize(std::distance(first, last));
    std::copy(first, last, begin());
  }
```

A special case is made for input iterators. With other iterator categories, the size can be calculated and the current instance (which may be nonempty) resized in one operation, followed by a direct copy of the elements. With input iterators, it is not possible to calculate the size because that would use up the only pass allowed on the given range (Section 1.3.1). Instead, the instance is cleared of all its elements, and each new element in the range is added using `push_back()`, via `std::back_inserter` (Section 1.3.7). The point of delineating between input iterators and all higher refinements is based on the assumption that, for forward iterators likely to be used with the `CArray` adaptations, an extra traversal of the range to determine the distance is likely to be less costly than throwing away all storage and allocating from scratch. This assumption may not be correct for other containers. It may not even be correct for this one. This would be adjusted by using a different tag on the second overload, for example, `random_access_iterator_tag`.

Although we've nicely handled the differences between different iterator types, a problem remains: The implementations are not sufficiently exception safe. In order to make them be so, we need a `swap()` method. Since `swap()` can be achieved only through a combination of compile-time constraints, casting, and the design restrictions pertaining to `protected` members, I'm devoting a whole separate section to the issue (Section 24.16), wherein we'll see the final implementations of the `assign()` overloads.

24.15.3 `pop_back()` and `clear()`

These two modifiers are very straightforward. Because they do not add any new elements, we do not have to worry about handling strange semantics from the `CArray` instance. All we need to do is remove the appropriate elements, as shown in Listing 24.31. `resize(size())` is used to potentially drop the growth increment back to a low level.

Listing 24.31 `pop_back()` **and** `clear()` **Modifiers**

```
public: // Modifiers
  void pop_back() throw()
  {
    MFCSTL_MESSAGE_ASSERT("pop_back() called on empty array"
                        , !empty());
    get_CArray().RemoveAt(get_CArray().GetUpperBound());
    resize(size());
  }
  void clear() throw()
  {
    get_CArray().RemoveAll();
    resize(size());
  }
```

`pop_back()` is straightforward to implement, with one caveat: The standard does not define whether or not the container on which it is invoked must be nonempty. It does say that "no `erase()`, `pop_back()` or `pop_front()` method throws an exception" (C++-03: 23.1;10), which means there are only two valid choices. Either the empty container should simply ignore the call, or the precondition for the method should stipulate that the container is not empty. We'll follow the example of several standard library implementations and assume the latter, and we'll use an assertion to enforce the precondition. A check against `empty()` would be equally standards-compliant.

24.15.4 `erase()`

Again, we can rely on straightforward semantics of the `CArray` methods in this case, yielding simple implementations, as shown in Listing 24.32.

Listing 24.32 `erase()` **Modifier**

```
public: // Modifiers
  iterator erase(iterator pos) throw()
  {
    MFCSTL_ASSERT(pos == end() || (pos >= begin() && pos < end()));
    difference_type index = pos - begin();
    get_CArray().RemoveAt(static_cast<int>(index), 1);
    MFCSTL_ASSERT(pos == begin() + index);
    resize(size());
    return pos;
  }
  iterator erase(iterator first, iterator last) throw()
  {
    MFCSTL_ASSERT(first <= last);
    MFCSTL_ASSERT(first == end()
                              || (first >= begin() && first < end()));
    MFCSTL_ASSERT(last == end() || (last >= begin() && last < end()));
    difference_type index = first - begin();
    get_CArray().RemoveAt(static_cast<int>(index)
                        , std::distance(first, last));
    MFCSTL_ASSERT(first == begin() + index);
    resize(size());
    return first;
  }
```

Preconditions enforce the validity of the iterators. The **MFC** array containers use indexes, rather than iterators, to denote points of insertion and removal, so pointer arithmetic is used to calculate the element positions. Both invoke the `RemoveAt()` method of the underlying container and then return the iterator position of the element after the removal. The postcondition enforcement in each method is used to ensure that the documented behavior of `RemoveAt()`, that it "decrements the upper bound of the array but does not free memory," holds true. Finally, `resize(size())` is called to update the growth increment.

24.15.5 insert()

When it comes to inserting elements into the array, the adaptor provides three overloads of the `insert()` method corresponding to those found in `std::vector`. Listing 24.33 shows possible implementations.

Listing 24.33 `insert()` **Modifiers**

```
public: // Modifiers
  iterator insert(iterator pos, value_type const& value)
  {
    MFCSTL_ASSERT(pos == end() || (pos >= begin() && pos < end()));
    difference_type index = pos - begin();
    // Translate Memory Exception . . .
    resize(size());
    get_CArray().InsertAt(static_cast<int>(index), value, 1);
    return begin() + index;
  }
  void     insert(iterator pos, size_type n, value_type const& value)
  {
    MFCSTL_ASSERT(pos == end() || (pos >= begin() && pos < end()));
    difference_type index = pos - begin();
    // Translate Memory Exception . . .
    resize(size());
    get_CArray().InsertAt(static_cast<int>(index), value, n);
  }
  template <typename I>
  void     insert(iterator pos, I first, I last)
  {
    MFCSTL_ASSERT(first <= last);
    MFCSTL_ASSERT(pos == end() || (pos >= begin() && pos < end()));
    array_type        ar;
    CArray_iadaptor<array_type
                   , array_traits_type
                   >   arp(ar);
    // Translate Memory Exception . . .
    arp.assign(first, last);
    difference_type   index = pos - begin();
    resize(size());
    get_CArray().InsertAt(static_cast<int>(index), &ar);
  }
```

Putting aside the precondition enforcements, each is implemented in terms of the `InsertAt()` overloads, after first invoking `resize()` to set the growth increment. The implementations of the first two overloads are identical except that the number of copies of the `value_type` to insert, which is specified as the second argument to `InsertAt()`, differs and that the first returns an iterator to the element inserted. The third form is a little more involved since it uses the overload of `InsertAt()` that takes as its argument a pointer to the type being adapted. (Why one cannot pass a built-in array, or why the `CArray` instance must be passed by mutable pointer, rather than nonmutable reference, one can but wonder!) This overload is documented to insert all the elements in the source array at the given index.

In order to be able to pass a `CArray` instance to `InsertAt()`, we must create an instance, `ar`. We use the `CArray_iadaptor` instance adaptor class template to define an adaptor instance `arp` that adapts the local instance `ar`. We do this so we can use its range `assign()` function template (Section 24.15.2) on the range [`first`, `last`). There are two main reasons for this. First, by not manually enumerating the iterator range ourselves, we save on new code, thereby avoiding a new potential source of bugs. Second, and more important, `assign()` handles differences between iterator categories for us. If we had tried to do it manually, and `first` and `last` were input iterators, any use of them to calculate the array size to optimally allocate en bloc would mean that we could not then reuse the range to copy in the elements. We would either have to discriminate between iterator categories and provide separate implementations (as we did in Section 24.15.2), or we'd have to insert an element at a time and so would not be able to offer the strong exception guarantee. By defining an instance, adapting it with `CArray_iadaptor`, and using `assign()`, we avoid all these problems in one fell swoop.

As with `push_back()` and `assign()`, however, the poor support for exception safety of the **MFC** containers has to be dealt with. Although the standard allows that insertion of multiple elements cannot support strong exception safety—C++-03 23.1;10 states, "If an exception is thrown by an `insert()` function while inserting a single element, that function has no effects"—the possibility that a `CArray` instance can contain an element that is constructed but not assigned is just too messy for my tastes. Hence, each of the three `insert()` methods shown earlier contains a similar `try-catch(...)` with a `resize(oldSize)` as was shown for `push_back()` (in addition to the memory exception translation).

24.16 Assignment and `swap()`

The `assign()` implementations provided in Section 24.15.2 are not strongly exception safe. In order to make them be so, we need to be able to swap arrays. Further, we need to be able to provide a `swap()` method in order for the class to be capable of being used in composition without being unduly restricting. Both of these require being able to swap the contents of **MFC** CArray instances.

24.16.1 swap()

Alas, the **MFC** arrays provide no direct support for this functionality. Doing it in terms of copying is a bad idea in general: Coupled with the copy-and-swap idiom, it results in infinite loops. And it is always inefficient. However, there is another way. In what may be oversight, the **MFC** array classes define all their member variables as protected, rather than private. This affords us a way to effect a constant-time CArray_cadaptor::swap(). A somewhat shamefaced way, to be sure, but it is verifiably correct, so don't reach for your mail client yet. (I expound on the reasons why this is valid in the extra material for this chapter included on the CD.) Listing 24.34 shows the definition of a special veneer class that derives from its parameterizing type so that it can see the CArray member variables in order to implement a true swap().

Listing 24.34 CArray_swap() **Utility Function and Supporting Class Template**

```
// In namespace mfcstl
template <class A>
class CArray_swap_veneer
  : public A
{
public: // Member Types
  typedef CArray_swap_veneer<A>   class_type;
public: // Operations
  static void swap(class_type& lhs, class_type& rhs)
  {
    std::swap(lhs.m_pData,    rhs.m_pData);
    std::swap(lhs.m_nSize,    rhs.m_nSize);
    std::swap(lhs.m_nMaxSize, rhs.m_nMaxSize);
    std::swap(lhs.m_nGrowBy,  rhs.m_nGrowBy);
  }
};
template <class A>
void CArray_swap(A& lhs, A& rhs)
{
  typedef CArray_swap_veneer<A>   swapper_t;
  swapper_t::swap(static_cast<swapper_t&>(lhs)
                , static_cast<swapper_t&>(rhs));
}
```

CArray_swap_veneer is used in the CArray_swap() free function used inside the real assign() methods of CArray_adaptor_base() and the assignment operator of CArray_cadaptor, as shown in Listing 24.35. The first overload of assign() creates an instance of the underlying CArray type and populates it with the required number of elements. It then uses CArray_swap() to swap that instance with this, thereby providing strong exception safety.

Listing 24.35 `CArray_adaptor_base` **Assignment Methods: Revised Form**

```
public: // Assignment
  void assign(size_type n, value_type const& value)
  {
    // Translate Memory Exception . . .
    if(n > 0)
    {
      array_type  ar;
      ar.SetSize(0, calc_increment_(n));
      ar.InsertAt(0, value, static_cast<int>(n));
      CArray_swap(this->get_CArray(), ar);
    }
    MFCSTL_ASSERT(size() == n);
  }
  template <typename I>
  void assign(I first, I last)
  {
    if(empty())
    {
      try
      {
        clear_and_assign_(first, last);
      }
      catch(...)
      {
        clear();
        throw;
      }
    }
    else
    {
      array_type                  ar;
      CArray_iadaptor<array_type
                    , array_traits_type
                    >             arp(ar);
      arp.assign(first, last);
      CArray_swap(this->get_CArray(), ar);
    }
  }
  . . .
  void swap(class_type& rhs) throw()
  {
    CArray_swap(get_CArray(), rhs.get_CArray());
  }
```

The second overload is a little more complex. If the instance is `empty()`, `clear_and_assign_()` can be called directly to add the requisite range of elements. If that fails, the instance is easily restored to its previous (empty) state via a call to `clear()`. If the instance is not empty, an instance of the underlying array is constructed and adapted using the `CArray_iadaptor` instance adaptor class template, on which `assign()` is called. Although this appears to be recursion, it may not actually be so because the two types on which `assign()` have been called may involve different specializations of `CArray_adaptor_base`. Even when it is, the new instance will follow the `empty()` path through `assign()`, so an infinite loop will not occur. If the adapted instance is assigned with success, its contents are swapped with `this`; again, this provides strong exception safety.

Finally, `swap()` is used in `CArray_cadaptor`'s copy assignment operator:

```
public: // Construction
  class_type& operator =(class_type const& rhs)
  {
    class_type t(rhs);
    t.swap(*this);
    return *this;
  }
```

24.17 Summary

As this is one of the largest chapters in the book, you can expect to have learned a lot. Here's a list of what I think are the salient lessons.

- We were able to achieve a near, albeit imperfect, match to `std::vector`, for the `CArray` family of containers, in particular nearly full support for iteration, element access (including contiguous iterators), and mutating operations.
- When adapting suites of similar containers, consider whether, as in the case of the `CArray` family, there is sufficient structural conformance to support the use of traits.
- When writing container (and other) classes, bear in mind the importance of member types, not only for increasing flexibility when implementing your component but also for advertising type to facilitate future adaptation.
- Consider applying CRTP to abstract all the common implementation of ***Class Adaptor*** and ***Instance Adaptor***.
- Simulate "safe" constant-time `swap()` facilities, based on the (nonprivate) interface of the adapted type(s).
- If possible, moderate inferior inherent memory optimization schemes by intercepting memory-allocating methods.
- Try to provide enhanced comparability between instances of adaptations and instances of the adapted type(s) (the *Principle of Least Surprise*).
- If necessary, layer additional mechanisms over adapted types' mutating operations to provide exception guarantees not provided by the adapted type(s).

Here are the things we cannot do.

- Offer identical treatment of elements in terms of their movement within the array. The standard requires that elements be moved by copy construction; `CArray` uses `memmove()`. Users *must* be aware of this difference.
- Provide meaningful `capacity()` and `reserve()` methods.
- Fully integrate the exception mechanisms, although we get a workable compromise.

It behooves us to look again at the motivation: Was it worth the effort? I believe the answer is yes since we are able to increase reuse, avail ourselves of standard library facilities, and reduce the amount of coding required at the application level. Even if you disagree and think **MFC** should just be left in the too-hard-to-adapt-to-STL basket, I am confident that this chapter has thrown many issues of STL extension into the light, and that has benefits in its own right.

24.18 On the CD

The CD contains a collection of sections that have been trimmed from this chapter for brevity. Check it out for the full rationale of the memory exception translation scheme and for more comments on other aspects of the implementation.

CHAPTER 25

A Map of the Environment

Very few people are willing to change the way they work in order to make someone else's life easier.
—Ian Bicking

Trying is the first step toward failure.
—Homer Simpson, *The Simpsons*

25.1 Introduction

This chapter describes `environment_map`, the only associative collection covered in the book. It is a ***Façade*** over standard and nonstandard facilities to provide a standard interface and portable implementation with which the process environment block may be queried and manipulated in an STL-compliant manner.

25.2 Motivation

Most use of the system environment involves simply looking up the value, if any, for a variable of a given name. However, there are occasionally more involved uses. For example, one of my system tools, **nvx** (which stands for eNVironment eXpander; it's included on the CD), looks up the values of one or more environment variables. One of its more useful features is that you can specify name fragments using wildcards. For example, the command *`NVX  *WORK*`* on my main Windows system gives the following output:

```
WORK_DIR:
H:\STLSoft\Releases\current\STLSoft\unittest\build

WORK_DIR_:
H:\Dev\Products\SysTools\build
H:\STLSoft\Releases\current\STLSoft\unittest\build
H:\STLSoft\Releases\XMLSTL\STLSoft\samples\vspparser
H:\Publishing\Books\XSTL\test\Pt2_Collections\scat
H:\freelibs\openrj\current\build
H:\Publishing\Books\XSTL\test\Pt3_Iterators\index_range_test
H:\Publishing\Articles\cuj\columns\flexible-c++\

WORK_DRV:
H:
```

In order to support this functionality, **nvx** must be able to enumerate the environment list. Another use case involving more than simple lookup is in the filtering and altering of the current environment of one of my development tools, in order to spawn child processes with suitable environments. This includes searching out and removing dead *BIN* (and *PATH*, *LIB*, and *INCLUDE*) directories and specifying additional variables.

Naturally, these and other nontrivial uses of the process environment could be performed using the nonstandard and variable APIs provided by different operating systems, but that's a real pain when you are writing cross-platform components. There must be a better way.

25.3 getenv(), putenv(), setenv()/unsetenv(), and environ

The C standard (C-99: 7.20.4.5) defines the `getenv()` function for retrieving the value of an environment variable from the environment list of the current process. It has the following signature:

```
char* getenv(char const* name);
```

If `name` matches a variable currently defined in the calling process's environment list, it returns a pointer to the value associated with `name`; otherwise, it returns *NULL*. The client code shall not alter the contents of the returned string, which may be overwritten by a subsequent call to `getenv()`. The C standard defines only the mechanism for retrieving named variables: It explicitly stipulates, "The set of names and the method for altering the environment list are implementation defined" (C-99: 7.20.4.5;2).

Thus, the facilities for altering the environment list and for accessing the whole collection of environment variables vary between operating systems. This variation is manageable, however, and supports our attempt to craft a general-purpose portable environment class. I don't propose to cover all permutations here; I'll just cover the common facilities on most UNIX variants (including Linux and Mac OS X) and on Windows. There are two main mechanisms for creating, updating, and deleting individual environment variables. POSIX, SVR4, Linux, and (most compilers on) Windows support the `putenv()` function.

```
int putenv(char const* str);
```

Passing a string with the format *NAME=VALUE* to `putenv()` adds a variable called *NAME* with the value *VALUE* to the current environment, or, if *NAME* already exists, updates its current value with *VALUE*. Passing a string with the format *NAME* causes the variable of that name to be removed from the current environment. Hence:

```
putenv("PATH"); // Removes PATH, if it exists, from environment
putenv("PATH=/usr/bin:/bin"); // Enters PATH into environment
putenv("PATH=/usr/bin:/bin:/Users/matthew/bin"); // Changes PATH
```

Things are slightly complicated on Windows, where these functions are wrappers, provided by the C libraries accompanying their respective compilers, over the Windows **System Information**

API functions. Most compilers prefix the symbol with an underscore to denote its nonstandard nature. Note the format of strings to remove items is *NAME=*, not *NAME*, as in the following:

```
_putenv("PATH"); // Runtime error: EINVAL
_putenv("PATH="); // Removes PATH, if it exists, from environment
```

BSD and related systems provide the alternative functions `setenv()` and `unsetenv()`:

```
int setenv(char const* name, char const* value, int bOverwrite);
int unsetenv(char const* name);
```

The semantics are self-evident, except to say that you must pass a non-*0* `bOverwrite` to `setenv()` to change the value of an existing entry. If you pass *0* and an entry of the given name already exists, the function will return success, but the entry's value will be unchanged. Note that `unsetenv()` returns `void` in some implementations.

```
unsetenv("PATH"); // Removes PATH, if it exists, from environment
setenv("PATH", "/usr/bin:/bin", 0); // Enters PATH into environment
setenv("PATH", "/usr/bin:/Users/matthew/bin ", 0); // PATH unchanged
setenv("PATH", "/usr/bin:/Users/matthew/bin ", 1); // Changes PATH
```

For accessing the environment variable collection as a whole, there's a greater degree of unanimity. Every system I've come across (or heard about) follows UNIX in providing a global variable `environ` (or `_environ` on many Windows compilers) defined as follows:

```
extern char** environ;
```

`environ` points to an array of *NAME=VALUE* strings, terminated by a null pointer. Calls to any of the environment-mutating functions may invalidate the contents of one of the strings referenced by `environ` or the `environ` array pointer itself. But don't mistake unanimity for simplicity. The exposure of this collection via a process-global variable causes huge challenges for the enterprising STL extender, as you'll see in this chapter. We enter into the strange land of contradictory physical and logical element reference categories, bash our heads against undefined behavior hither and thither, and finally settle on the well-defined and highly useful compromise of iterator-group-bound snapshots.

25.4 `platformstl::environment_variable_traits`

When presented with nontrivial inconsistencies between underlying API variants, the tool to reach for is, of course, traits (Section 12.1). The **PlatformSTL** project defines the `environment_variable_traits` class, whose interface is shown in Listing 25.1.

Listing 25.1 Declaration of `environment_variable_traits`

```
// In namespace platformstl
struct environment_variable_traits
{
public: // Operations
  static char const** get_environ();
  static void         release_environ(char const* *environ) throw();
  static char const*  get_variable(char const* name) throw();
  static int          set_variable(char const* name
                                 , char const* value) throw();
  static int          erase_variable(char const* name) throw();
private: // Implementation
#if defined(PLATFORMSTL_ENVVAR_SET_BY_PUTENV) || \
    defined(PLATFORMSTL_ENVVAR_ERASE_BY_PUTENV)
  static int  call_putenv_(char_type const* str) throw();
  static int  call_putenv_(char_type const* name
                         , char_type const* value) throw();
#endif /* putenv ? */
};
```

get_environ() and release_environ() control access to the environment variable list or to a copy of it, as appropriate to the given operating environment. For all currently supported platforms, get_environ() simply returns the value of the environ variable, suitably const_cast, to dissuade client code from direct manipulation. The function pair is designed to allow for a copy to be allocated on demand and given to the client code, and then released in the call to release_environ(), should that be more suitable for a given platform.

get_variable() is semantically identical to getenv() and for most platforms will be a straight call to it. set_variable() and erase_variable() abstract away the different support for putenv() and setenv()/unsetenv() and between the different flavors of putenv() semantics. Note that all methods apart from get_environ() do not throw, as indicated by the throw() clauses. This provides degenerate consistency between the different underlying APIs and helps in implementing strongly exception-safe methods in the environment wrapper class.

The variations between operating environments are handled in the implementation of these functions, based on the definitions of the following preprocessor symbols:

```
PLATFORMSTL_ENVVAR_SET_BY_PUTENV
PLATFORMSTL_ENVVAR_SET_BY_SETENV
PLATFORMSTL_ENVVAR_ERASE_BY_PUTENV
PLATFORMSTL_ENVVAR_ERASE_BY_PUTENV_EQUALS
PLATFORMSTL_ENVVAR_ERASE_BY_UNSETENV
PLATFORMSTL_ENVVAR_ENVIRON_HAS_UNDERSCORE
PLATFORMSTL_ENVVAR_PUTENV_HAS_UNDERSCORE
```

I won't dig into the discrimination logic here since you can peruse the code (included on the CD) if you wish. By using these symbols, the implementation of most methods is pretty straightforward. For example, the single-parameter `call_putenv_()` helper function invokes `_putenv()` if `PLATFORMSTL_ENVVAR_PUTENV_HAS_UNDERSCORE` is defined; otherwise, it invokes `putenv()`. It's worth noting that when using `setenv()`, the third parameter is always nonzero, and that when using `unsetenv()`, the return value is assumed to be `void` to avoid compilation failures with implementations where that is so.

The one method whose implementation I want to show you is the two-parameter `call_putenv_()` (Listing 25.2), which is called by `set_variable()` and `erase_variable()` (with the second parameter *NULL*) when they're to be implemented in terms of `putenv()`.

Listing 25.2 Implementation of the `call_putenv_()` Worker Method

```
int environment_variable_traits::call_putenv_(char const* name
                                             , char const* value) throw()
{
#ifndef PLATFORMSTL_ENVVAR_ERASE_BY_PUTENV_EQUALS
  if(NULL == value)
  {
    return call_putenv_(name);
  }
#endif /* !PLATFORMSTL_ENVVAR_ERASE_BY_PUTENV_EQUALS */
  try
  {
    const size_t              cchName  = ::strlen(name);
    const size_t              cchValue = stlsoft::c_str_len(value);
    auto_buffer<char, 1024>   buff(cchName + 1 + cchValue + 1);
    ::strncpy(&buff[0], name, cchName);
    buff[cchName] = '=';
    ::strncpy(&buff[cchName + 1], value, cchValue);
    buff[cchName + 1 + cchValue] = '\0';
    STLSOFT_ASSERT(::strlen(buff.data()) == buff.size() - 1);
    return call_putenv_(buff.data());
  }
  catch(std::bad_alloc&)
  {
    errno = ENOMEM;
    return -1;
  }
}
```

When `value` is *NULL* and the implementation uses the erasing format that doesn't require an equal sign, indicating a call from `erase_variable()`, the function defers to `putenv()`, passing only the `name` parameter. Otherwise, a string must be prepared. The length of the `name` parameter is calculated by `strlen()`, but that of the `value` parameter is calculated by the **c_str_len** string access shim (Section 9.3.1) because it handles null pointers. Thus, the total length cchName + 1 + cchValue + 1 is always correct, for both insertion/update and for erasure. As is my wont, I used an auto_buffer (Section 16.2) to provide scoped, nominally local memory, and its contents are populated via strncpy() and direct element access. In the case where a total length exceeds 1,024 and results in allocation failure, the std::bad_alloc exception is caught and translated into an error code. This is not only consistent with alternate implementations of set_variable() and erase_variable() based on setenv() and unsetenv(), but also consistent within itself. The call to putenv() might fail for lack of memory after the auto_buffer instance has been successfully constructed, indicating this by return value. If we did not catch and translate the exception in this way, the client code would have to be prepared to handle both. However much you might detest error codes and love exceptions, it's fair to say that having to deal with exceptions *and* error codes is worse than having to deal with just error codes.

25.5 Planning the Interface

At first blush, writing an STL extension over the environment might seem like a simple thing. We'd want to be able to look up values by name; add, update, and delete values by name; enumerate (forward or backward) over the set of values; and look up values by index. However, because the memory is owned by the language runtime and potentially shared between multiple threads, matters are somewhat muddied. We'll take an incremental approach, dealing with these issues in turn.

25.6 Lookup by Name

In April 2005, *Dr. Dobb's Journal* published my article "C++ & operator []=", in which I discuss the flaws as I see them, in the semantics of the std::map interface and the subscript operator in particular. The main thrust is that it was a mistake to have the subscript operator be a mutating (non-const) method that inserts a default value if no corresponding element is present in the container:

```
std::map<int, int>  m;

. . .

int v = m[1]; // Inserts an entry (1 => 0) if not already in!
```

and that cannot be applied to const instances:

```
int lookup(std::map<int, int> const& m, int key)
{
  return m[key]; // Compilation error: violates const correctness!
}
```

I go on to suggest that the subscript operator should instead have been a nonmutating (const) method that throws an exception if the requested element does not exist. In the case of an environment map, I think it is reasonable to assert that the majority of use cases will involve lookup of a value, rather than insertion of a value. The first increment of the class, therefore, might look like Listing 25.3.

Listing 25.3 First Version of `environment_map`

```
// In namespace platformstl
class environment_map
{
public: // Element Access
  char const* operator [](char const* name) const
  {
    char const* value = ::getenv(name);
    if(NULL == value)
    {
      throw std::out_of_range("variable does not exist");
    }
    return value;
  }
};
```

This can be used as follows:

```
environment_map env;
std::cout << "INCLUDE=" << env["INCLUDE"] << std::endl;
```

Unfortunately, it’s very easy to use this class in a manner in which its behavior will be undefined:

```
std::cout << "INCLUDE=" << env["INCLUDE;] << std::endl
          << "LIB=" << env["LIB"] << std::endl;
```

Because of the defined semantics of `getenv()`, it is conceivable that the second invocation of the subscript operator may overwrite the buffer returned by the first. It’s the same even if they’re in separate statements:

```
char const* inc = env["INCLUDE"];
char const* lib = env["LIB"]; // inc is now invalid!

std::cout << "INCLUDE=" << inc << std::endl
          << "LIB=" << lib << std::endl; // Still undefined behavior!
```

In Section 3.3 I mentioned that there were a few cases where *transient* references, or higher, could not be provided by STL collections. This is such a case: The return value of `environment_map::operator[]()` must be *by-value temporary* (Section 3.3.5). Any attempts to provide a higher reference category fail. Please join me on an ad absurdum (but enlightening) voyage to see why. Four options suggest themselves:

1. Return a (physically) fixed/(logically) transient reference to a cached object with an up-to-date value
2. Return a fixed reference to a cached object with a snapshot value
3. Return a fixed reference to a cached object with an up-to-date value
4. Return a by-value temporary reference with an up-to-date value

25.6.1 Option 1: Return a Fixed/Transient Reference to a Cached Object with an Up-to-Date Value

In order to be able to return a genuine reference, we must store a string object corresponding to each invocation of the operator, as shown in Listing 25.4.

Listing 25.4 Alternate Version of `environment_map`**, Using Option 1**

```
class environment_map
{
public: // Member Types
  typedef std::string         string_type;
  typedef string_type const&  const_reference;
public: // Element Access
  const_reference operator [](char const* name) const
  {
    char const* value = ::getenv(name);
    . . . // Test for NULL, and throw
    m_cache.push_back(value);
    return m_cache.back();
  }
private: // Member Variables
  mutable std::list<string_type>  m_cache;
};
```

Every returned reference is valid for the duration of the lifetime of the container. Hence, the supported reference category is fixed. But there are problems. First, separate invocations for the same variable will return (references to) physically distinct referents. Although the standard does not explicitly stipulate that such logically identical references should refer to the same referents, it seems a decidedly suspect thing to not do so, and I believe it would violate common assumptions in any client code that ascribes meaning to the addresses of contained elements. Thus, the references are physically fixed but logically transient. Naturally, such a schizophrenic nature doesn't do the user any favors.

Second, the number of cached string instances increases without limit until the `environment_map` instance is destroyed. Although the authors of such a creature might caution users to avoid long-lived instances, this is, in reality, an organized memory leak.

25.6.2 Option 2: Return a Fixed Reference to a Cached Object with a Snapshot Value

Rather than creating a new instance for each invocation, we might create and store a new instance upon the first retrieval of each variable and thereafter return the cached object (Listing 25.5).

Listing 25.5 Alternate Version of `environment_map`, Using Option 2

```
class environment_map
{
public: // Member Types
  typedef std::string         string_type;
  typedef string_type const&  const_reference;
public: // Element Access
  const_reference operator [](char const* name) const
  {
    if(m_cache.end() == m_cache.find(name))
    {
      char const* value = ::getenv(name);
      . . . // Test for NULL, and throw
      m_cache[name] = value;
    }
    return m_cache[name];
  }
private: // Member Variables
  mutable std::map<string_type, string_type>  m_cache;
};
```

The problem in this case is that changes to the environment will not be reflected to client code. This reference is genuinely fixed. Updates and/or deletions will not be reflected to clients of any out-of-date `environment_map` instances, as in the following:

```
void reset_PATH()
{
  ::putenv("PATH="); // Remove PATH variable from environment proper
}

std::string  v1 = env["PATH"];
reset_PATH();
std::string  v2 = env["PATH"];

// Potential surprise: No exception thrown, and v1 == v2
```

25.6.3 Option 3: Return a Fixed Reference to a Cached Object with an Up-to-Date Value

The previous option uses an internal `std::map` instance to provide both object persistence and value caching. However, it's possible to provide just the former, wherein the map acts as a keep-alive mechanism for returned references, which are updated upon each invocation (Listing 25.6).

Listing 25.6 Alternate Version of `environment_map`**, Using Option 3**

```
class environment_map
{
public: // Member Types
  typedef std::string         string_type;
  typedef string_type const&  const_reference;
public: // Element Access
  const_reference operator [](char const* name) const
  {
    char const* value = ::getenv(name);
    . . . // Test for NULL, and throw
    return m_cache[name] = value;
  }
private: // Member Variables
  mutable std::map<string_type, string_type>  m_cache;
};
```

This implementation has the opposite problem of its predecessor: It is possible to change the values of some references out from under them:

```
void dump_message(char const* message)
{
  // Append to path, load library, and invoke
  std::string path = ::getenv("PATH");
  ::putenv(("PATH=" + path + ":./log-libs/").c_str());
  load_library_and_invoke(. . . , message);
}

std::string const&  v1    = env["PATH"];
std::string const   path  = v1;

dump_message("Apparently innocent call"); // Value of PATH now changed!

std::string const&  v2    = env["PATH"];

// Potential surprise: v1 != path
```

With STL collections, the aging of a copy of an element's value as a result of collection-mutating operations on the element is not a problem. Indeed, it's a feature! The difference between such cases and the code shown above is that the change was made *outside* the purview of the collection, using a system call that is not part of the collection's interface.

25.6.4 Option 4: Return a By-Value Temporary Reference with an Up-to-Date Value

The fourth, and simplest, solution is to employ the by-value temporary reference category and return an instance of a string class to the caller, as shown in Listing 25.7. Note that I've shown the call to `getenv()` as a call to `traits_type::get_variable()`, which is as it actually occurs in the `environment_map` class.

Listing 25.7 Alternate Version of `environment_map`, Using Option 4

```
class environment_map
{
public: // Member Types
  typedef environment_variable_traits traits_type;
  typedef std::string                 string_type;
  typedef const string_type           const_reference; // BVT
public: // Element Access
  const_reference operator [](char const* name) const
  {
    char const* value = traits_type::get_variable(name);
    . . . // Test for NULL, and throw
    return value;
  }
};
```

As with the other three options, the original multiexpression call is well defined. And because we've followed the appropriate convention for defining the `const_reference` member type for by-value temporary reference, all the following client code cases are well defined:

```
environment_map                  env;
std::string                      path1 = env["PATH"];
std::string const&               path2 = env["PATH"];
environment_map::const_reference path3 = env["PATH"];

std::cout << path1 << std::endl;
std::cout << path2 << std::endl;
std::cout << path3 << std::endl;
```

But remember that the restrictions of by-value temporary reference apply. The behavior of the following code is undefined:

```
std::string const& lookup_variable(environment_map const& env
                                  , char const*            name)
{
  return env[name];
}
```

```
environment_map     env;
std::string const&  inc = lookup_variable(env, "INCLUDE");
std::string const&  lib = lookup_variable(env, "LIB");
```

`lookup_variable()` should instead have been written as:

```
environment_map::const_reference
    lookup_variable(environment_map const& env, char const* name)
{
  return env[name];
}
```

Tip: Use collections' member types when defining functions that operate on them. In the case of by-value temporary references, the functions will automatically be well defined (in using elements by value). In the case of void references, the compiler will reject any attempt to use a return value (of type `void`) from such a function.

25.6.5 Lookup by Name: Coda

The separation between data and representation causes a leak in our abstraction (Chapter 6) in whichever implementation we choose. We have either wasted memory and nonunique referents, or out-of-date data, or potentially unexpected changes to referents, or the constraints of by-value temporary references. It's *possible* that you might wish to use a policy to select between these alternatives; I have chosen to go with the last implementation, by-value temporary references. In addition to its simplicity, there are other motivations for using this model that will be evident as we expand the capabilities of the collection in the following sections. One such motivation is that it's far easier to facilitate update operations when you do not need to ensure consistency between an underlying collection and a cache of its values in circumstances where the underlying collection is susceptible to mutation by external action.

25.7 Inserting, Updating, and Deleting Values by Name

Assuming the availability of implementations of `set_variable()` and `erase_variable()`, we can implement operations for insertion, update, and deletion by name. As discussed previously, insertion and update are largely equivalent at the API level. Similarly, for the standard name-keyed containers—`std::map`, `std::set`—these two operations are equivalent and can be expressed in the form of an `insert()` method. Coupled with an `erase()` method, we can postulate the expanded interface shown in Listing 25.8.

Listing 25.8 Modifier Methods

```
class environment_map
{
  . . . // Member Types and Element Access
public: // Modifiers
  void insert(char const* name, char const* value)
  {
    STLSOFT_ASSERT(NULL != value);
    traits_type::set_variable(name, value);
  }
  void erase(char const* name)
  {
    traits_type::erase_variable(name);
  }
  . . .
```

25.8 Iteration

Manual enumeration of the environment using the `environ` variable involves stepping along the pointer array until the null pointer is found, as shown in the C code of Listing 25.9. (Note the use of the `printf()` format `%.*s`, which allows us to avoid a gratuitous allocation purely to nul-terminate a string.)

Listing 25.9 Manual Enumeration of the Environment, Using `environ`

```
char** p;
for(p = environ; NULL != *p; ++p)
{
  char* equals = strchr(*p, '=');
  printf("name=%.*s; value=%s\n", equals - *p, *p, *p + 1);
}
```

Unfortunately, any modification of the environment, such as removal of duplicates, would be a considerable undertaking using the raw API because each change would potentially invalidate `environ`. The effort involved in such cases would be motivation enough to eschew the use of C, assuming a safe C++ class can be implemented.

25.8.1 Version 1: Contiguous Iterator

The simplest possible implementation of STL iteration functionality in `environment_map` defines the value type as `char const*` and the (nonmutating) iterator type as `char const**`—a *contiguous* iterator (Section 2.3.6). As shown in Listing 25.10, `begin()` simply returns a (suitably cast) `environ`, and `end()` returns the address of the *NULL* sentinel.

Listing 25.10 Iteration Methods

```
class environment_map
{
public: // Member Types
  char const*          value_type;
  value_type const*    const_iterator;
  . . . // Element Access and Modifiers
public: // Iteration
  const_iterator begin() const
  {
    return const_cast<char const**>(environ);
  }
  const_iterator end() const
  {
    const_iterator  it  = begin();
    for(; NULL != *it; ++it) // Advance to "end()" position
    {}
    return it;
  }
};
```

For all its simplicity, this is a truly hopeless STL extension. It scores poorly in usability because each enumerated value is still in *NAME=VALUE* format. Users are required to perform the same tokenization of the returned value as in the C client code:

```
environment_map env;
for(environment_map::const_iterator b = env.begin(), e = env.end();
    b != e; ++b)
{
  char* equals = ::strchr(*b, '=');
  ::printf("name=%.*s; value=%s\n", equals - *b, *b, *b + 1);
}
```

That's no improvement, just more typing. Worse is the fact that `end()` is not constant time. And worse still is that the `environment_map` abstraction provides no protection against the invalidation of `environ` by any calls to `putenv()` that might occur during an iteration of its variables. To be sure, that same risk is attendant in the plain C version (Listing 25.9), but in that case the user is using the raw API and can reasonably be expected to be cognizant of the relationships between the API members and the restrictions on their use. Notwithstanding the *Law of Leaky Abstractions* (Chapter 6), the purpose of abstractions is to be able to appropriately set aside knowledge. It is unreasonable to expect users to observe as many restrictions in an abstraction as in the thing abstracted.

25.8.2 Version 2: Bidirectional Iterator

We can easily address the usability of the value type by defining it to be `std::pair<std::string, std::string>`. This in turn requires that `const_iterator` be a class, something we're already familiar with from previous extensions (Sections 17.3, 19.3, and 20.5). Listing 25.11 shows an abridged version of such an implementation.

Listing 25.11 Version Supporting a Bidirectional Iterator

```
class environment_map
{
public: // Member Types
  typedef std::string                         string_type;
  typedef string_type                         first_type;
  typedef string_type                         second_type;
  typedef std::pair<const first_type, const second_type>
                                                 value_type;
  typedef ptrdiff_t                              difference_type;
  class                                          const_iterator;
  typedef const value_type                       const_reference; // BVT
  typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
public: // Iteration
  const_iterator          begin() const
  {
    return const_iterator(const_cast<char const**>(environ));
  }
  const_iterator          end() const; // Same as Listing 25.10
  const_reverse_iterator  rbegin() const;
  const_reverse_iterator  rend() const;
public:
  class const_iterator
    : public std::iterator< std::bidirectional_iterator_tag
                          , value_type, difference_type
                          , void, const value_type // BVT
                          >
  {
  public: // Member Types
    typedef const_iterator  class_type;
  public: // Construction
    const_iterator()
      : m_p(NULL)
    {}
    const_iterator(char const** p)
      : m_p(p)
    {}
  public: // Forward Iteration
    class_type& operator ++()
    {
      ++m_p;
      return *this;
    }
    class_type operator ++(int); // Canonical form (see Listing 19.8)
    const_reference operator *() const
    {
      first_type  first;
      second_type second;
      stlsoft::split(*m_p, '=', first, second);
      return value_type(first, second);
    }
  public: // Bidirectional Iteration
    class_type& operator --()
    {
      --m_p;
      return *this;
    }
    class_type operator --(int); // Canonical form
  public: // Equality
    bool equal(class_type const& rhs) const;
    . . .
  private: // Member Variables
    char const**  m_p;
  };
};
```

The associative containers `std::map` and `std::multimap` both define their value type as `std::pair<const key_type, mapped_type>`. `environment_map` follows this example, but in this case both parameterizing types are declared `const` because there's no `insert()` method that takes a value type (not yet, anyway). This prevents vacuous client code constructions such as the following:

```
environment_map::value_type v1 = *env.begin();

v1.first  = "A different name";   // Compile error
v1.second = "A different value";  // Compile error
```

Tip: Consider using `const` qualification for value types in order to constrain client code semantics to meaningful expressions.

This version of the collection has much more reasonable client code, as follows:

```
for(environment_map::const_iterator b = env.begin(); b != env.end();
    ++b)
{
  std::cout << "   " << (*b).first;          // From iterator, or . . .
  environment_map::const_reference  r = *b;
  std::cout << " = " << r.second << std::endl;  // . . . from reference
}
```

There are two other interesting aspects to the class definition. First, the types for `pointer` and `reference` in the parameterization of `std::iterator` for the `const_iterator` nested class are `void` and `const value_type`, respectively. This is in accordance with the fact that the reference category for the collection (and the iterator) is by-value temporary. This is further reflected in the fact that there is no member selection operator (`operator ->()`) and that the return type of the dereference operator (`operator *()`) is `const_reference`, which is defined as `const value_type`.

Second, the implementation of `operator *()` uses the **STLSoft** string split function `split()`, which takes a source string, a delimiter, and mutable references to two string instances into which it will write the split results. (The string type is arbitrary: It can be `std::wstring`, stlsoft::simple_string, or even a specialization of basic_string_view [see Section 27.6], in which case the splitting involves no allocation or copying at all!)

25.8.3 Version 3: Snapshot

Unfortunately, we still haven't done anything about the fragility of the iterated range with respect to changes to environ during iteration. We cannot control what code users may write within environment_map's enumeration loops, so a better solution is to take a copy of the contents of the environment in a context where nothing can intervene.

Because we're dealing with a global variable, deriving well-defined behavior in the use of the non-thread-safe environment API in a multithreaded environment is simply not possible, so the first stipulation is that all discussion hereafter pertains only to use in single-threaded contexts. The behavior of environment_map is undefined in a multithreaded process wherein one or more instances are used concurrently with each other or with any other environment-mutating operations in other threads. The challenge for the remainder of this chapter, therefore, is how to provide maximal usability in the interface of environment_map in a robust and well-defined manner in the stipulated well-defined contexts. By taking a snapshot, either in the environment_map constructor or in the begin() method, we can ensure that all intrathread use of the class is well defined.

The snapshot could be a list or vector of string pairs. Listing 25.12 shows part of the class definition with the snapshot taken in the constructor.

Listing 25.12 Version Using a Snapshot Constructor

```
class environment_map
{
public: // Member Types
  . . . // As before: string_type . . . difference_type
private:
  typedef std::vector<value_type>           variables_type_;
public:
  typedef variables_type_::const_iterator   const_iterator;
  typedef variables_type_::const_reverse_iterator
                                            const_reverse_iterator;
public: // Construction
  environment_map()
  {
    char const** p = const_cast<char const**>(environ);
    for(; NULL != *p; ++p)
    {
      first_type  name;
      second_type value;
      stlsoft::split(*p, '=', name, value);
      m_variables.push_back(value_type(name, value));
    }
  }
```

```
public: // Iteration
  const_iterator begin() const
  {
    return m_variables.begin();
  }
  const_iterator end() const
  {
    return m_variables.end();
  }
  const_reverse_iterator rbegin() const
  {
    return m_variables.rbegin();
  }
  const_reverse_iterator rend() const
  {
    return m_variables.rend();
  }
  . . .
```

Note that the member type for the container is named `variables_type_`. It's uglified with a trailing separator, in line with my convention for private names. You may wonder why I did not instead use, for example, `container_type`. The reason is that public member types imply connotations about the nature and capabilities of their owning types. We might well imagine a traits class that makes use of such a `container_type` member type—including its very presence/absence (see Section 13.4). Hence, to define this (whether `private` or `public`) when it serves no purpose to any external code is to needlessly invite incompatibility with other STL extension libraries.

Tip: Exercise care in naming internal member types, and favor an unambiguous convention that will minimize clashes with likely common names in other STL extensions.

The implementation takes the snapshot in the constructor of `environment_map`. This provides the collection with fixed references, but the downside is that changes will not be reflected in successive enumerations:

```
environment_map env;

std::for_each(env.begin(), env.end(), . . . ); // A

env.erase("PATH");
env.erase("LIB");
env.erase("INCLUDE");

std::for_each(env.begin(), env.end(), . . . ); // Same results as A!
```

We could implement the mutating methods to perform lookups on the internal snapshot, to keep the enumerable range in sync. This would be something of a false sense of security, however, because direct changes to the environment via `putenv()` (or equivalent) would still not be reflected. It would be possible to remove every variable from the process's environment but still be informed by the `environment_map` instance of the original set of variables and their values!

Alternatively, the snapshot could be taken in `environment_map::begin()` and thereby associated with the lifetime of the iterator. The problem here is that iterators are subject to certain constraints in their relationships. For example, in a *forward* iterator (or higher category), the following relationship holds for non-`end()` instances:

```
(it1 == it2) == (++it == ++it2)
```

In other words, when a forward (or higher) iterator compares equal to another, then advancing each one step will result in them still being equal. Or in *other* words: Two non-`end()` forward (or higher) iterators that are equal refer to the same position in the range over which they are iterating. Hence, given any collection `cont` that provides forward (or higher) iterators, we would expect the following code to be well defined:

```
C cont;

C::const_iterator b1 = cont.begin();
C::const_iterator b2 = cont.begin();

for(; b1 != end(); ++b1, ++b2)
{
  assert(b1 == b2);
}
```

But taking a snapshot in `environment_map::begin()` presents two incongruent problems. The version shown in Listing 25.12 is such that the iterators from two snapshots can never compare equal (because comparison is done by direct comparison of the underlying iterators of `m_variables`). This gives the following very strange relation:

```
const environment_map env;
assert(env.begin() != env.begin()); // Holds in ALL cases
```

Conversely, if the snapshot representation were such that the iterators from two snapshots could be compared (perhaps by comparison of iterators, taking account an index into the underlying snapshot), we run the real risk of violating the expected relation of forward (or higher) iterators stated earlier. At this point, you may be regretting that you started this chapter or at least suspecting that this is a fruitless exercise. Thankfully, there is an answer.

25.8.4 Version 4: Reference-Counted Snapshot

The solution to this apparent conundrum is that of a reference-counted snapshot, shared between related iterator *groups*. The first call to `begin()` (or `end()`) will create a snapshot that will be shared with any additional iterator instances created as copies of that iterator, or by further calls to `begin()` or `end()` *during the lifetime of the snapshot*. Consider the code in Listing 25.13.

Listing 25.13 Example Code Using a Snapshot Implementation Illustrating Separation of Snapshots

```
 1 environment_map env;
 2
 3 std::for_each(env.begin(), env.end(), . . . );
 4
 5 env.erase("PATH");
 6
 7 if(. . .)
 8 {
 9   environment_map::const_iterator b = env.begin();
10   environment_map::const_iterator e = env.end();
11   environment_map::const_iterator it;
12
13   it = std::find(env.begin(), env.end(), . . . );
14
15   if(env.end() == it)
16   {
17     . . .
18   }
19   ::putenv("LIB"); // Or putenv("LIB="), or unsetenv("LIB")
20 }
21
22 std::for_each(env.begin(), env.end(), . . . );
```

The iterators in the first `std::for_each` statement (line 3) are created and destroyed in that statement. By the time execution has proceeded to the call to `erase()` (line 5), the snapshot that they shared has been destroyed. Therefore, the creation of the iterator `b` (line 9) creates a new snapshot, one in which the *`PATH`* variable does not exist. When `e` is created (line 10), it shares that same snapshot, as do the iterators created in the `find` statement (line 13), including `it` when it is assigned the result of the `find` operation. The call to `putenv()` (line 19) erases the *`LIB`* variable from the environment. The snapshot is destroyed when the iterator instances go out of scope (line 20). A third snapshot is created in the second `std::for_each` statement (line 22), in which *`LIB`* will not exist. Thus, the aging issue is addressed.

Now consider the code in Listing 25.14, which searches for environment variables whose names are contained within the names of others (e.g., *`LIB`* in *`RUBYLIB`*):

Listing 25.14 Example Code Using a Snapshot Implementation to Search for Referenced Environment Variables

```
environment_map env;
for(environment_map::const_iterator b1 = env.begin();
    b1 != env.end(); ++b1)
{
  for(environment_map::const_iterator b2 = env.begin();
      b2 != env.end(); ++b2)
  {
    if( b1 != b2 &&
        (*b1).first.end() != std::search( (*b1).first.begin()
                                        , (*b1).first.end()
                                        , (*b2).first.begin()
                                        , (*b2).first.end()))
    {
      . . . // Do whatever with this correlation
    }
  }
}
```

Unlike the case with separate snapshots per `environment_map::begin()` (Section 25.8.3), this is well formed because construction of `b1` creates a snapshot that is shared in the constructor of `b2`. (The `environment_map` instance actually holds the snapshot on behalf of the iterators, discarding it only when all are destroyed.) Both loops look at the same underlying range, and the iterators obey the normal conventions in all facets of their behavior.

25.9 Final Iteration Implementation

Before we examine the implementation, let's look at the class interface for the final version of the class (Listing 25.15).

Listing 25.15 Final Definition of `environment_map`

```
// In namespace platformstl
class environment_map
{
private: // Member Types
  typedef environment_variable_traits           traits_type;
  typedef std::string                           string_type;
public:
  typedef string_type                           first_type;
  typedef string_type                           second_type;
  typedef std::pair<const first_type, second_type>
                                                value_type;
  typedef size_t                                size_type;
  typedef ptrdiff_t                             difference_type;
  typedef const value_type                      const_reference; // BVT
  class                                         const_iterator;
  typedef reverse_iterator<const_iterator>      const_reverse_iterator;
  typedef environment_map                       class_type;
public: // Construction
  environment_map();
public: // Element Access
  second_type operator [](char const* name) const;
  second_type operator [](first_type const& name) const;
  bool        lookup(char const* name, second_type& value) const;
  bool        lookup(first_type const& name, second_type& value) const;
public: // Operations
  void refresh();
public: // Modifiers
  void      insert(first_type const& name, second_type const& value);
  void      insert(char const* name, char const* value);
  size_type erase(first_type const& name);
  size_type erase(char const* name);
  void      erase(const_iterator it);
public: // Iteration
  const_iterator          begin() const;
  const_iterator          end() const;
  const_reverse_iterator  rbegin() const;
  const_reverse_iterator  rend() const;
public: // Iteration
  class const_iterator
  {
    . . . // See Listing 25.18
  };
private: // Implementation
  friend class const_iterator;
  struct snapshot
  {
    . . . // See Listing 25.24
  };
  void check_refresh_snapshot_() const;
private: // Member Variables
  mutable snapshot::ref_type  m_snapshot;
private: // Not to be defined
  environment_map(environment_map const&);
  class_type& operator =(class_type const&);
};
```

The class provides nonmutating element access methods, modifying operations (including `refresh()`) and iteration. It has two nested classes, `const_iterator` and `snapshot`, whose definitions we'll examine in the remainder of this chapter. `environment_map` maintains only a single member variable, `m_snapshot`. The class is not copyable because that would break the Goose Rule (Section 10.1.3): Two copies might yield different results.

25.9.1 Mutable Snapshot?

The `environment_map` collection provides the mutating operations `insert()` and `erase()`. As it currently stands, all this provides is an abstraction of the underlying operating system facilities, via `environment_variable_traits::set_variable()` and `environment_variable_traits::erase_variable()`. The reference-counted snapshot implementation facilitates up-to-date enumeration loops, within which the use of environment-mutating operations is well defined, but isn't there more we can do?

For the standard associative containers, the standard stipulates, "The `insert()` members shall not affect the validity of iterators and references [into] the container, and the `erase()` members shall invalidate only iterators and references to the erased elements" (C++-03: 23.1.2;8). For `std::list`, `insert()` does not invalidate any iterators or references, and `erase()` affects the validity of iterators and references to the elements removed. For `std::vector`, `erase()` affects all iterators and references to elements from the point of erasure onward, and `insert()` affects all iterators and references to elements from the point of insertion unless reallocation is required, in which case all iterators and references to all elements are invalidated. With these characteristics in mind, we might postulate the following ideal behavior for `environment_map`, or rather for its current snapshot, if it exists, in respect to changes due to `erase()` and `insert()`.

- A call to `erase()` will remove the corresponding element, if it exists, from the snapshot. (Remember that a variable could have been inserted into the environment list after the snapshot was created, and therefore a call to `erase()` might be valid even when no corresponding entry is to be found in the snapshot.)
- A call to `insert()` that overwrites an existing element should not invalidate any iterators or references to elements in the snapshot.
- A call to `insert()` that inserts a new element should not invalidate any iterators or references to elements in the snapshot.

This behavior suggests that the snapshot should not be stored as a vector, but rather as a list of string pairs, in order to facilitate the invalidation requirements. But lists are slow to search, so in order to implement `erase()` and `insert()` in an efficient manner, we use a `std::map` to navigate from the name to the element. And because it's a map, we will emulate the required characteristics of an associative container with respect to mutating operations.

25.9.2 Creating the Snapshot

A new snapshot is created when the first of a new set of iterators is created. This may be in either the `begin()` or the `end()` method, so they have similar logic (Listing 25.16). Each calls `check_refresh_snapshot_()` and then creates an instance of `const_iterator`, passing in the requisite snapshot map iterator and `m_snapshot`, whose copy in the iterator instance will hold another reference count on it. (`rbegin()` and `rend()` are implemented in the same way.)

Listing 25.16 Final Implementation of the Iteration Methods

```
environment_map::const_iterator environment_map::begin() const
{
  check_refresh_snapshot_();
  return const_iterator(m_snapshot->begin(), m_snapshot);
}
environment_map::const_iterator environment_map::end() const
{
  check_refresh_snapshot_();
  return const_iterator(m_snapshot->end(), m_snapshot);
}
```

The function of `check_refresh_snapshot_()` is to check that an up-to-date snapshot is available and, if not, to make one. Listing 25.17 shows its implementation. Because the `environment_map` instance holds a reference to the snapshot in `m_snapshot` (of type `shared_ptr<snapshot>`), a snapshot is deemed active only if its reference count is 2 or greater, that is, if it has at least one `const_iterator` using it. If the count is less than 2, either there is no snapshot (`use_count()` returns *0*) or the snapshot is stale (`use_count()` returns *1*). In either case, a new snapshot is created and the previous one, if any, is destroyed by the copy assignment operator of `shared_ptr<snapshot>`.

Listing 25.17 Implementation of the `check_refresh_snapshot_()` Worker Method

```
void environment_map::check_refresh_snapshot_() const
{
  if(m_snapshot.use_count() < 2)
  {
    m_snapshot = snapshot::ref_type(new snapshot());
  }
}
```

25.9.3 `const_iterator` Nested Class

Listing 25.18 shows the definition of `environment_map::const_iterator`. What's remarkable about this class is that there's almost nothing remarkable about it. I've included the definition of the three methods just so we can remark on the unremarkability.

Listing 25.18 Definition of `environment_map::const_iterator`

```
class environment_map::const_iterator
  : public std::iterator< std::bidirectional_iterator_tag
                        , value_type, ptrdiff_t
                        , void, const value_type // BVT
                        >
{
public: // Member Types
  typedef const_iterator  class_type;
private: // Construction
  friend class environment_map;
  const_iterator(snapshot::iterator it, snapshot::ref_type snapshot);
public:
  const_iterator();
  const_iterator(class_type const& rhs);
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    ++m_it;
    return *this;
  }
  class_type operator ++(int);
  const_reference operator *() const
  {
    return *m_it;
  }
public: // Bidirectional Iterator Methods
  class_type& operator --();
  class_type operator --(int);
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    return m_it == rhs.m_it;
  }
private: // Member Variables
  snapshot::iterator  m_it;
  snapshot::ref_type  m_snapshot;
};
```

It's all so ordinary that you'd be forgiven for wondering why we should bother having a dedicated iterator class at all. The reason becomes clear when we look at the conversion constructor:

```
const_iterator::const_iterator(snapshot::iterator it
                             , snapshot::ref_type snapshot)
  : m_it(it)
  , m_snapshot(snapshot)
{}
```

The `const_iterator` class ensures that the snapshot is shared appropriately between all related iterator instances simply by holding a reference to the cached snapshot of the environment list in `m_snapshot`, which is copied (and its reference count bumped accordingly) in copy assignment and copy construction.

25.9.4 `insert()` Method

The collection's `insert()` method takes care to be strongly exception safe, that is, if an insert operation fails, the process (well, the thread, anyway) remains in the same state it was in before the call, as shown in Listing 25.19.

Listing 25.19 Implementation of `insert()`

```
void environmentmap::insert(first_type const&  name
                          , second_type const& value)
{
  STLSOFT_MESSAGE_ASSERT("Name may not be empty", !name.empty());
  STLSOFT_MESSAGE_ASSERT("Name may not contain '='"
           , name.end() == std::find(name.begin(), name.end(), '='));
  STLSOFT_MESSAGE_ASSERT("Value may not be empty", !value.empty());

  second_type* pstr = NULL;
  if( 1 < m_snapshot.use_count() &&
      m_snapshot->lookup(name, pstr))
  {
    pstr->reserve(value.size());
    if(0 != traits_type::set_variable(name.c_str(), value.c_str()))
    {
      throw std::runtime_error("Cannot set environment variable");
    }
    m_snapshot->set(name, value);
  }
  else
  {
    if(1 < m_snapshot.use_count())
    {
      m_snapshot->insert(name, value);
    }
    if(0 != traits_type::set_variable(name.c_str(), value.c_str()))
    {
      if(1 < m_snapshot.use_count())
      {
        m_snapshot->erase(name);
      }
      throw std::runtime_error("Cannot set environment variable");
    }
  }
}
```

If a snapshot is currently being shared and the given name is present in the snapshot, the code uses the string pointer returned by `lookup()` to reserve the requisite space in the string instance holding that variable's value within the snapshot. If this fails, and an exception is thrown, the state of the map and the snapshot (shared by extant iterators) is unchanged. If it succeeds, however, the logical state is also unchanged, so this reservation of extra memory will not have to be rolled back if a later part of the operation fails. Thus, we can proceed to attempt to modify the process's environment list via `set_variable()`, and if this fails, we can bail out straightaway. If this succeeds, however, we can call `set()`, which is declared `throw()`, knowing that the operation is guaranteed to succeed as a consequence of our earlier reservation.

Tip: Where possible, obviate the need for complex exception-safety rollback logic by putting code elements into guaranteed physical states via logically agnostic operations, to guarantee the success of subsequent logically manipulative operations.

The remainder of this method, concerned with insertion of a new variable, also has exception-safety connotations. At first glance they seem quite obvious. If there's a snapshot, insert into it. If that insertion fails, an exception will be thrown and the remainder of the operation will not be carried out. If it succeeds, but the subsequent call to `set_variable` fails, we erase the previously inserted element (an operation that cannot throw) before throwing a `std::runtime_error` to the caller. Unfortunately, there's still a possible flaw in the exception safety here, as a consequence of the unfortunate semantics of `std::map`'s subscript operator. The snapshot stores its variables in an instance of `std::map<const first_type, second_type>`, called `m_variables`. We might implement `snapshot::insert()` as follows:

```
void environment_map::snapshot::insert( first_type const& name
                                      , second_type const& value)
{
  m_variables[name] = value;
}
```

Alas, this is not exception safe because there are actually two separate operations involved in this statement. First, `operator [](key_type const& key)` is invoked to insert the given key, `name`, into `m_variables`, along with a default-constructed value, and returns a mutable reference to the value. In a separate step, `value` is assigned to the returned reference. If this second step causes an exception, the changes from the first step stand and the snapshot is now in an inconsistent state. (This was another plank in my argument against the design of `std::map` and one motivation for an `operator []=` for the language.) Thankfully, the remediation in this case is very simple:

```
void environment_map::snapshot::insert( first_type const& name
                                      ,  second_type const& value)
{
  m_variables.insert(value_type(name, value));
}
```

Now the insertion of name *and* value is atomic. It either works or it doesn't. In either case, the map is left in a consistent state, and, therefore, the implementation of `environment_map::insert()` (Listing 25.19) is strongly exception safe.

Tip: Avoid `std::map::operator []` and its attendant exception-safety issues.

25.9.5 `erase()` Method

The three `erase()` overloads have interesting implementations, in that they must account for failure in a benign way. Remember that there's nothing stopping other code in our thread from changing the environment via direct calls to the APIs or via another instance of `environment_map`, for that matter. Therefore, we cannot be sure that the variable exists in the environment when we come to erase it, so the logic has to take this into account.

When erasing by name, there are two overloads. The one that does all the work, shown in Listing 25.20, takes a parameter of type `first_type`. The named variable may exist in the process environment, or it may not. And the same applies to the snapshot. Thus there is no error related to whether it exists, merely a nonzero return if it was found, and erased, from either or both places. The return of `traits_type::erase_variable()` is tested to detect failure of the underlying API (e.g., memory exhaustion), in which case an exception is thrown.

Listing 25.20 Implementation of the First `erase()` Overload

```
size_type environment_map::erase(first_type const& name)
{
  // Preconditions
  STLSOFT_MESSAGE_ASSERT("Name may not be empty", !name.empty());
  STLSOFT_MESSAGE_ASSERT("Name may not contain '='"
            , name.end() == std::find(name.begin(), name.end(), '='));
  size_type b = 0;
  if(0 != traits_type::erase_variable(name.c_str()))
  {
    throw std::runtime_error("Cannot erase environment variable");
  }
  b = 1;
  if(1 < m_snapshot.use_count())
  {
    if(m_snapshot->erase(name))
    {
      b = 1;
    }
  }
  return b;
}
```

This first overload is the main actor of the two because the `snapshot::erase()` method takes a (nonmutating) reference to `first_type`. Taking the given approach avoids the gratuitous creation of an instance of `first_type` for the `erase(first_type const&)` overload. The second overload, which takes `char const*` (Listing 25.21), simply constructs an instance of `first_type` and calls the first overload.

Listing 25.21 Implementation of the Second `erase()` Overload

```
size_type environment_map::erase(char const* name)
{
  STLSOFT_ASSERT(NULL != name);
  STLSOFT_MESSAGE_ASSERT("Name may not be empty", 0 != ::strlen(name));
  STLSOFT_MESSAGE_ASSERT("Name may not contain '='"
                        , NULL == ::strchr(name, '='));
  return erase(first_type(name));
}
```

The design decisions involved here pertain to efficiency, and you might well suggest that this is a fatuous exercise in a class that deals with a system API and caches snapshots of the entire enumerable range. In a strict sense you'd be right, but this is not really a case of premature optimization since you have only two choices. I believe that, even when it doesn't matter terribly much, it's better to prefer the more efficient one, all other things being equal.

The third overload (Listing 25.22), which takes an iterator, is functionally similar to the first. You might question the somewhat curious notion of erasing by `const_iterator`: Surely a Goose Rule violation? The alternative to this would be to provide an `iterator` member type, which would make the `erase()` method seem more logical. But the implied semantics of an `iterator` method suggest a general ability to perform mutating actions, which we've already established is not appropriate in this case. It's simply a case of the lesser evil. The only way to be clean-handed about this would be to eschew this method entirely.

Listing 25.22 Implementation of the Third `erase()` Overload

```
void environment_map::erase(const_iterator it)
{
  STLSOFT_MESSAGE_ASSERT( "No snapshot assigned, so cannot erase()"
                         , 1 < m_snapshot.use_count());
#if 0
  const first_type& name = (*it).first;
#else /* ? 0 */
  const first_type& name = (*it.m_it).first; // Avoid CUR
#endif /* 0 */
  if(0 != traits_type::erase_variable(name.c_str()))
  {
    throw std::runtime_error("Cannot erase environment variable");
  }
  m_snapshot->erase(it.m_it);
}
```

Note the highlighted lines showing the preference for the more convoluted `(*it.m_it).first` rather than `(*it).first`. This avoids a curious untemporary reference (Chapter 4), saving on an unnecessary construction of a temporary, instead providing a genuine reference.

25.9.6 `operator []()` and `lookup()`

The functionality for instantaneous lookup of variables is rounded out by the addition of a `lookup()` method, which overloads for `first_type const&` as well as `char const*`. These are shown, along with the subscript operator overloads, in Listing 25.23. In this case, the main actors are the `char const*` overloads because the implementation is in terms of the traits type methods.

Listing 25.23 Implementation of the Subscript Operator

```
second_type environment_map::operator [](char const* name) const
{
  char const* value = traits_type::get_variable(name);
  if(NULL == value)
  {
    throw std::out_of_range("variable does not exist");
  }
  return value;
}

second_type environment_map::operator [](first_type const& name) const
{
  return operator [](name.c_str());
}

bool environment_map::lookup(char const* name
                           , second_type& value) const
{
  char const* value_ = traits_type::get_variable(name);

  return (NULL == value_) ? false : (value = value_, true);
}

bool environment_map::lookup(first_type const& name
                           , second_type& value) const
{
  return lookup(name.c_str(), value);
}
```

You may wonder whether we should return values from the snapshot in circumstances where a snapshot is active. I chose not to do so, partly to avoid the additional implementation complexity, but primarily so that the user has a single semantic to understand: The subscript operators and lookup methods always give a “fresh” look at the process environment.

25.9.7 `snapshot` Nested Class

The bulk of the implementation of the `snapshot` class (Listing 25.24) is straightforward. The constructor uses the `get_environ()` and `release_environ()` methods from `environment_variable_traits`, enumerating the array and calling `split()` on each entry to deduce name and value, then inserting them into m_variables. The begin(), end(), and both erase() overloads are directly implemented in terms of m_variables. Since the only reason for a snapshot is to control the apparent aging of the environment list, this shouldn't be too much of a surprise.

Listing 25.24 Definition of the snapshot Nested Class

```
struct environment_map::snapshot
{
public: // Member Types
  typedef stlsoft::shared_ptr<snapshot>           ref_type;
  typedef std::map<const first_type, second_type> variables_type_;
  typedef variables_type_::iterator               iterator;
public: // Construction
  snapshot();
public: // Operations
  bool erase( first_type const&   name) throw();
  void erase( iterator            it) throw();
  void insert(first_type const&   name
            , second_type const&  value);
  void set(   first_type const&   name
            , second_type const&  value);
  bool lookup(first_type const&   name
            , second_type*&       pvalue) throw();
public: // Iteration
  iterator  begin();
  iterator  end();
private: // Member Variables
  variables_type_   m_variables;
};
```

However, two methods are of interest, in terms of how they must be implemented to support the strong exception safety of environment_map::insert(). The first is environment_map::snapshot::insert(), which was discussed in Section 25.9.4. The second method, environment_map::snapshot::set() (Listing 25.25), has a related constraint: It must not cause any reallocation. It's called when the snapshot must have been put into a state where the value will not attempt reallocation and therefore will not throw. Thus, the precondition checks this before a new value is assigned. The assertions verify that our assumptions about how environment_map::insert() is called hold true.

Listing 25.25 Implementation of the snapshot::set() Method

```
void environment_map::snapshot::set(first_type const&   name
                                  , second_type const&  value)
{
  variables_type_::iterator it= m_variables.find(name);
  STLSOFT_ASSERT(m_variables.end() != it);
  STLSOFT_ASSERT((*it).second.capacity() >= value.size());
  (*it).second.assign(value);
}
```

25.10 Heterogeneous Reference Categories?

This implementation raises a curious issue about the reference categories. The reference category from the by-name subscript operator is by-value temporary. For the iterators, we might infer that, because there is a tangible underlying collection in the snapshot, the reference category could be *invalidatable*. However, because the snapshot can be destroyed as a result of nonmutating operations on the collection—that is, when the last reference-holding iterator goes out of scope—the reference category must be transient.

Having different element reference categories for the collection's methods and its iterators would be a strange position to be in. Thankfully, we don't have to address that conundrum *in this case* because the very nature of the snapshot makes this unappealing for use cases. Consider the following code, assuming a transient reference:

```
{
  environment_map env; // We are assuming this is not empty

  environment_map::value_type const& value = *env.begin();
  env.begin();
  std::cout << value.first << "=" << value.second << std::endl;
}
```

That innocent-looking code is not well formed because the second call to `env.begin()` would cause the `env` instance to discard the snapshot created in the first call for the temporary iterator also created *and destroyed* in that statement. Thus, we must eschew transient reference categorization for the iteration and adopt by-value temporary reference support for the `environment_map` as a whole.

25.11 `size()` and Subscript by Index

In principle, the environment variables should be amenable to linear access, in the form of the `size()` and `operator []( size_type)` methods. To do this, however, `size()` would have to be implemented by a traversal of the `environ` array on each call, as we have seen for the implementation of the `end()` iterator. Similarly, each invocation of the subscript operator would, at minimum, require a precondition test that the given index is within `size()`. Furthermore, it's arguable that we might instead opt for semantics like `basic_string`'s `at()`, given the mutability of the environment information storage.

All in all, the nonconstant time access for both `size()` and subscript operators combined with the likely lack of need to manipulate environment variables in this way suggests that it's not worth the effort. The **PlatformSTL** version of `environment_map` does not provide these methods. Consequently, despite being able to produce bidirectional iterators, the `environment_map` collection is not able to implement `size()`. There seems to be no end to the variations on the *STL container* concept when implementing STL extension collections!

25.12 Summary

You may feel that the hassle in determining the subscript return type was quite enough to put this in the too-hard basket, not to mention the apparent complexity of the iterator. However, remember that the justification for this extension, as for all extensions, is to make the life of users simpler. There can be no doubt that working with a familiar interface, one that has sensible semantics and supports succinct and simple client code, is worth the effort, as long as the constraints placed on the users are not onerous. Listing 25.26 shows an extract from the platform-independent **nvx** tool mentioned earlier (Section 25.2), where `environment_map` is used in combination with the **shwild** library to search for environment variables by wildcard. Writing this with the native API(s) would be much less fun.

Listing 25.26 Use of `environment_map` in the Environment Search Tool

```
using platformstl::environment_map;
std::string const &var         = . . . // Variable to search for
unsigned          matchFlags   = . . . // Match flags
environment_map   ENV;

fastformat::fmt(stdout, "\n{0}:\n", var);
{ for(environment_map::const_iterator b = ENV.begin(); b != ENV.end();
      ++b)
{
  if(shwild::match(var.c_str(), (*b).first.c_str(), matchFlags))
  {
    fastformat::fmt(stdout, "\n{0}:\n", (*b).first);
    . . .
```

What we've achieved with `environment_map` covers most common use cases.

- All intrathread use of the class is well defined, even in the context of extraclass (i.e., direct) manipulation of the process environment variable list.
- We maintain coherent views of the data between related iterator instances.
- We can write correct code that enumerates multiple iterator pairs concurrently.
- We employ measured obsolescence: The data is aged after all related iterators are destroyed, to prevent overly stale views of the data.
- We've minimized, as much as (I think) is possible, the violation of the Goose Rule: The element lookup and mutating operations operate on the actual process environment, and the iterators are as "fresh" as they can be while still supporting sensible adherence to the STL iterator rules.

25.13 On the CD

The CD contains an intermezzo describing some argument-dependent lookup issues of a couple of compilers that are precipitated by `environment_map::iterator`, as well as a way to work around the problem.

CHAPTER 26

Traveling Back and Forth on the Z-Plane

If you want to serve the age, betray it.

—Brendan Kennelly

Go! Confront the problem!! Fight!!! Win!!!!

—Edna Mole, *The Incredibles*

26.1 Prologue

I wrote this chapter late in the business of writing this book, after the completion of all of Parts I and III and most of Part II. I'd had it planned for a long time and was keen to write it, if only for the opportunity for another lame pun with its title. But I was concerned that the subject matter was not particularly interesting, and, from the perspective of my trajectory (after all the complexities of iterator adaptations [Chapters 36, 38, and 41], COM [Chapters 28 and 30], external iterator invalidation [Chapter 33], and so on), I was concerned that it would not be challenging and that you, gentle readers, would greet it as mere pedestrian filler.

Thankfully, I could not have been more wrong. More than anything else, this chapter reveals the fragility of many assumptions in STL or, rather, on the part of people who, like your humble author, accord themselves a degree of expertise in STL extension, that expertise is in constant need of upgrading. I learned a great many things during the research for this chapter, most of which brought the blood rushing to my cheeks in embarrassment at what I had previously released in the public domain as ready software. What is presented here, therefore, has been annealed in the examination by someone who himself has learned a whole lot more than expected in writing one and a bit books on STL extension.

26.2 Introduction

The Windows **USER** API provides myriad functions for creating and manipulating windows. There are a number of relationships in which individual windows participate to implement convincingly the virtual world that we come to believe they inhabit. Windows are hidden, shown, minimized, maximized, restored, resized, and moved. Windows are related hierarchically, where top-level frame windows and dialog boxes have child windows, which may themselves have child windows, wherewith the groups of related windows may be controlled en bloc. Another relationship, which is the subject of this chapter, pertains to peer windows—windows that share a level in a given hierarchy—and their relative placement in the Z-plane. A window that precedes its peer in the Z-order will be obscured by it if their screen areas overlap. This chapter describes two relatively simple components, `winstl::window_peer_sequence` and `winstl::child_window_sequence`, which can be used to enumerate a set of window peers in their Z-order, along with the iterator class, `winstl::zorder_iterator`, which they use to handle their iteration.

Consider the case where you want to enumerate the top-level windows on your Windows machine. (If you don't own a Windows machine, you'll just have to imagine all those busy little 1s and 0s whizzing around, giving that Intel chip a harder workout than it ever dreamed of as it was being stamped out of wafer at the factory.) To do this using the Windows API, you would undoubtedly use the `GetWindow()` function, which looks like this:

```
HWND GetWindow(HWND hwndFrom, UINT searchType);
```

The `hwndFrom` parameter is the window relative to which the returned window will be retrieved, according to the `searchType` parameter. When searching the Z-order of a group of peer windows, `searchType` can be *`GW_HWNDFIRST`* to retrieve the first peer, *`GW_HWNDLAST`* to retrieve the last peer, *`GW_HWNDNEXT`* to retrieve the next peer, or *`GW_HWNDPREV`* to retrieve the previous peer. There are also flags for retrieving the first child window of a given window (*`GW_CHILD`*) and retrieving the owner of a given window (*`GW_OWNER`*).

Using `GetWindow()`, we can enumerate all top-level windows and print out their window text, as shown in Listing 26.1.

Listing 26.1 Enumeration of Top-Level Windows Using the Windows API

```
void long_hand()
{
  HWND                          hwndDesktop = ::GetDesktopWindow();
  HWND                          hwnd;
  stlsoft::auto_buffer<TCHAR> buff(100);
  for(hwnd = ::GetWindow(hwndDesktop, GW_CHILD);
      NULL != hwnd;
      hwnd = ::GetWindow(hwnd, GW_HWNDNEXT))
  {
    buff.resize(1 + ::GetWindowTextLength(hwnd));
    ::GetWindowText(hwnd, &buff[0], buff.size());
    buff[buff.size() - 1] = '\0';
    printf("hwnd=0x%08x; text=%s\n", hwnd, buff.data());
  }
  if(ERROR_INVALID_WINDOW_HANDLE == ::GetLastError())
  {
    printf("Search interrupted due to window destruction\n");
  }
}
```

Listing 26.2 shows how this would look if we used `window_peer_sequence`.

Listing 26.2 Enumeration of Top-Level Windows Using `window_peer_sequence`

```
void short_hand()
{
  using winstl::window_peer_sequence;
  try
  {
    HWND                        hwndDesktop = ::GetDesktopWindow();
    window_peer_sequence        wps(::GetWindow(hwndDesktop,GW_CHILD));
    stlsoft::auto_buffer<TCHAR> buff(100);
    for(window_peer_sequence::iterator b = wps.begin(); b != wps.end();
        ++b)
    {
      buff.resize(1 + ::GetWindowTextLength(*b));
      ::GetWindowText(*b, &buff[0], buff.size());
      buff[buff.size() - 1] = '\0';
      printf("hwnd=0x%08x; text=%s\n", *b, buff.data());
    }
  }
  catch(stlsoft::iteration_interruption& x)
  {
    printf("Search interrupted: %s\n", x.what());
  }
}
```

Unlike the case with other STL extensions, there's no apparent saving of client code here; in fact, the second version has more lines than the first one. This is because `GetWindow()` is a syntactically concise API, due to the fact that it deals in window handle values. Nonetheless, there are three main advantages to defining an STL collection for enumerating Z-order peers. First, the calls to `GetWindow()` are hidden and thereby proof from programming error. Indeed, we could have one less line of code and one less invocation of an API function had we used the related sequence `child_window_sequence`, as shown in Listing 26.3. (Except that this sequence provides iteration over the children of a given window, rather than its peers, it is identical to `window_peer_sequence`. I'm going to focus on `window_peer_sequence` for now, but we'll see how these two can be merged later.)

Listing 26.3 Enumeration of Top-Level Windows Using `child_window_sequence`

```
void short_hand()
{
  using winstl::child_window_sequence;
  try
  {
    child_window_sequence       cs(::GetDesktopWindow());
    stlsoft::auto_buffer<TCHAR> buff(100);
    for(child_window_sequence::iterator b = cs.begin(); b != cs.end();
        ++b)
    {
      . . .
```

The second advantage is that external iterator invalidation—whatever *that* is (it's explained later, in Section 26.5)—is automatically detected and reflected in a thrown exception. The final advantage is that we can change the code to enumerate in reverse order just by making simple adjustments to the `for` statement.

```
for(window_peer_sequence::reverse_iterator b = wps.rbegin();
    b != wps.rend(); ++b)
```

Although there are three changes involved, all three must be done correctly or the compiler will issue an error. This is quite robust: It's the *Principle of Most Surprise* at compile time, which is almost always a good thing. Contrast this with the corresponding changes to the manual version:

```
for(hwnd = ::GetWindow(::GetWindow(hwndDesktop, GW_CHILD)
                     , GW_HWNDLAST);
    NULL != hwnd;
    hwnd = ::GetWindow(hwnd, GW_HWNDPREV))
```

If the additional call to `GetWindow()` is omitted or its search type is incorrectly specified, or the search type of the second existing `GetWindow()` call is not changed correctly, the programmer will not find out until runtime.

26.3 Version 1: Forward Iteration

Since this is the first extension collection that highlights the *bidirectional iterator* concept (Section 1.3.4), I'm going to use that old author's trick and show incorrect versions first, in order to illustrate the particular requirements of implementing bidirectional semantics correctly. Nonetheless, I will start with the basic architecture of these components as they appear in **WinSTL**: The `window_peer_sequence` and the `child_window_sequence` are implemented in terms of a separate iterator class, `zorder_iterator`.

26.3.1 `zorder_iterator`, Version 1

The semantics of `GetWindow()` suggest that an iterator would need only to hold a window handle, be returned by the dereference operator, and be passed, with *GW_HWNDNEXT*, to `GetWindow()` to implement the increment operator. Thus, we might imagine `zorder_iterator` to be defined as shown in Listing 26.4.

Listing 26.4 Definition of `zorder_iterator`

```
// In namespace winstl
class zorder_iterator
  : public std::iterator< std::forward_iterator_tag
                        , HWND, ptrdiff_t
                        , void, HWND  // BVT
                        >
{
```

```
public: // Member Types
  typedef zorder_iterator   class_type;
public: // Construction
  zorder_iterator()
    : m_hwnd(NULL)
  {}
  explicit zorder_iterator(HWND hwnd)
    : m_hwnd(hwnd)
  {}
public: // Forward Iterator Methods
  HWND operator *() const
  {
    return m_hwnd;
  }
  class_type& operator ++()
  {
    WINSTL_ASSERT(NULL != m_hwnd);
    m_hwnd = ::GetWindow(m_hwnd, GW_HWNDNEXT);
  }
  class_type& operator ++(int); // Canonical form
  bool equal(class_type const& rhs) const
  {
    return m_hwnd == rhs.m_hwnd;
  }
private: // Member Variables
  HWND  m_hwnd;
};
```

As defined by the types specified to the specialization of `std::iterator`, `zorder_iterator` exhibits the *by-value temporary* element reference category. In this case, there's little to put between supporting *invalidatable* and by-value temporary element reference categories (Section 3.3) because the value type is the iteration state. I opted for by-value temporary because I think it's cleaner to return `HWND` than `HWND&` since the latter doesn't make a whole lot of sense. In Section 26.5, when we look at external iterator invalidation, we'll see how this choice of element reference category is significant.

26.3.2 `window_peer_sequence`, Version 1

Given this definition of `zorder_iterator`'s interface, we could implement `window_peer_sequence` as shown in Listing 26.5.

Listing 26.5 Initial Implementation of `window_peer_sequence`

```
// In namespace winstl
class window_peer_sequence
{
public: // Member Types
  typedef zorder_iterator          iterator;
  typedef window_peer_sequence     class_type;
public: // Construction
  window_peer_sequence();
  explicit window_peer_sequence(HWND hwnd)
    : m_hwnd(hwnd)
  {}
public: // Iteration
  iterator        begin() const
  {
    return iterator(::GetWindow(m_hwnd, GW_HWNDFIRST)); // First peer
  }
  iterator        end() const
  {
    return iterator();
  }
public: // Size
  bool  empty() const;
private: // Member Variables
  HWND  m_hwnd;
};
```

This definition fulfills the requirements of Listing 26.2 and gives the required behavior. However it doesn't support reverse iteration. For that we need bidirectional semantics.

26.4 Version 2: Bidirectional Iteration

`GetWindow()` is a bidirectional API, so at first blush it seems that we should easily be able to have `zorder_iterator` implement bidirectional iterator semantics. However, it's not quite as straightforward as it sounds. One of the features of a bidirectional iterator is that an endpoint instance can be decremented, and becomes dereferenceable, as long as the underlying range is non-empty. In other words, the following is well defined:

```
some_container            cont;
some_container::iterator  b = cont.begin();
some_container::iterator  e = cont.end();

if(b != e)
{
  --e;  // This is meaningful . . .
  *b;   // . . . and so is this
}
```

This causes a little complication. Note from Listing 26.5 that m_hwnd is the only member variable. At the endpoint, this will be *NULL*. Given that, there's no way for the decrement operator to be able to move back to the last element in the range since to do so there needs to be a valid window handle to pass (along with *GW_HWNDLAST*) to GetWindow(). Thus, a bidirectional zorder_iterator needs to maintain two member variables: one for the current context and one as a reference with which to get the last element. Listing 26.6 shows the updated interface.

Listing 26.6 Adjusted Version of zorder_iterator

```
class zorder_iterator
  : public std::iterator< std::bidirectional_iterator_tag
                        , HWND, ptrdiff_t, void, HWND>
{
  . . .
public: // Construction
  zorder_iterator(HWND hwnd, HWND hwndRoot)
    : m_hwnd(hwnd)
    , m_hwndRoot(hwndRoot) // Used to get to last window
  {}
public: // Bidirectional Iterator Methods
  HWND operator *() const;
  class_type& operator ++();
  class_type& operator ++(int); // Canonical impl
  class_type& operator --()
  {
    if(NULL != m_hwnd)
    {
      m_hwnd = ::GetWindow(m_hwnd, GW_HWNDPREV);
    }
    else
    {
      m_hwnd = ::GetWindow(m_hwndRoot, GW_HWNDLAST);
    }
  }
  class_type& operator --(int); // Canonical impl
  bool equal(class_type const& rhs) const;
private: // Member Variables
  HWND  m_hwnd;
  HWND  m_hwndRoot;
};
```

This is incorporated into window_peer_sequence as shown in Listing 26.7. The iterator instance returned by begin() is passed the first window in the group of peers of m_hwnd, along with m_hwnd as a search root in the case where the endpoint iterator will be decremented; m_hwnd can be any window within the group of peers, since *GW_HWNDLAST* works identically for all peers. As is standard fare with the bidirectional and random access iteration implementations, the reverse iterator member types are defined in terms of std::reverse_iterator, and the `rbegin()` and `rend()` methods pass the reverse iterator specialization constructions the opposing iterator methods: In other words, `rbegin()` returns the adapted form of `end()`, and `rend()` returns the adapted form of `begin()`.

Listing 26.7 Adjusted Version of `window_peer_sequence`

```
class window_peer_sequence
{
public: // Member Types
  typedef zorder_iterator                       iterator;
  typedef std::reverse_iterator<iterator>     reverse_iterator;
  typedef window_peer_sequence                class_type;
public: // Construction
  explicit window_peer_sequence(HWND hwnd)
    : m_hwnd(hwnd)
  {}
public: // Iteration
  iterator           begin() const
  {
    return iterator(::GetWindow(m_hwnd, GW_HWNDFIRST), m_hwnd);
  }
  iterator           end() const;
  reverse_iterator   rbegin() const
  {
    return reverse_iterator(end());
  }
  reverse_iterator   rend()const
  {
    return reverse_iterator(begin());
  }
private: // Member Variables
  HWND  m_hwnd;
};
```

Note: If you're looking at the listing with suspicion, wondering whether all is right with `end()`, give yourself an extra 50 house points. I've deliberately not shown its implementation because it contains a gotcha that I'm saving for later, when we sit round the campfire, get out the harmonica, and sing the bidirectional iterator blues (Section 26.7).

26.5 Handling External Change

Aficionados of Windows should be panicking at the definition provided thus far because they know that the Z-order can change dynamically. Although this is not the case for child windows, for example, dialog controls, which tend to stay where they've been put, top-level windows can move around at any time, under the control of their individual programs or due to the user simply clicking on one or using Alt-Tab to bring other windows to the fore.

In all the collections we've examined thus far, either the elements are (directly or as copies) under the control of the collection (`glob_sequence`, Chapter 17; `pid_sequence` and `process_module_sequence`, Chapter 22; `Fibonacci_sequence`, Chapter 23; `CArray_cadaptor` and `CArray_iadaptor`, Chapter 24) or the API abstraction ensures that changes to the elements by other threads or processes do not cause the enumeration to fail or exhibit spurious behavior (`readdir_sequence`, Chapter 19; `findfile_sequence`, Chapter 20). The exception to this is `environment_map` (Chapter 25), which avoids inconsistencies by caching its enumerable contents in short-lived iterator snapshots, and which we decreed must not be used concurrently in more than one thread within a given process. In this case, however, there is the distinct possibility that the enumeration can be interrupted by the actions of other processes whose effects we can neither control nor ignore, and against which the enumeration API itself does not protect us. This is because the window whose handle an iterator instance may hold at a given point can be destroyed—it might be the frame window of the DVD player application you've decided to close in order to give *Extended STL* your full attention—in which case the call to `GetWindow()` will fail, returning *`NULL`*. In a very real sense, action outside the process is invalidating the iterator. This is not something that classic STL has ever sought to address; we require a different approach.

26.5.1 `stlsoft::external_iterator_invalidation`

When we pass a handle to `GetWindow()`, it is possible that it refers to a window that no longer exists. We must detect this event and act accordingly. If `GetWindow()` returns *`NULL`*, this might be because it was the last window in the Z-order, or it might be that the window whose handle value we passed to it is no longer valid. We can determine the reason by immediately calling the Windows API function `GetLastError()`, which returns *`ERROR_SUCCESS`* (*`0`*) in the former case, or another error code, usually *`ERROR_INVALID_WINDOW_HANDLE`*, in the latter. So we can detect an external invalidation event as follows:

```
zorder_iterator& zorder_iterator::operator ++()
{
  . . .
  m_hwnd = ::GetWindow(m_hwnd, GW_HWNDNEXT);
  if( NULL == m_hwnd &&
      ERROR_SUCCESS != ::GetLastError())
  {
    . . . // External invalidation was detected
  }
}
```

The question is what to do when this happens. Because the iterator holds on to only the current point in the iteration, which is now invalid and therefore useless, we cannot step back a place and then continue onward. What we could do, however, is restart the iteration by invoking `GetWindow(m_hwndRoot, GW_HWNDFIRST)`. It's entirely possible that, due to external events that do *not* invalidate the iterator's window handle but that nonetheless change the arrangement of the peers, elements may appear in the apparent sequence more than once. Indeed, client code must be aware of this—another leaky abstraction (Chapter 6)—and be codified accordingly. Hence, it's arguable that restarting the iteration is a viable option.

The reason I chose not to use this is not the extraordinarily remote (and practically ignorable) possibility that this might result in a loop that never completes. Rather, it's because it gains nothing and takes away the choice that the programmer using the sequence class might make about what response he or she might wish to make. That response might indeed be to restart the iteration, but it might also be to ask the user to stop closing windows while the program is doing its business. We need to let the programmer decide. Thus, the action taken is to throw an instance of the exception class `stlsoft::external_iterator_invalidation`, whose class interface is shown below. This class is one of a family of runtime exceptions that pertain to unwanted but legitimate effects on STL collection contents as a result of the actions of external entities, that is, other processes, or the operating system itself.

```
// In namespace stlsoft
class external_iterator_invalidation
  : public iteration_interruption
{
  . . .
public: // Construction
  external_iterator_invalidation();
  explicit external_iterator_invalidation(char const* message);
  external_iterator_invalidation(char const* message, long errorCode);
  . . .
};
```

So, if the code in the increment or decrement operators detects that the window handle is invalid, an instance of `external_iterator_invalidation` is thrown:

```
zorder_iterator& zorder_iterator::operator ++()
{
  . . .
  m_hwnd = ::GetWindow(m_hwnd, GW_HWNDNEXT);
  if( NULL == m_hwnd &&
      ERROR_SUCCESS != ::GetLastError())
  {
    throw stlsoft::external_iterator_invalidation("z-order search
failed: window has been destroyed", static_cast<long>(dwErr));
  }
}
```

The advantage of using this mechanism instead of the manual approach is that, being an exception, it cannot be forgotten. It would be easy to forget to remember to call `GetLastError()` and to test it against *`ERROR_INVALID_WINDOW_HANDLE`*, but it's not possible to forget the exception since it's thrown by the callee. The caller must take appropriate action. This is a clear case where exceptions are superior to error codes.

26.6 `winstl::child_window_sequence`

Now it's time to look at `child_window_sequence` (Listing 26.8). In every aspect but one, it is identical to `window_peer_sequence`.

Listing 26.8 Definition of `child_window_sequence`

```
// In namespace winstl
class child_window_sequence
{
public: // Member Types
  typedef zorder_iterator                     iterator;
  typedef std::reverse_iterator<iterator>     reverse_iterator;
  typedef child_window_sequence               class_type;
public: // Construction
  explicit child_window_sequence(HWND hwnd)
    : m_hwnd(::GetWindow(hwnd, GW_CHILD))
  {}
public: // Iteration
  iterator              begin() const;
  iterator              end() const;
  reverse_iterator      rbegin() const;
  reverse_iterator      rend() const;
public: // Size
  bool  empty() const;
private: // Member Variables
  HWND  m_hwnd;
};
```

Obviously there is a swag of different ways to refactor such duplication, which we'll look at later. But first we need to address a couple of bugs related to the bidirectional nature of `zorder_iterator`, one of which is pretty much a showstopper. The solution will require a significant refactoring of `zorder_iterator`'s constructors, so we'll deal with this first.

26.7 Bidirectional Iterator Blues

26.7.1 `end()` Sentinels Gotcha

To support *input* and *forward iterator* semantics (Section 1.3), there's no need to have the endpoint iterator do anything special or even to have state. (The practical exception to this is that the instance might maintain knowledge of its source collection to help detect bugs; we saw this in Section 20.5.) Thus the following yield logically equivalent instances:

```
some_collection             c;
some_collection::iterator   it;
some_collection::iterator   e = c.end();
assert(e == it);
```

However, this nature of endpoint iterators does *not* apply to bidirectional iterators.

Rule: For any collection type C exhibiting bidirectional iterators of type I, the relationship C().end() == C::I() can never hold.

In other words:

```
some_collection             c;
some_collection::iterator   it;
some_collection::iterator   e = c.end();
assert(e != it);
```

The reason for this is that it must be possible to decrement an endpoint instance of a bidirectional iterator, in order to get back into the enumerated range. In other words, the following code must be valid:

```
some_collection             c;
some_collection::iterator   b = c.begin();
some_collection::iterator   e = c.end();
if(b != e)
{
  *--e; // Well defined
}
```

This is a gotcha waiting for intrepid STL extension developers, particularly those who are making the transition from collections supporting input or forward iterators to those supporting bidirectional iterators. Remember that the original implementation of window_peer_sequence implemented end() as follows:

```
iterator window_peer_sequence::end() const
{
  return iterator();
}
```

Since zorder_iterator::equal() determines equality of two instances by the equality of their m_hwnd members, this meant that the code continued to work correctly in a forward direction, and all the existing unit tests passed. Alas, I neglected to expand the unit tests for reverse iteration, and thus a dormant bug was born. When I came, some time later, to use the component in the reverse direction, I discovered that it did not work. In fact, nothing happened. The implementation of zorder_iterator::operator --() did not have the precondition enforcement that it now does, whereby it asserts that m_hwndRoot is not *NULL*. Thus a decrement of the endpoint iterator simply yielded the endpoint iterator—*NULL* begets *NULL* with GetWindow()—and the bug acted silently!

As soon as I discovered this, I placed the precondition enforcement that now resides there, and that in turn led me to discover that the definition of end() in window_peer_sequence (and child_window_sequence) should instead have been:

```
iterator window_peer_sequence::end() const
{
  return iterator(NULL, m_hwnd);
}
```

Now the iterator instance will have a non-*NULL* `m_hwndRoot` with which to elicit the last peer, via *GW_HWNDLAST*.

26.7.2 Deadly Double Dereference

This second bug is exceedingly subtle, but it means, alas, that the current definition of `zorder_iterator` is fatally flawed. The conventional way to implement reverse iterators is to use the `std::reverse_iterator` iterator adaptor class template (C++-03: 24.1.1.1). Each reverse iterator instance has an equivalent instance of the underlying forward iterator, known as the base iterator. The important thing to know is that a reverse iterator instance and its base iterator instance refer to different points in the underlying collection. The standard (C++-03: 24.4.1;1) defines the relationship between a reverse iterator and its corresponding iterator as established by the identity: `&*(reverse_iterator(it)) == &*(it - 1)`. That doesn't mean a whole lot to the average person, so I'll illustrate it with some code. Consider the following vector specialization and four of its iterators:

```
std::vector<int>    v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
std::vector<int>::iterator          b = v.begin();
std::vector<int>::iterator          e = v.end();
std::vector<int>::reverse_iterator  rb = v.rbegin();
std::vector<int>::reverse_iterator  re = v.rend();
```

The four iterator instances have the following characteristics.

- `b` refers to `v[0]`, and `*b == 1`.
- `e` refers to `v[3]`, which doesn't exist, of course, so `e` cannot be dereferenced.
- `rb` holds an instance of the base type (`std::vector<int>::iterator`) equivalent to `e`, which refers to `v[3]` (which doesn't exist). But `rb` is said to *refer* to `e - 1`, which is `v[2]`, so dereferencing it is valid, and `*rb == 3`.
- `re` holds an instance of the base type (`std::vector<int>::iterator`) equivalent to `b`, which refers to `v[0]` (which *does* exist). But `re` is said to *refer* to `b - 1`, which is `v[-1]`, which doesn't exist, of course. Hence, `re` cannot be dereferenced.

The way `std::reverse_iterator` works is that it maintains an actual instance of its base type iterator and implements all its methods in terms of that. Listing 26.9 shows a snippet of an implementation of `std::reverse_iterator`. The implementation of `operator *()`

clearly illustrates the manner in which the off-by-one relationship between a reverse iterator and its base instance is implemented.

Listing 26.9 Definition of `std::reverse_iterator`

```
// In namespace std
template <typename I>
class reverse_iterator
  : public iterator<iterator_traits<I>::iterator_category
                  , iterator_traits<I>::value_type
                  , iterator_traits<I>::difference_type
                  , iterator_traits<I>::pointer
                  , iterator_traits<I>::reference
                  >
{
public: // Member Type
  typedef I                   iterator_type;
  typedef reverse_iterator<I> class_type;
public: // Construction
  reverse_iterator(iterator_type base)
    : m_base(base)
  {}
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    --m_base;
    return *this;
  }
  reference operator *()
  {
    iterator_type base = m_base;
    --base;
    return *base;
  }
  . . .
private: // Member Variables
  iterator_type m_base;
};
```

Now here's the problem with `zorder_iterator`. Each time an instance of the reverse iterator specialization (`std::reverse_iterator<zorder_iterator>`) is dereferenced, `GetWindow()` is called, via `zorder_iterator::operator --()`. Since the Z-order can change as a result of action in another process or thread between these two calls, any code that invokes the dereference operator multiple times is not well defined. For example, the algorithm `for_each_if()`, which we'll discuss in Volume 2, has a default (unspecialized) implementation, as shown in Listing 26.10.

Listing 26.10 Implementation of the `for_each_if()` Algorithm

```
template< typename I  // Iterator type
        , typename UF // Unary function to apply if suitable
        , typename UP // Unary predicate to test suitability
        >
UF for_each_if(I first, I last, UF func, UP pred)
{
  for(; first != last; ++first)
  {
    if(pred(*first))
    {
      func(*first);
    }
  }
  return func;
}
```

The `first` iterator is dereferenced twice within this algorithm. Using the current implementation of `zorder_iterator` with this algorithm, or any like it, could lead to the unpleasant position of having selected one window for some kind of action and then performing that action on a different window. Imagine the kerfuffle!

You would be forgiven at this point for just shrugging and saying: *Too hard!* But that's not the creed of the imperfect practitioner. Thankfully, the apparent hopelessness of the situation is predicated on the assumption that a reverse iterator has to be written in terms of `std::reverse_iterator`. But while it's overwhelmingly the case that this is a sound option—I've never encountered any other component that did not do so—it's not mandatory. It's important to realize that the standard provides `std::reverse_iterator` as a convenience; there's no grand dictate from on high that says it must be used.

In this case, we can relegitimize our iterator class by eschewing the use of `reverse_iterator` and providing a custom iterator class. As long as this fulfills the requirements of a reverse iterator—it provides the methods for all necessary iterator refinements, has the required relationship with its base iterator, and provides a `base()` method with which to elicit its equivalent base iterator instance—and it also does not exhibit anything like the double-dereference problem, all will be well. Thus, we require a different implementation for `zorder_iterator`, presented later in this chapter.

26.7.3 When a Bidirectional Iterator Is Not a Forward Iterator but a Reversible Cloneable Iterator

Smart STL cookies will have read the last item and scratched their heads in wonder. If two copies of a bidirectional iterator that are decremented may not be equal, the iterator violates the requirements of the bidirectional iterator concept (C++-03: 24.1.4;1) that include this very relationship. Such smart STL cookies would be right.

It gets worse. Since we know this relation does not hold, we also know that the corresponding requirement of forward iterators (C++ -03: 24.1.3;1) does not hold: `GetWindow()` no more promises repeatable results with *GW_HWNDNEXT* than it does with *GW_HWNDPREV*. It seems we're in a right pickle.

We have components that provide meaningful behavior corresponding to a large degree with official iterator concepts—forward and bidirectional—but that fail to do so in one crucial aspect. When an iterator fails to fulfill the forward iterator concept, the only recourse within the standard is to denote it as being an input iterator. But, by definition, input iterators are not bidirectional since the bidirectional iterator concept is a refinement of the forward iterator concept.

The failure to fulfill forward iterator semantics is partial. A forward iterator is distinguished from an input iterator in two major ways: Independent copies can be taken, and these copies obey the advancement equality relation whereby if `i1 == i2`, then `++i1 == ++i2`. Here we have a case where independent copies can be taken, but these copies do not obey the advancement equality relation. What we have in `zorder_iterator` is a *reversible cloneable iterator*.

One solution might be to eschew any reverse semantics and present an input iterator interface, but, in my opinion at least, that's throwing the very functional baby out with the bath water. As we will discover in detail in Volume 2, iterator categories are almost entirely for the purpose of specializing algorithms. To be sure, this is a *very* important purpose, but it can perhaps overreach. If we throw away the reverse capabilities, we achieve no material gain since, despite the lack of advancement equality relation, using reverse iteration with statements such as the following is entirely well defined:

```
std::for_each(wps.rbegin(), wps.rend()
          , predicate_function(::IsWindowEnabled, window_show(true)));
```

The answer is incredibly simple, but it's the furthest thing from obvious. It requires just one symbol change. We preserve all the current functionality of `zorder_iterator` but simply change it from declaring itself bidirectional to being input, as follows:

```
class zorder_iterator
  : public std::iterator< std::input_iterator_tag
                        , HWND, ptrdiff_t, void, HWND>
{
  . . .
```

Now, algorithms can't make any unfounded assumptions about the capabilities of the iterators, but we can still use the apparent bidirectional functionality directly if we wish. The standard defines a reversible container (C++-03: 23.1) as a container with the `reverse_iterator` and `const_reverse_iterator` member types, which are bidirectional or random access, and `rbegin()` and `rend()` methods. Although it seems odd to present an ostensibly reversible STL collection that exhibits iterators that are not bidirectional, there is actually no purporting of behavior inherent in the component definitions: The only concept checking done in such a case is on the iterator category, and in that case we are entirely safe in claiming the minimum. We have not lain a Goose (Section 10.1.3).

26.8 winstl::zorder_iterator: A Reversal of Self

Without further ado, I'll just pull out the white rabbit: The reverse iterator for zorder_iterator, is, drum roll . . . zorder_iterator! Well, obviously there's a little more to it. Nonetheless, it is reasonably straightforward. First, we require some constants and some traits. The constants are provided within a base class for the iterator, zorder_iterator_base, as shown in Listing 26.11.

Listing 26.11 Definition of the zorder_iterator_base **Class**

```
// In namespace winstl
struct zorder_iterator_base
{
public:
  enum search
  {
      fromFirstPeer   = 1 // Move to start of list of peers
    , fromCurrent     = 2 // Move to current point in list of peers
    , atLastPeer      = 3 // Move to the end of list of peers
    , fromFirstChild  = 4 // Move to the start of list of children
    , atLastChild     = 5 // Move to the end of list of children
  };
};
```

26.8.1 zorder_iterator Traits

The traits class is not, in this case, a class template and its specialization(s), but rather two separate classes, zorder_iterator_forward_traits and zorder_iterator_reverse_traits. Listing 26.12 shows the required forward declarations for both and the definition of zorder_iterator_forward_traits.

Listing 26.12 Definition of the zorder_iterator_forward_traits **Class**

```
// In namespace winstl
struct zorder_iterator_forward_traits;
struct zorder_iterator_reverse_traits;

struct zorder_iterator_forward_traits
{
public: // Member Types
  typedef zorder_iterator_forward_traits  this_type;
  typedef zorder_iterator_reverse_traits  alternate_type;
public: // Functions
  static HWND get_first_child(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_CHILD);
  }
  static HWND get_first_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDFIRST);
  }
  static HWND get_next_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDNEXT);
  }
  static HWND get_previous_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDPREV);
  }
  static HWND get_last_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDLAST);
  }
};
```

The five methods correspond to the five constants in `zorder_iterator_base`, and their implementations are straightforward. Note the member type `alternate_type`, which is `zorder_iterator_reverse_traits`. The definition of this class, shown in Listing 26.13, is a functional reverse of `zorder_iterator_forward_traits`. Each function does the opposite by simply using the opposing *GW_HWND** flag, except `get_first_child()`, which uses *GW_CHILD* and then *GW_HWNDLAST* in serial to effect the opposite of *GW_CHILD*.

Listing 26.13 Definition of the `zorder_iterator_reverse_traits` Class

```
// In namespace winstl
struct zorder_iterator_reverse_traits
{
public: // Member Types
  typedef zorder_iterator_reverse_traits  this_type;
  typedef zorder_iterator_forward_traits  alternate_type;
public: // Functions
  static HWND get_first_child(HWND hwnd)
  {
    return ::GetWindow(::GetWindow(hwnd, GW_CHILD), GW_HWNDLAST);
  }
  static HWND get_first_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDLAST);
  }
  static HWND get_next_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDPREV);
  }
  static HWND get_previous_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDNEXT);
  }
  static HWND get_last_peer(HWND hwnd)
  {
    return ::GetWindow(hwnd, GW_HWNDFIRST);
  }
};
```

26.8.2 zorder_iterator_tmpl<>

The traits classes are used by a templatized form of `zorder_iterator`, with the rather horrible name `zorder_iterator_tmpl`, shown in Listing 26.14.

Listing 26.14 Definition of the `zorder_iterator_tmpl` Class Template

```
// In namespace winstl
template <typename T = zorder_iterator_forward_traits>
class zorder_iterator_tmpl
  : public zorder_iterator_base
  , public std::iterator< std::input_iterator_tag
                        , HWND, ptrdiff_t
                        , void, HWND // BVT
                        >
{
public: // Member Types
  typedef T                                              traits_type;
  typedef zorder_iterator_tmpl<T>                        class_type;
  typedef zorder_iterator_tmpl<typename T::alternate_type>
                                                         base_iterator_type;
  typedef base_iterator_type                             iterator_type;
private: // Construction
  zorder_iterator_tmpl(HWND hwndRoot, HWND hwndCurrent);
public:
  static class_type create(HWND hwndRoot, search from);
  zorder_iterator_tmpl();
public: // Iteration
  class_type&         operator ++();
  class_type          operator ++(int);
  value_type          operator *() const;
  bool                equal(class_type const& rhs) const;
  class_type&         operator --();
  class_type          operator --(int);
  base_iterator_type  base() const;
private: // Implementation
  static HWND get_next_window_(HWND hwnd, HWND (*pfn)(HWND ));
private: // Member Variables
  HWND    m_hwndRoot;
  HWND    m_hwndCurrent;
};

typedef zorder_iterator_tmpl<zorder_iterator_forward_traits>
                                                            zorder_iterator;
```

There are several things to note straightaway. First, `zorder_iterator_tmpl` inherits from `zorder_iterator_base` in order to have direct access to the constants. This means that the search flags may be expressed in terms of the iterator. This is the technique used in the **IOStreams**, where the flags are defined in `std::ios_base` but are directly accessible to, and in terms of, all derived classes.

Second, it specializes `std::iterator` to exhibit the input iterator category and the by-value temporary element reference category, catering to the requirements discussed earlier.

The conversion constructor is declared `private` to avoid possible initialization by an invalid window pair by a user. To use this constructor, the static `create()` method must be used, which takes a window, and one of the `search` enumerators, as in the following:

```
child_window_sequence::iterator child_window_sequence::begin()
{
  return iterator::create(m_hwnd, iterator::fromFirstChild);
}
```

It's the job of `create()` (shown in Listing 26.15) to determine the appropriate values for `m_hwndRoot` and `m_hwndCurrent`, according to the given search type. `hwndRoot` is adjusted in the case where a child window is requested, by invoking the `get_first_child()` method on the traits type. `hwndCurrent` is set to `hwndRoot` for searching from the current position, or to the endpoint marker *`NULL`* if searching from the last peer or child. If searching from the first peer or child, `hwndCurrent` is set to the result of a call to the `get_first_peer()` method of the traits type.

Listing 26.15 Implementation of `zorder_iterator_tmpl::create()`

```
template <typename T>
zorder_iterator_tmpl<T>
 zorder_iterator_tmpl<T>::create(HWND hwndRoot, search from)
{
  HWND  hwndCurrent;
  switch(from)
  {
    case    fromFirstChild:
    case    atLastChild:
      hwndRoot = get_next_window_(hwndRoot
                                , traits_type::get_first_child);
    default:
      break;
  }
  switch(from)
  {
    case    fromCurrent:
        hwndCurrent = hwndRoot;
        break;
    case    fromFirstPeer:
    case    fromFirstChild:
      hwndCurrent = get_next_window_(hwndRoot
                                    , traits_type::get_first_peer);
      break;
    case    atLastChild:
    case    atLastPeer:
      hwndCurrent = NULL;
      break;
  }
  return zorder_iterator_tmpl<T>(hwndRoot, hwndCurrent);
}
```

All traits type methods are invoked via the private static worker method `get_next_window_()` (Listing 26.16), which tests for external iterator invalidation.

Listing 26.16 Implementation of `zorder_iterator_tmpl::get_next_window_()`

```
template <typename T>
HWND zorder_iterator_tmpl<T>::get_next_window_(HWND hwnd
                                              , HWND (*pfn)(HWND))
{
  hwnd = (*pfn)(hwnd);
  if(NULL == hwnd)
  {
    DWORD dwErr = ::GetLastError();
    if(ERROR_SUCCESS != dwErr)
    {
      throw external_iterator_invalidation("z-order search failed:
window has been destroyed", static_cast<long>(dwErr));
    }
  }
  return hwnd;
}
```

The default constructor sets both member variables to *NULL*. The copy constructor, copy assignment operator, and destructor are all left undeclared, as the compiler-generated versions are perfectly adequate.

The iterator class template has all the bidirectional iterator methods we saw in the previous non-template version. The increment and decrement operators have the same logic as previously described, but this time they use the `get_next_peer()`, `get_previous_peer()`, and `get_last_peer()` methods from the traits type, invoked via `get_next_window_()`. Listing 26.17 shows the definitions of the preincrement and predecrement operators. The post variants are, as always, done in the canonical fashion (see Listing 19.8).

Listing 26.17 Implementation of the Preincrement and Predecrement Operators

```
template <typename T>
typename zorder_iterator_tmpl<T>::class_type&
 zorder_iterator_tmpl<T>::operator ++()
{
  WINSTL_MESSAGE_ASSERT("Attempt to increment an invalid / out-of-range
iterator", NULL != m_hwndCurrent);
  m_hwndCurrent = get_next_window_(m_hwndCurrent
                                  , traits_type::get_next_peer);
  return *this;
}
template <typename T>
typename zorder_iterator_tmpl<T>::class_type&
 zorder_iterator_tmpl<T>::operator --()
{
  WINSTL_MESSAGE_ASSERT("Attempt to decrement an invalid / out-of-range
iterator", NULL != m_hwndRoot);
  if(NULL != m_hwndCurrent)
  {
    m_hwndCurrent = get_next_window_(m_hwndCurrent
                                    , traits_type::get_previous_peer);
  }
  else
  {
    m_hwndCurrent = get_next_window_(m_hwndRoot
                                    , traits_type::get_last_peer);
  }
  return *this;
}
```

The dereference operator always returns `m_hwndCurrent`, which is either the current window or *NULL*.

26.8.3 Reversed Semantics

In order to support standard reverse iterator requirements, `zorder_iterator_tmpl` provides a `base()` method, which returns an instance of type `base_iterator_type`. This is where the `alternate_type` member type of the traits classes comes in. It is used to specialize `zorder_iterator_tmpl`, in order to derive the reverse iterator type `base_iterator_type`. (I follow the convention of `std::reverse_iterator` and include a typedef `iterator_type`, though I don't rate that as a good name choice: It's far too general.) `base()` is implemented as follows:

```
template <typename T>
typename zorder_iterator_tmpl<T>::base_iterator_type
 zorder_iterator_tmpl<T>::base() const
{
  base_iterator_type bi = base_iterator_type::create(m_hwndCurrent
                                                    , fromCurrent);
  return ++bi;
}
```

This is how we achieve a reversible iterator without the double-dereference problem. The `create()` method is used to give an instance of the `base_iterator_type` that has the same internal position as the instance on which `base()` is called. This is then incremented, which moves it *back*—it is the reversed type, remember—and returned. This type then has all the same behavior as the forward version, except its direction. When it is incremented, the call goes out to `zorder_iterator_reverse_traits::get_next_peer()`, which invokes `GetWindow()` with *GW_HWNDPREV*. When this returns *NULL*, the iterator is at an end, at the *start* of the window list. Similarly, when it is decremented, the calls go out to `zorder_iterator_reverse_traits::get_previous_peer()` and `zorder_iterator_reverse_traits::get_last_peer()`, which invoke `GetWindow()` with *GW_HWNDNEXT* and *GW_HWNDFIRST*, respectively.

When it is dereferenced, there is but the simple return of `m_hwndCurrent`: no double dereference, and therefore no risk of presenting different values in algorithms that use the iterator's value more than once. And finally, because it exhibits the `input_iterator_tag`, there is no risk of it being used with any algorithms that assume reliable multipass, rather than cloneable, behavior. But we can still call any algorithm with either `begin()` and `end()`, or `rbegin()` and `rend()`, since both forward and reverse iterators exhibit the input iterator category. *QED*.

Tip: If you can't support a bidirectional input iterator for a collection that facilitates reverse enumeration, you may instead supply independent input iterators, where one goes forward over the collection and the other one goes backward.

26.9 Finalizing the Window Peer Sequences

Given the refinements for `zorder_iterator`, the sequence classes `window_peer_sequence` and `child_window_sequence` become both simpler and nearly identical. The only differences now are in the `begin()`, `end()`, `rbegin()`, and `rend()` methods. Listing 26.18 shows `window_peer_sequence`.

Listing 26.18 Final Version of `window_peer_sequence`

```
class window_peer_sequence
{
public: // Member Types
  typedef zorder_iterator                     iterator;
  typedef iterator::value_type                value_type;
  typedef iterator::base_iterator_type        reverse_iterator;
  typedef window_peer_sequence                class_type;
public: // Construction
  explicit window_peer_sequence(HWND hwnd);
public: // Iteration
  iterator                begin() const
  {
    return iterator::create(m_hwnd, iterator::fromFirstPeer);
  }
  iterator                end() const
  {
    return iterator::create(m_hwnd, iterator::atLastPeer);
  }
  reverse_iterator        rbegin() const;
  {
    return reverse_iterator::create(m_hwnd
                                  , reverse_iterator::fromFirstPeer);
  }
  reverse_iterator        rend() const
  {
    return reverse_iterator::create(m_hwnd
                                  , reverse_iterator::atLastPeer);
  }
public: // State
  bool   empty() const;
private: // Member Variables
  HWND    m_hwnd;
private: // Not to be implemented
  window_peer_sequence(class_type const&);
  class_type& operator =(class_type const&);
};
```

Note that the implementations of `begin()` and `rbegin()`, and `end()` and `rend()`, are identical except for the return type; this is in stark contrast to the canonical form (see Section 24.11.2).

The implementation of `child_window_sequence` is *exactly* the same, except that where `window_peer_sequence` uses the constants `fromFirstPeer` and `atLastPeer`, `child_window_sequence` uses `fromFirstChild` and `atLastChild`. Naturally, this situation is anathema for all refactoring-prone engineers, which is to say all good engineers. Thus, we must refactor. Included in the extra material on the CD is a section explaining how this is done, via a common sequence template parameterized by search constants.

26.10 Summary

This chapter started out as an exercise in illustrating the correct implementation of bidirectional iterator semantics and ended up being much more than that. It's fair to say we've fairly covered the issue of bidirectional iterators, including their particular requirements of statefulness and equivalence, and you should be ready to write your own after absorbing this material. Along the way, we've looked at the issue of external iterator invalidation—the changing of an iterating series' contents by action outside the program's control—and why the best way to deal with it is to throw an exception. (We'll see a lot more of this topic in Chapter 33.) Finally, let's not forget the discombobulating notion of an iterator class template being its own reverse iterator type.

26.11 Z-Plane: Coda

If you'll permit me an indulgent digression, I must say that as soon as I realized that the previous form using `std::reverse_iterator` was logically flawed, I imagined this solution, or something close to it. Though I knew I needed to convert different function calls—that is, one class called `GetWindow(GW_CHILD)` in its constructor; the other did not—into constants, I did not *design* the final result. It was a refactoring evolution. In my opinion, most good systems are well designed, and most good libraries are well evolved. In either case, you must start from good founding principles, but the former is largely top-down, and the latter largely bottom-up, with a few sweeps of the design/refactoring scythe thrown in every now and then as necessary. This has been a good example of such a scything.

CHAPTER 27

String Tokenization

How vain it is to sit down and write when you have not stood up to live.

—Henry David Thoreau

Racing doesn't get any easier; you just get faster.

—Greg Lemond

27.1 Introduction

By now you're used to my writing style and may be expecting a sequence of expository steps describing the evolution of a collection component into a beautifully crafted class that handles all possible cases with aplomb. If so, you're in for somewhat of a disappointment in this chapter. What you'll actually get here is a story of how the effects of complexity, efficiency, flexibility, and several hundred refactoring steps conspire to yield a class that's really flexible and very fast but that grossly violates *Henney's Hypothesis* (Chapter 14).

As I wrote the chapters in Part II, most components underwent a nontrivial degree of refactoring due to the highly detailed reexamination they've received in the documentation of their designs and implementations for this book. However, that's not the case with the subject of this chapter, **STLSoft**'s `string_tokeniser` class template. Its implementation has been honed to a fine point over the years, such that nothing was improved during the research and writing of this chapter. Conversely, its interface leaves something to be desired in certain use cases. Its class interface is succinct and extremely simple, consisting of three straightforward constructors, `begin()`, `end()`, and `empty()`. Contrarily, its template interface is obscure, overblown, and almost the perfect counter-motivating example for *Henney's Hypothesis*. Though this class has only two non-defaulted template parameters, *Henney's Hypothesis* applies because some of the most useful modes are when one or more of the four (!) defaulted template parameters must be explicitly specified.

So this chapter has two main purposes. First, `string_tokeniser` is an excellent motivating example for the *by-value temporary* reference categorization (Section 3.3.5), in that it highlights tradeoffs between element reference categories, iterator categories, and performance considerations. Second, the very imperfection of the template interface is a great subject for study in illustrating the difficulties of writing flexible and discoverable general-purpose template components.

27.2 strtok()

String tokenization is the process of breaking a character string into smaller pieces based on a delimiter. The notion of a delimiter is a plastic one: It may consist of a single character, a composite string, or a set of characters, or it may be even more complex, involving positional and/or context-relative characteristics. Standard C supports string tokenization via the `strtok()` function (C-99: 7.21.5.8), which allows for the tokenization of a C-style string based on one of a set of single-character delimiters.

```
char* strtok(char* str, char const* delimiterList);
```

This is the only tokenization functionality provided by either standard C or standard C++. (The C standard also defines a wide-char equivalent, called `wcstok()`, whose arguments and return type are defined in terms of `wchar_t`. It has behavior otherwise identical to `strtok()`.) The function tokenizes the contents of `str`, based on the delimiter(s) specified in `delimiterList`, returning the start of each token. When no tokens remain, *`NULL`* is returned. Tokenization commences by passing the string to be tokenized as the `str` parameter. Subsequent tokens are retrieved by passing *`NULL`* as the `str` parameter. The current tokenization point is stored in an internal static variable and is reset when a non-*`NULL`* value is passed as `str`. For example, we can tokenize a whitespace-delimited string as follows:

```
char*       str = . . .
char*       tok;
const char  delims[]  = " \r\n\t";

for(tok = ::strtok(str, delims); NULL != tok;
     tok = ::strtok(NULL, delims))
{
  ::puts(tok);
}
```

While this is simple, efficient, and reasonably transparent, it has a number of drawbacks. First, and most important, it uses a shared internal variable for the tokenization state, making it not reentrant and not thread-safe. Two threads using `strtok()` at the same time, even on different strings, are subject to a race condition on the internal variable that remembers the previous position. Although the standard does not address threading concerns, several standard library implementations employ thread-specific storage techniques to make calls to `strtok()` in different threads independent. A worse problem, and one not amenable to any such technical amelioration, is that intrathread nested tokenization loops will have undefined behavior. Consider the following code:

```
char  str[] = "abc,def;ghi,jkl;;";
char* outer;
char* inner;
for(outer = strtok(str, ";"); NULL != outer; outer = strtok(NULL, ";"))
```

```
{
  std::cout << "Outer token:   " << outer << std::endl;
  for(inner = strtok(outer, ","); NULL != inner;
      inner = strtok(NULL, ","))
  {
    std::cout << "  Inner token: " << outer << std::endl;
  }
}
```

It does not print out, as its author presumably intends, the following:

```
Outer token:   abc,def
  Inner token: abc
  Inner token: def
Outer token:   ghi,jkl
  Inner token: ghi
  Inner token: jkl
```

Instead (with several compilers on my machine), it produces the following:

```
Outer token:   abc,def
  Inner token: abc
  Inner token: abc
```

This situation is similar to the use of `getenv()` that we saw in Chapter 25. It's all very well to use such standard functions that have side effects in a tight context, where you have full control over all code paths encountered and are confident of knowing that you've avoided nested `strtok()` calls. But consider the case where the contents of the outer loop are instead a call to another application function. It's not hard to imagine that two members of a development team working on different parts of a nontrivial project might unknowingly reproduce this inner/outer nesting.

The second problem is that in order to return C-style strings to the caller, the function must write the nul character (`'\0'`) into the source string (`str`) before returning. Consequently, the tokenized string must be writable and client code written with the expectation that the contents will be (partially) destroyed. We saw an example of this in Section 20.1.1, where the character buffer `tokenBuff` had to be allocated each recursion.

Third, the tokenized string must be in the form of a (mutable) C-style string, that is, a nul-terminated contiguous array of characters. Thus, using `strtok()` with string class instances is either ungainly or downright invalid. For example, it is not possible to combine `std::string` with `strtok()` (or `std::wstring` with `wcstok()`) because the standard does not guarantee that `std::basic_string` stores its contents contiguously, and any code that assumes it does has undefined behavior. Consequently, it leads to inefficiency because you must make a mutable copy of your source string. When all you're doing is testing whether a string contains a particular token, this is a galling waste of processing time.

Fourth, the function always skips blanks. In other words, if the string to be tokenized is `";;abc;;;def;;"` and the delimiter list is `";"`, the client code will receive two strings: `"abc"` and `"def"`. Though this is usually desired behavior, sometimes the blanks are meaningful and required, making `strtok()` unsuitable in such cases.

Fifth, it has, in my opinion, an ugly and unique interface whose semantics are not shared idiomatically with any other standard or widely used functions. I find this an unpleasant function to use, and each time I must do so I have to look up its semantics, which is not the case with most other standard C functions. Whatever you may think of it, you must agree that it *ain't* STL!

Finally, it supports only tokenization based on a single character. If you need to tokenize based on composite delimiters—for example, `"%%"`—or on delimiter pairs—for example, `"<"` (token start) and `">"` (token end)—you're out of luck. For all these reasons, `strtok()` is not a very popular function, and the search for better alternatives leads us on the path to the `string_tokeniser` class template.

27.3 SynesisSTL::StringTokeniser

A string tokenizer is such an obvious and necessary component, you might wonder why it was omitted from the C++-98 standard. I certainly did, and so I wrote one. The string tokenizer described in this chapter started life in the mid 1990s as a component of the Synesis Software company, under the name `StringTokeniser`, defined within the `SynesisSTL` namespace. The ramifications of its original purpose manifest themselves to this day—for good and ill—in the design of `stlsoft::string_tokeniser`.

Absent a few embarrassing gaffs with respect to STL correction (that I don't plan to repeat here to avoid inviting just opprobrium upon my nascent STL extension skills), its first definition was pretty much as shown in Listing 27.1.

Listing 27.1 Definition of Synesis's `StringTokeniser` Class Template

```
// In namespace SynesisSTL
template <typename C> // Character encoding
class StringTokeniser
{
public: // Member Types
  typedef C                                char_type;
  typedef std::basic_string<char_type>     string_type;
  typedef string_type                      value_type;
  class                                    const_iterator;
  typedef const value_type                 const_reference;
public: // Construction
  StringTokeniser(char_type const* str, char_type delim);
  StringTokeniser(string_type const& str, char_type delim);
  StringTokeniser(char_type const* str, size_t n, char_type delim);
  StringTokeniser(string_type const& str, size_t n, char_type delim);
  template <typename II>
  StringTokeniser(II from, II to, char_type delim);
public: // Iteration
  class const_iterator
    . . .
  {
    . . .
  public: // Forward Iteration
    value_type operator *() const;
    const_iterator& operator ++();
    const_iterator operator ++(int);
    . . . // And comparison operations
  };
  const_iterator begin() const;
  const_iterator end() const;
private: // Member Variables
  const string_type m_str;
  const char_type   m_delim;
};
```

I originally wrote the tokenizer for some file-processing utilities that perform automatic source modifications to source code bases en bloc. Its first application was in tokenizing file search patterns, for example, *"*.hpp;;;*.h;*.cpp;*.c;"*, *"*rb:*py"*. For this, the design shown in Listing 27.1 is pretty optimal. As it stands, this `StringTokeniser` has a nice set of features.

- The tokenizing logic is separately maintained within each iterator instance, so tokenization loops can be nested.
- It does not alter the tokenized string.
- It works with different character encodings.
- It is an *STL collection* (Section 2.2).
- It is highly discoverable, with nothing like `strtok()`'s "subsequent `NULL`" semantics.

Shortly after, another requirement of the utilities presented itself: to be able to retain the blanks in the search. (Imagine processing a C/C++ source file and having every blank line stripped!) Very soon after that, I decided I wanted to be able to use memory-mapped file techniques on the source files. As I'm sure you're aware, text files on Windows use the character sequence *"\r\n"* to delimit lines, as opposed to the single *'\n'* character used by UNIX. The standard C buffered streams library—`fopen()`, `fread()`, and so on—abstracts away these differences in text mode, and C/C++ code that uses them to examine file contents sees only a *'\n'* character-line delimiter. The same applies to the C++ **IOStreams** library. But when using memory-mapped files, you see an exact view of the file contents in memory. So, in order to take advantage of the considerable performance improvements afforded by the use of memory-mapped files, it became necessary to be able to use a composite string delimiter (*"\r\n"*) as a tokenizer.

In light of these two new requirements, along with a little hindsight on other matters that will receive attention throughout the chapter, it's possible to point out a number of drawbacks of the design so far.

- It works only for a single-character delimiter. If you want to use a character delimiter set (à la `strtok()`), composite string delimiters, or delimiter pairs, you're out of luck.
- It has a fixed string type, `std::basic_string<C>`. This raises a number of issues.
  - It is inflexible. You might need to use another string class, for example, `ACE_CString`.
  - A copy of the string to be tokenized is always taken in `m_str`. This is a wise default, but in circumstances where you're going to be tokenizing a string that persists for the lifetime of the tokenizer, the costs of memory allocation and copying are unnecessary.
  - Similarly, taking a copy of each enumerated token is potentially inefficient. We'll see later in the chapter how it's possible to tokenize a string without a single memory allocation.
- It always skips blanks.
- It tokenizes only C-style strings (albeit without requiring nul-termination), instances of `std::basic_string<C>`, and iterator ranges that are acceptable to the constructor of `std::basic_string<C>`. Although this covers a lot of cases, users who like to mix STL with other technologies will be forced into unnecessary conversions.

27.4 Tokenization Use Cases

Before we proceed, I want to discuss the permutations of string tokenization features that we might reasonably expect an efficient general-purpose component to provide. We've already touched on the following aspects:

- Mutating the tokenized string, versus taking a copy and mutating that, versus not mutating at all
- Having a one-character delimiter, versus a composite string delimiter, versus a character delimiter set
- Supporting different types for the tokenized string
- Supporting different types for resultant tokens
- Skipping blank tokens versus preserving them

The version of **STLSoft**'s `string_tokeniser` (Section 27.6) that I will show you covers all these cases. It's not a perfect job, however: Delimiting via character delimiter set is a bit of a mess. But there are more sophisticated types of tokenization you might wish to perform:

- Having delimiter pairs, for example, `'<'` and `'>'`, or `"<"` and `"/>"`, and so on
- Deducing columnar position, for example, having characters 0–3 in the first token, 5–9 in the second, and so on
- Matching context-dependent character sequences, that is, interpreting a subsequence as a delimiter based on prior elements in the sequence

Since I favor simplicity and efficiency in components as low level as this, I would instinctively steer clear of anything approaching regular expression capabilities. I've used the tokenizer component described here in its various guises for a decade or so, and I can confidently say that the vast majority of cases are covered by tokenization by single character, by composite string delimiter, or by character delimiter set, with or without stripping blanks. Thus, the extant version of `stlsoft::string_tokeniser` provides all the features in the first list above but does not provide the more advanced features in the second.

27.5 Other Tokenization Alternatives

Before we look at the **STLSoft** string tokenizer, let's see what other alternatives are out there.

27.5.1 `strtok_r()`

Still in the world of C we find `strtok_r()`, the re-entrant form of `strtok()`. This reasonably common function, though not C standard, is provided on many UNIX variants; some Windows compilers also support it.

```
char* strtok_r(char* str, char const* delimiterList, char** state);
```

Instead of the internal tokenization state variable, `strtok_r()` pushes state management out to the client code, which must pass the address of a `char*` state variable to each invocation. All other semantics are the same. Tokenization loops can nest and be well defined, but all other limitations remain. It's safe, but still not very useful.

27.5.2 IOStreams

Although it's a bit of a stretch, the **IOStreams** library can be said to support tokenization, as shown in the following example.

```
std::stringstream   sstm(";;abc;def;ghi;;;;;jkl;;;;;;");
std::string         tok;
while(std::getline(sstm, tok, ';'))
{
  std::cout << tok << std::endl;
}
```

As you'd expect, this thing doesn't exactly rocket along. It also preserves blanks, which is usually not desired. I've included it in the performance tests (Section 27.9) mainly to dissuade you from ever resorting to such a thing. On the plus side, this library, being standard, is always there.

27.5.3 `stlsoft::find_next_token()`

In Section 20.5.3, I introduced the `find_next_token()` restartable token parser functions. These are great if you're tokenizing based on single-character delimiters, but they do leave something to be desired on the usability scale. They're best left as implementation tools, as was shown for tokenizing multipart search patterns in `winstl::basic_findfile_sequence` (Section 20.5). However, I include them in the performance tests later in this chapter, as they probably represent an upper limit of how fast we might hope to perform tokenization.

27.5.4 boost::tokenizer

The **Boost** libraries contain a tokenizing component, called `tokenizer`, that has been available since about 2001. It takes an implementation approach quite different than the **STLSoft** component does, and it exists at a somewhat higher level of abstraction, insofar as it poaches the territory of regular expressions and custom parsers more than the low-level tokenization facilities provided by `stlsoft::string_tokeniser`. In addition to providing a similarly safe and nonmutating equivalent to `strtok()`, it also offers other capabilities, including, at the time of writing, escape character processing and the columnar fixed-width field splitting mentioned above. The downside of this sophistication is that it pays for the complexity in speed, even for the comparatively straightforward character delimiter set processing, as we'll see later in the chapter.

27.6 stlsoft::string_tokeniser

I'm now going to present what has been, for a long time, the stable implementation of the tokenizer component in the **STLSoft** libraries. Listing 27.2 shows its definition.

Listing 27.2 Definition of the `stlsoft::string_tokeniser` Class Template

```
template< typename S
        , typename D
        , typename B = skip_blank_tokens<true>
        , typename V = S
        , typename T = string_tokeniser_type_traits<S, V>
        , typename P = string_tokeniser_comparator<D, S, T>
        >
class string_tokeniser
{
public: // Member Types
  typedef string_tokeniser<S, D, B, V, T, P>  class_type;
  typedef string_tokeniser<S, D, B, V, T, P>  tokeniser_type;
  typedef S                                   string_type;
  typedef D                                   delimiter_type;
  typedef B                                   blanks_policy_type;
  typedef V                                   value_type;
  typedef T                                   traits_type;
  typedef P                                   comparator_type;
  typedef typename traits_type::value_type    char_type;
  typedef typename traits_type::size_type     size_type;
  typedef const value_type                    const_reference;
  class                                       const_iterator;
private:
  typedef typename traits_type::const_iterator_type
                                              underlying_iterator_type;
public: // Construction
  template <typename S1>
  string_tokeniser(S1 const& str, delimiter_type const& delim)
    : m_str(c_str_data(str), c_str_len(str))
    , m_delimiter(delim)
  {}
  template <typename S1>
  string_tokeniser(S1 const& str, size_type n
                  , delimiter_type const& delim)
    : m_str(c_str_data(str), n)
    , m_delimiter(delim)
  {}
  template <typename I>
  string_tokeniser(I from, I to, delimiter_type const& delim)
    : m_str(from, to)
    , m_delimiter(delim)
  {}
public: // Iteration
  const_iterator begin() const
  {
    STLSOFT_ASSERT(is_valid());
    return const_iterator(traits_type::begin(m_str)
                        , traits_type::end(m_str), m_delimiter);
  }
  const_iterator end() const
  {
    STLSOFT_ASSERT(is_valid());
    return const_iterator(traits_type::end(m_str)
                        , traits_type::end(m_str), m_delimiter);
  }
public: // Attributes
  bool   empty() const
  {
    STLSOFT_ASSERT(is_valid());
    return begin() == end();
  }
private: // Invariant
  bool is_valid() const;
private: // Member Variables
  const string_type     m_str;
  const delimiter_type  m_delimiter;
private: // Not to be implemented
  class_type& operator =(class_type const&);
};
```

There's little to be remarked upon with the class interface. As you would expect, the constructor templates use string access shims (Section 9.3.1) to provide compatibility with a large variety of string types. Things are less simple with the template interface. The first obvious issue is the number of template parameters. Six is a lot, and, as I mentioned earlier, this causes problems in the cases where two or more of the defaulted parameters are to be explicitly stipulated.

The first template parameter, `S`, is the tokenized string type. This is the type of the `m_str` member that holds a copy of the tokenized string. The second parameter, `D`, is the type of the delimiter. This is typically the same type as `S` if the delimiter is a composite string or a character delimiter set, or the character type apropos `S` for tokenization by a single-character delimiter (e.g., [`S` is `std::wstring`] => [`D` is `wchar_t`]). The third parameter, `B`, is a policy type used to determine whether blanks will be stripped or preserved. It defaults to `skip_blank_tokens<true>` (Listing 27.3), meaning that blanks will be skipped. You specify `skip_blank_tokens <false>` for blank tokens to be preserved in the range [`begin()`, `end()`). (The reason this template parameter is a type, rather than a Boolean value, is historical. I expected more sophistication than there turned out to be.)

Listing 27.3 Definition of `skip_blank_tokens` Policy Class Template

```
template <bool B>
struct skip_blank_tokens
{
  enum { value = B };
};
```

The fourth parameter, `V`, is the value type of the tokenizer (and its nested iterator class). It defaults to `S` but can be any other suitable string type (this suitability is determined by the traits type, discussed shortly). In this way, the different permutations for copying/noncopying of tokenized strings and/or values are supported. For example, if `S` is a string view (e.g., `stlsoft::basic_string_view<char>`) and `V` is a bona fide string class (e.g., `std::string`), the tokenized string will *not* be copied (or mutated), but the values returned by dereferencing the iterators will be stand-alone string instances. (String views are types that hold a length and a pointer into an array of characters but that do not assume nul-termination of the slice of the array to which they refer. String views will be discussed in detail in Volume 2.)

Explanation of the final two template parameters, `T` and `P`, can wait awhile. First, we're going to look at how the iterator is implemented and see how it achieves the blistering performance I keep going on about (but have yet to prove; Section 27.9). Then we'll look at a number of example specializations covering common tokenization scenarios, see how well the tokenizer affects the common tokenization scenarios, and, finally, consider some mitigating measures in the cases in which it proves to be decidedly imperfect.

27.6.1 `stlsoft::string_tokeniser::const_iterator`

The full definition of the nested `const_iterator` class is shown in Listing 27.4.

Listing 27.4 Definition of `const_iterator`

```
public: // Iteration
  class const_iterator
    : public std::iterator< std::forward_iterator_tag
                          , value_type, ptrdiff_t
                          , void, value_type  // BVT
                          >
  {
  public: // Member Types
    typedef const_iterator    class_type;
    . . .
  private: // Construction
    friend class  string_tokeniser<S, D, B, V, T, P>;
    const_iterator( underlying_iterator_type first
                  , underlying_iterator_type last
                  , delimiter_type const&    delimiter)
      : m_find0(first)
      , m_find1(first)
      , m_next(first)
      , m_end(last)
      , m_delimiter(&delimiter)
      , m_cchDelimiter(comparator_type::length(delimiter))
    {
      if(m_end != m_find0)
      {
        increment_();
      }
    }
  public:
    const_iterator();
    const_iterator(class_type const& rhs);
    class_type const& operator =(class_type const& rhs);
  public: // Forward Iterator Methods
    value_type operator *() const
    {
      return traits_type::create(m_find0, m_find1);
    }
    class_type& operator ++()
    {
      increment_();
      return *this;
    }
    const class_type operator ++(int);
    bool equal(class_type const& rhs) const
    {
      STLSOFT_MESSAGE_ASSERT( "Comparing iterators from different
tokenisers", m_end == rhs.m_end);
      return m_find0 == rhs.m_find0;
    }
  private: // Implementation
    void increment_()
    {
      STLSOFT_MESSAGE_ASSERT( "Attempting to increment an invalid
iterator", m_find0 != m_end);
      if(blanks_policy_type::value) // Blank skipping/preserving
      {
        for(m_find0 = m_next; m_find0 != m_end; )
        {
          if(comparator_type::not_equal(*m_delimiter, m_find0))
          {
            break;
          }
          else
          {
            m_find0 +=  static_cast<ptrdiff_t>(m_cchDelimiter);
          }
        }
      }
      else
      {
        m_find0 = m_next;
      }
      for(m_find1 = m_find0; ; ) // Main tokenization loop
      {
        if(m_find1 == m_end)
        {
          m_next = m_find1;
          break;
        }
        else if(comparator_type::not_equal(*m_delimiter, m_find1))
        {
          ++m_find1;
        }
        else
        {
          m_next = m_find1 + static_cast<ptrdiff_t>(m_cchDelimiter);
          break;
        }
      }
    }
```

```
private: // Member Variables
  underlying_iterator_type  m_find0;        // Start of current token
  underlying_iterator_type  m_find1;        // End of current token
  underlying_iterator_type  m_next;         // Start of next item
  underlying_iterator_type  m_end;          // Endpoint of sequence
  delimiter_type const*     m_delimiter;    // Delimiter
  size_t                    m_cchDelimiter; // # chars in delimiter
};
. . .
```

The iterator maintains six member variables, the first four of which are of type `underlying_iterator_type`, the `const_iterator` type of the tokenized string type `S`. The `m_find0` and `m_find1` members demark the current iteration point, in a manner similar to the `p0` and `p1` variables used by `stlsoft::find_next_token()` (Section 27.5.3): The range [`m_find0`, `m_find1`) denotes the position and extent of the current token. Thus, the dereference operator simply invokes `traits_type::create()` with these two members to create and return the current value. The `m_next` member holds the start of the next token, the point at which the `m_find0` member will resume. `m_end` is the end of the tokenized string. `m_delimiter` points to the copy of the delimiter held in the tokenizer instance. `m_cchDelimiter` is the number of characters in the delimiter. For one-character delimiters and character delimiter sets this is 1; for composite string delimiters, it is the number of the characters in the delimiter.

The conversion constructor and the preincrement operator are both implemented in terms of the private worker method `increment_()`. This, in turn, is implemented in terms of the `blanks_policy_type::value` constant and the `comparator_type::not_equal()` method. The latter is passed a reference to the delimiter and a copy of the underlying iterator instance to determine whether the given underlying iterator instance refers to (the start of) a delimiter.

The tokenization algorithm is reasonably simple and is divided into two parts. The first part is concerned with the business of blank skipping. If blanks are to be skipped, `m_find0` is traversed until it doesn't point to (the start of) a delimiter. Otherwise, it's just set to start at the next point (`m_next`). The main part of the algorithm sets `m_find1` to the current point of `m_find0` and then proceeds to move `m_find1` to the (first character of the) next delimiter in the sequence. When the algorithm completes, the two members `m_find0` and `m_find1` define the range of the current token within the sequence.

27.6.2 Selecting Categories

The iterator category is forward. Since the tokenizer class does not alter the tokenized string, we'd be rightly surprised if its iterators could not support multipass semantics. To support any higher category would either be impossible or would add a level of complexity that's unlikely to be justified by any use case.

As we might expect, given that the tokenizer generates the tokens, the element reference category is by-value temporary. But this is not a fait accompli; we could have supported the *transient* element reference category (Section 3.3.4) by having each (non-endpoint) iterator instance maintain an instance of the current token, assigning it in the `increment_()` method. However, this could be inefficient in two ways. First, copying iterator instances would involve copying the cached current value. Where the string type is a bona fide string, this is a nontrivial cost. Since iterators are (at least in my experience) copied more often than they are dereferenced, this would be an anti-optimization. (This does not apply to input iterators, since, in sharing iterator state, they often also shared cache values.)

Second, holding an instance in this way would unnecessarily frustrate the compiler's ability to apply the Return Value Optimization in the case where each iterator is dereferenced at most once.

Tip: Favor the by-value temporary element reference category over the transient element reference category for collections whose value types must be synthesized intracollection, except where the iterator category is input iterator.

27.6.3 `stlsoft::string_tokeniser_type_traits`

The stock traits class template, `string_tokeniser_type_traits`, defines a few member types and three static functions, as shown in Listing 27.5. It serves the vast majority of tokenization scenarios.

Listing 27.5 Definition of `string_tokeniser_type_traits`

```
template< typename S
        , typename V
        >
struct string_tokeniser_type_traits
{
public: // Member Types
  typedef typename S::value_type        value_type;
  typedef typename S::const_iterator    const_iterator_type;
public: // Operations
  static const_iterator_type begin(S const& s)
  {
    return s.begin();
  }
  static const_iterator_type end(S const& s)
  {
    return s.end();
  }
  static V create(const_iterator_type from, const_iterator_type to)
  {
    return V(from, to);
  }
};
```

If you want to tokenize with string types that define the two member types, two nonmutating accessor methods, and one range constructor (all highlighted) that are required by the primary template, you have two choices. You can specialize `stlsoft::string_tokeniser_type_traits` for your type or provide a custom type traits parameter in your specialization of the tokenizer. An example of the former approach is shown in Listing 27.6, which allows `CString` to be used with the tokenizer.

Listing 27.6 Specialization of `string_tokeniser_type_traits` for MFC's `CString`

```
// In namespace stlsoft
template <>
struct string_tokeniser_type_traits<CString, CString>
{
public: // Member Types
  typedef TCHAR     value_type;
  typedef LPCTSTR   const_iterator_type;
public: // Operations
  static const_iterator_type begin(CString const& s)
  {
    return s;
  }
  static const_iterator_type end(CString const& s)
  {
    return begin(s) + s.GetLength();
  }
  static CString create(const_iterator_type from
                      , const_iterator_type to)
  {
    return CString(from, to - from);
  }
};
```

With a specialization defined, using `string_tokeniser` with `CString` is as simple as it is for any other string class:

```
string_tokeniser<CString, char>  tokens("abc;def;ghi;;jkl;;;", ';');
std::copy(tokens.begin(), tokens.end()
        , std::ostream_iterator<CString>(std::cout, "\n"));
```

27.6.4 `stlsoft::string_tokeniser_comparator`

Comparator policy classes are as conceptually simple as the traits class. From Listing 27.4 we can see that all they are required to provide is the two functions `not_equal()` and `length()`, as follows:

```
template <????>
struct arbitrary_comparator
{
  typedef ????  delimiter_type;
  template <typename const_iterator>
  static bool   not_equal(delimiter_type const& delim
                        , const_iterator&       it);
  static size_t length(delimiter_type const& delim);
};
```

However, it's not quite as simple as that. In order to cater to different string types, and to heterogeneous mixes of string type (`S`) and delimiter type (`D`), the stock comparator policy class takes the three template parameters `S`, `D`, and `T` that are specified to the tokenizer and provides a number of overloads of its private worker methods in order to cater to `char` (and `wchar_t`) delimiter types (as shown in Listing 27.7). The alternative would be a lot more jiggery-pokery in matching comparator to delimiter type. This would have required partial template specialization, and the tokenizer was written a long time before most compilers had that, so the pragmatic choice was to overload the methods in a common class template.

Listing 27.7 Definition of `string_tokeniser_comparator`

```
template <typename D, typename S, typename T> // Same as tokeniser
struct string_tokeniser_comparator
{
public: // Member Types
  typedef D                                          delimiter_type;
  typedef S                                          string_type;
  typedef T                                          traits_type;
  typedef typename traits_type::const_iterator_type  const_iterator;
private:
  typedef string_tokeniser_comparator<D, S, T>       class_type;
private: // Implementation
  template <typename I1, typename I2>
  static bool is_equal_(I1 p1, I2 p2, size_t n)
  {
    for(; n– > 0; ++p1, ++p2)
    {
      if(*p1 != *p2)
      {
        return false;
      }
    }
    return true;
  }
  template <typename D2, typename I>
  static bool is_equal_(D2 const& delim, I& p2)
  {
    return class_type::is_equal_(delim.begin(), p2, delim.length());
  }
  static bool is_equal_(char const delim, const_iterator& it)
  {
    return delim == *it;
  }
  static bool is_equal_(wchar_t const delim, const_iterator& it)
  {
    return delim == *it;
  }
  template <typename D2>
  static size_t get_length_(D2 const& delim)
  {
    return delim.length();
  }
  static size_t get_length_(char /* delim */)
  {
    return 1;
  }
  static size_t get_length_(wchar_t /* delim */)
  {
    return 1;
  }
public: // Operations
  static bool not_equal(delimiter_type const& delim
                      , const_iterator&       it)
  {
    return !is_equal_(delim, it);
  }
  static size_t length(delimiter_type const& delim)
  {
    return get_length_(delim);
  }
};
```

27.7 Test Drive

Now it's time to take this puppy out and see how it drives.

27.7.1 Single-Character Delimiter

This scenario involves choosing a string type and a corresponding character type. The following example splits the contents of a `std::string` with the delimiter `';'` and prints the results. The results would be *"abc,def,ghi,"*.

```
std::string                                       str = ";abc;;def;ghi";
stlsoft::string_tokeniser<std::string, char>  tokens(str, ';');
std::copy(tokens.begin(), tokens.end()
        , std::ostream_iterator<std::string>(std::cout, ","));
```

The equivalent could be achieved with some of the other components. First, with `strtok()`:

```
std::string   str = ";abc;;def;ghi";
char*         writeableCopy = ::strdup(str.c_str());
char*         tok;
for(tok = ::strtok(string, ";"); NULL != tok;
    tok = ::strtok(NULL, ";"))
{
  std::cout << tok << ",";
}
::free(writeableCopy);
```

Using `strtok_r()` would be an identical form, except for the third state parameter.

Using `std::stringstream` would be a virtually identical form to the example shown in Section 27.5.2, except that the token (an instance of `std::string`) would have to be tested to determine whether it was blank (`empty()`), as in the following:

```
. . . // Declare stream "sstm" and string "tok"
while(std::getline(sstm, tok, ';'))
{
  if(!tok.empty())
  {
    std::cout << tok << ",";
  }
}
```

Using `stlsoft::find_next_token()` would look as follows:

```
std::string str = ";abc;;def;ghi";
char const* p0  = str.c_str(); // p0 valid unless str changes!
char const* p1  = p0;
for(; stlsoft::find_next_token(p0, p1, ';'); )
{
  if(p1 != p0)
  {
    std::cout.write(p0, p1 - p0) << ",";
  }
}
```

And here's `boost::tokenizer`:

```
std::string   str = ";abc;;def;ghi";
boost::tokenizer<boost::char_separator<char> >
              tokens(str, boost::char_separator<char>(";"));
std::copy(tokens.begin(), tokens.end()
        , std::ostream_iterator<std::string>(std::cout, ","));
```

27.7.2 Composite String Delimiter

This scenario involves choosing a string type and a corresponding character type. The following example loads a file as a memory-mapped file, tokenizes the full contents en bloc, and then passes the sequence of lines to a notional `process_lines()` function. This is equivalent to the original motivating case for the composite string delimiter.

```
winstl::memory_mapped_file  mmf("H:\\temp\\big-test-file.cpp");
stlsoft::string_tokeniser<stlsoft::simple_string
                        , stlsoft::simple_string
                        >   lines(static_cast<char const*>(mmf.memory())
                                , mmf.size(), "\r\n");
process_lines(lines.begin(), lines.end());
```

There's a far more compellingly efficient permutation of this scenario. In the following case, rather than using a bona fide string class, a string view (`stlsoft::string_view`) is used, for both string type, `S`, and delimiter type, `D`. (Since the value type, `V`, defaults to the string type, it too is a string view.)

```
platformstl::memory_mapped_file mmf("H:\\temp\\big-test-file.cpp");
stlsoft::string_view       view( static_cast<char const*>(mmf.memory())
                               , mmf.size());
stlsoft::string_tokeniser<stlsoft::string_view
                        , stlsoft::string_view
                        >  lines(view, "\r\n");
process_lines(lines.begin(), lines.end());
```

Note that, apart from the committal of operating system pages inherent in the memory mapping, there is *no memory allocation* in this code.

If we are tokenizing on UNIX, we don't need a composite string delimiter:

```
platformstl::memory_mapped_file mmf("~/temp/big-test-file.cpp");
stlsoft::string_view       view( static_cast<char const*>(mmf.memory())
                               , mmf.size());
stlsoft::string_tokeniser<stlsoft::string_view
                        , char
                        >  lines(view, '\n');
process_lines(lines.begin(), lines.end());
```

27.7.3 Preserving Blanks

We can amend the earlier example to preserve blanks by explicitly stipulating the third template parameter, as shown here. The result in this case is *",abc,,def,ghi,"*.

```
std::string                      str = ";abc;;def;ghi";
stlsoft::string_tokeniser<std::string
                          , char
                          , stlsoft::skip_blank_tokens<false>
                          >       tokens(str, ';');
std::copy(tokens.begin(), tokens.end()
        , std::ostream_iterator<std::string>(std::cout, ","));
```

27.7.4 Copying versus Referencing: Considering Views

So far we've seen the tokenizer used with bona fide string classes and with string views. The former approach conducts full copies of the source string; the latter takes no copies whatsoever and merely holds pointer(s) into the source string. But `string_tokeniser` provides for separate specification of the string type (`S`) and the value type (`V`) in order to suit different scenarios.

Where possible, I favor use of string views for efficiency, but they have nontrivial restrictions. A string view's underlying contents must be stored contiguously, and they must persist for its lifetime. There are four permutations to consider.

27.7.4.1 `S` = view, `V` = view

This configuration is appropriate when the source string is a contiguous array of bytes and we have control over its lifetime. Combination with a `memory_mapped_file` is a perfect example of this.

Another good use of this permutation is when you wish to perform basic checks on a sequence of tokens. Consider the following function and the associated test code:

```
bool contains_token(char const* str, char delim, char const* tok)
{
  tokeniser_t   tokens(str, delim);
  return tokens.end() != std::find(tokens.begin(), tokens.end(), tok);
}

bool  b = contains_token(";abc;def;;ghi;;jkl;", ';', "jkl");
```

If `tokeniser_t` is a specialization involving a string type (`S`) with value semantics, for example, `stlsoft::string_tokeniser<stlsoft::simple_string, char>`, the code above will result in at least six memory allocations, each of which is associated with at least one invocation of `memcpy()` or equivalent functionality. That's an awful lot of effort for one simple check. Conversely, if `tokeniser_t` is `stlsoft::string_tokeniser<stlsoft::string_view, char>`, there are no allocations and no calls to `memcpy()`.

You might think an ugly but direct alternative would be preferable, that is, to search for the substring *`";jkl;"`*, but this would also need to search for the trailing substring *`";jkl"`* and the leading substring *`"jkl;"`*, *and* you'd need to test the entire string against *`"jkl"`*.

27.7.4.2 `S` = string, `V` = view

This configuration is appropriate when the source string is not long-lived, meaning that we need to take a copy of it. This situation comes up when the source string is a nonstring type, whose string equivalent form will be available, temporarily, to the tokenizer via the string access shims used in the constructor. Furthermore, the string type (`S`) must store its contents contiguously. Thus, you cannot use `std::string` for the string type (`S`) in conjunction with a string view as the value type (`V`), even though every implementation of `std::string` (that I know of) in common use stores its elements contiguously.

27.7.4.3 `S` = view, `V` = string

This permutation is appropriate when you can use a view for the string type but you wish to make copies of the values obtained by dereferencing the tokenizer's iterator that will persist beyond the lifetime of the tokenizer instance. (When this is the case, there's little advantage in returning a string view if a bona fide string instance is required by the client code; compilers should be able to apply the Return Value Optimization.) This is illustrated by the following code:

```
stlsoft::string_tokeniser<stlsoft::string_view
                        , stlsoft::string_view
                        , stlsoft::skip_blank_tokens<true>
                        , std::string
                        >     lines(view, "\r\n");
std::vector<std::string>      nonBlankLines(lines.begin(), lines.end());
```

This permutation is also suitable when you need to modify the values locally, before passing them off to other code, as shown in the following code. Using a string view for the value type in this case would be just an inconvenience.

```
. . . // Same specialization as above
for(tokeniser_t::const_iterator b = lines.begin(); b != lines.end();
    ++b)
{
  nonBlankLines.push_back("[" + *b + "]");
}
```

27.7.4.4 `S` = string, `V` = string

This permutation is appropriate when the other three are not, hence it corresponds to the majority of cases.

27.7.5 Character Set Delimiter

I expect you'll agree that the examples so far demonstrate that `string_tokeniser` is powerful, simple, and straightforward to use. And I'll show you later in the chapter how efficient it is, too. So what of the portents of doom I made in the introduction?

The next case readily illustrates the problem. Listing 27.8 shows a specialization of the tokenizer required to emulate the character set delimiting offered by `strtok()`, `strtok_r()`, and `boost::tokenizer`. If you're stuck for words, let me help you out: Yikes!

Listing 27.8 Definition of `charset_comparator` and Client Code

```
// In namespace stlsoft
template <typename S>
struct charset_comparator
{
  typedef S  delimiter_type;
  template <typename const_iterator>
  static bool not_equal(delimiter_type const& delim, const_iterator& it)
  {
    return delim.end() == std::find(delim.begin(), delim.end(), *it);
  }
  static size_t length(delimiter_type const& delim)
  {
    return 1;
  }
};

typedef stlsoft::string_tokeniser<string_t
                      , string_t
                      , stlsoft::skip_blank_tokens<true>
                      , string_t
                      , stlsoft::string_tokeniser_type_traits<string_t
                                                             , string_t
                                                             >
                      , stlsoft::charset_comparator<string_t>
                      >           tokeniser_t;
char*       str = . . .
tokeniser_t tokens(str, " \r\n\t");
std::copy(tokens.begin(), tokens.end()
        , std::ostream_iterator<std::string>(std::cout));
```

Contrast that with the equivalent `strtok()` example given earlier, and you'd be forgiven for thinking STL's gone mad. In fact, the problem is far more prosaic: The `string_tokeniser` class template is not well designed. Thankfully, there are resolutions to this problem.

27.8 The Policy Folly

My acknowledgment of the design problems is not mere stereotypical English self-deprecation. I'm quite sincere. Note, however, that I didn't say that the class template was badly designed. In fact, it's fair to say that it wasn't really *designed* much at all. In the process of going from the definition of `SynesisSTL::StringTokeniser` (Listing 27.1) to being able to handle composite string delimiters (Section 27.4), the class template design largely ceased, and its evolution began.

By the time I came back to consider the use of character set delimiters, it had taken root, and it was too late to change. Such design by intuition is not always recommended since it leaves you in messes such as this. To be sure, if you're experienced in writing libraries, you probably work this way much of the time without realizing it, and you'll often get away with it because your hard-won instincts are good and mostly well aimed.

But even when an evolutionary development results in a less-than-perfect component, as in this case, there may remain some vestiges of a wise insight spread around between the haphazardry. Here, in what appears a moment of fantastic prescience on my part, I included a comparator policy. At the time—late 1990s—I was riding the crest of the policy wave, like many of my colleagues. So I can only assume I stuck in a comparator because it seemed like a good idea to have a comparison policy. Thankfully, this accidental cleverness can be harnessed into rescuing the component from the black hole of unusability by a simple, but often overlooked, technique: inheritance, as we'll see in the next section. But tempering any desire to slap myself on the back here is the total failure of the comparator facility to fulfill pair delimiter tokenization with the current definition of the tokenizer.

27.8.1 Refactoring Template Parameters via Inheritance

We already have the `charset_comparator` (Section 27.6.4), but using it with `string_tokeniser` is horrible, largely as a consequence of the template parameter ordering. A simple and effective way to reorder template parameters is to define a new template, `charset_tokeniser`, which is implemented in terms of the existing one. The question is whether we use inheritance or composition. In this case, we're not introducing any state or any new runtime functionality to the new class, so there's no impediment to using inheritance, and it allows the compiler to apply the *Empty Derived Optimization* (see Section 12.4 of *Imperfect C++*).

Using inheritance also relieves us of the effort involved in writing forwarding functions for `begin()`, `end()`, and `empty()`. It does not, however, relieve us of this same task for the constructors, but they would be required using either strategy. Listing 27.9 shows the full implementation of `stlsoft::charset_tokeniser`.

Listing 27.9 Definition of `charset_tokeniser`

```
// In namespace stlsoft
template< typename S
        , typename B = skip_blank_tokens<true>
        , typename V = S
        , typename T = string_tokeniser_type_traits<S, V>
        , typename D = S
        , typename P = charset_comparator<S>
        >
class charset_tokeniser
  : public string_tokeniser<S, D, B, V, T, P>
{
private: // Member Types
  typedef string_tokeniser<S, D, B, V, T, P>          parent_class_type;
public:
  typedef charset_tokeniser<S, B, V, T, D, P>         class_type;
  typedef typename parent_class_type::string_type    string_type;
  . . . // And for delimiter_type, value_type, and so on
  typedef typename parent_class_type::const_iterator const_iterator;
public: // Construction
  template <typename S1>
  charset_tokeniser(S1 const& str, delimiter_type const& charSet)
    : parent_class_type(str, charSet)
  {}
  template <typename S1>
  charset_tokeniser(S1 const& str, size_type n
                  , delimiter_type const& charSet)
    : parent_class_type(str, n, charSet)
  {}
  template <typename I>
  charset_tokeniser(I from, I to, delimiter_type const& charSet)
    : parent_class_type(from, to, charSet)
  {}
};
```

27.8.2 Type Generator Templates

Although C++ does not (yet) support typedef templates, it is possible to approximate their utility with type generator templates (Section 12.2), which provide an alternative solution to our character delimiter set problem, as shown in Listing 27.10.

Listing 27.10 Type Generator Template for Character Set Tokenizer

```
// In namespace stlsoft
template< typename S
        , typename B = skip_blank_tokens<true>
        , typename V = S
        , typename T = string_tokeniser_type_traits<S, V>
        , typename D = S
        , typename P = charset_comparator<S>
        >
struct charset_tokeniser_selector
{
public: // Member Types
  typedef string_tokeniser<S, D, B, V, T, P>    tokeniser_type;
};
```

We've seen one of these generator templates already: `allocator_selector` (Section 12.2.1). In this case, the selection is between alternate specializations of the same template, rather than between identical specializations of alternate templates. But the overall intent is the same: The template generates a type on behalf of its user.

```
stlsoft::charset_tokeniser_selector<std::string>::tokeniser_type
                      tokens("\rabc def\nghi\tjkl  ", " \r\n\t");

std::copy(tokens.begin(), tokens.end()
        , std::ostream_iterator<std::string>(std::cout, "\n"));
```

The advantages to this form are that the generator template is small, succinct, and easy to produce. Further, it does not introduce *any* runtime code, so theoretically it introduces no new code to test. The disadvantages are that, compared with a class template, a generator template is somewhat cumbersome to use, less easy to grok for users who are not familiar with generator templates, and more likely to slip under the radar of users hunting for such facilities. (You can grep for *`"\s*class\s+\w*token"`* to find any classes that involve tokenization.)

27.8.3 Handling *Henney's Hypothesis*

We've looked at two relatively simple solutions to a violation of *Henney's Hypothesis* resulting from the evolution (and incomplete early design) of a component. Both are readily comprehensible, and neither requires any kind of metaprogramming. For the **STLSoft Tokenising** library, I chose the inheritance-based solution. In Volume 2, we'll look at more advanced techniques for handling these kinds of situations. For now, we'll content ourselves with an adequate solution and move on to see how the various components perform.

27.9 Performance

There are nine tokenization scenarios, as follows:

1. Short string (2 tokens), single-character delimiter, preserving blanks
2. Medium string (6 tokens), single-character delimiter, skipping blanks
3. Medium string, single-character delimiter, preserving blanks
4. Long string (60 tokens), single-character delimiter, skipping blanks
5. Medium string, composite string delimiter, skipping blanks
6. Medium string, composite string delimiter, preserving blanks
7. Long string, composite string delimiter, skipping blanks
8. Medium string, character set delimiter, skipping blanks
9. Long string, character set delimiter, skipping blanks

Where they support the required functionality, I used the following tokenizer components:

A. `stlsoft::string_tokeniser / charset_tokeniser` (s = string, v = string)
B. `strtok()`
C. `strtok_r()` (a custom-written implementation)
D. `std::stringstream + std::getline()`
E. `stlsoft::find_next_token()`

F. `boost::tokenizer`
G. `stlsoft::string_tokeniser / charset_tokeniser` (`S` = **string,** `V` = **view**)
H. `stlsoft::string_tokeniser / charset_tokeniser` (`S` = **view,** `V` = **string**)
I. `stlsoft::string_tokeniser / charset_tokeniser` (`S` = **view,** `V` = **view**)

Since `std::getline()` and `stlsoft::find_next_token()` always preserve blanks, I've added a simple test on the token size to emulate blank skipping. Obviously, the reverse was not possible for those components that always skip blanks. The tokenization processing comprised three actions, repeated 100,000 times, and their combined time was measured. The three actions were functionally equivalent to the following operations:

```
std::copy(tokens.begin(), tokens.end(), . . . );
size_t  n1  = std::distance(tokens.begin(), tokens.end());
size_t  n2  = std::accumulate(tokens.begin(), tokens.end()
                             , 0, invoke_length());
```

where `invoke_length` is a binary function object that sums an integral operand with the length of the string operand. The test thread was run at high priority in order to avoid perturbation by other processes, and the times were taken on the second repeat of the overall scenario to avoid effects of processor caching, memory paging, and so on. The results are shown for both Visual C++ 7.1 and Intel C/C++ version 8.0; several other compilers were tested, and the results were similar in all cases. The results (Tables 27.1 and 27.2) are expressed as a percentage of the time taken for the `string_tokeniser` (scenarios 1–6) / `charset_tokeniser` (scenarios 7–9) with `S` = S (string), `V` = S (string).

In most cases, the results reflect expectations. For the moment we'll put aside the variants of `string_tokeniser` that use string views. Of the other six tokenizing components, `stlsoft::find_next_token()` takes the honors, in those scenarios in which it is meaningful, by a country mile. This shouldn't be a surprise, as it is effectively a combination of a nonmutating, noncopying mechanism that pushes some manual processing (e.g., `if(p1 != p0)`) out to the client code.

Another result that's not surprising is that `strtok()` performs better than `strtok_r()`, `stlsoft::string_tokeniser` (when using strings for both `S` and `V`), `std::stringstream`, and `boost::tokenizer`. Interestingly, `strtok_r()` performs better than `string_tokeniser` with single character delimiter tokenization, but worse in character delimiter set tokenization; I imagine this is because the implementation of `strtok_r()` (included on the CD) uses the C library function `strpbrk()` and so is not as amenable to optimization as the inline code of the header-only C++ components.

I think it's fair to say that `std::stringstream` is unlikely to be anyone's first choice for tokenization, whether performance or feature set is the prime concern. It's competitive on a couple of occasions, but with different scenarios for the two compilers. As predicted, the `boost::tokenizer` component is also penalized by both compilers for its complexity, and it's not likely to be a first choice for single-character delimiter or character set delimiter tokenization when performance is an issue. But, as I mentioned earlier, it has advanced capabilities for tokenization that none of the others can provide, in which cases it would prove to be infinitely faster.

Table 27.1 Relative Performances of String Tokenization Components (Visual C++)

Visual C++ 7.1	**Single-Character Delimiter**				**Composite String Delimiter**			**Character Set Delimiter**	
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`strtok()`		51.1%		33.7%				60.8%	49.0%
`strtok_r()`		55.4%		42.6%				132.7%	140.3%
`stringstream` + `getline()`	188.3%	227.7%	107.6%	152.3%					
`stlsoft::find_next_token()`	3.7%	9.5%	4.6%	9.9%					
`boost::tokenizer`		198.2%		152.8%				247.1%	234.1%
`string_tokeniser` (S, V)	18.9%	18.7%	10.7%	13.2%	47.8%	28.8%	27.2%	56.4%	45.8%
`string_tokeniser` (V, S)	89.7%	91.3%	97.6%	97.9%	94.3%	98.3%	99.8%	101.3%	117.3%
`string_tokeniser` (V, V)	8.2%	11.9%	8.1%	12.7%	22.7%	17.3%	26.1%	33.0%	38.6%

Table 27.2 Relative Performances of String Tokenization Components (Intel)

Intel C/C++ 8.0	**Single-Character Delimiter**				**Composite String Delimiter**			**Character Set Delimiter**	
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`strtok()`		48.4%		30.7%				59.5%	36.2%
`strtok_r()`		55.8%		39.8%				137.0%	107.8%
`stringstream` + `getline()`	271.8%	202.6%	172.6%	97.3%					
`stlsoft::find_next_token()`	10.0%	15.3%	12.9%	15.4%					
`boost::tokenizer`		145.6%		102.9%				172.8%	117.8%
`string_tokeniser` (S, V)	23.9%	15.6%	17.5%	9.2%	41.6%	41.1%	16.1%	48.5%	28.9%
`string_tokeniser` (V, S)	81.8%	90.8%	90.8%	93.9%	86.1%	88.1%	65.1%	98.1%	89.6%
`string_tokeniser` (V, V)	7.5%	8.0%	11.0%	8.5%	18.5%	18.6%	15.0%	27.0%	23.9%

The cases I find most interesting involve string views. Using a bona fide string type (one with value semantics) for the collection string type (`S`), rather than a string view, accounts for about 10% of the performance cost, while using it for the value type (`V`) accounts for 40% to 90%. Averaging out the relative performances, where data points exist for both comparands, over the two compilers, it appears that `string_tokeniser` using string views is about on a par with `find_next_token()`, 3 times as fast as `strtok()`, 8.5 times as fast as `string_tokeniser` with bona fide strings, 10.5 times as fast as `boost::tokenizer`, and a whopping 20 times as fast as `std::stringstream` / `std::getline()`. And `string_tokeniser` using string view only for the value type (`V`) is only about 50% slower. As long as you don't need to convert the value into a nul-terminated string or copy it into a bona fide string type, the `string_tokeniser` using string view represents a compelling combination of performance and STL collection features.

27.10 Summary

It's important to realize that it's not always possible to reach an acceptably optimal solution. There can be several reasons for this.

- We might be attempting to provide too much functionality in one component.
- We might have reached a local maximum, and the prospect of attempting to backtrack and start again is understandably daunting.
- We might need to maintain backward compatibility.
- It might not even be clear that there is a superior solution, let alone what that might be.

With `string_tokeniser`, it's clear that it's stuck in a local maximum. Getting it to address tokenization via character set delimiter as well as it does with single-character and composite string delimiters would require a serious rewrite. Despite its template parameter list being a discoverability problem, I've resisted the temptation to enhance the functionality of the tokenizer for a long time, simply because it is well tested, easy to use (in most scenarios), and very fast. A rewrite is out of the question, and not only because its correctness would have to be reestablished. Using decent enough testing techniques, rigorously applied, this is achievable. Indeed, this is one of the areas in which test-driven development shines since your tests can directly compare old and new forms of a component being reworked. But it's a lot harder to design in performance characteristics that have had as much evolution as those presented here. To be frank, the cost/reward balance has never been there. (Of course, if some talented reader were stimulated to produce such a work, I would be *very* happy to incorporate it into **STLSoft**.) Finally, and probably most significantly, the ameliorating measure of defining `charset_tokeniser`, while not glamorous, is eminently effective. Sometimes the prosaic solution suffices.

C H A P T E R 2 8

Adapting COM Enumerators

A subtle thought that is in error may yet give rise to fruitful inquiry that can establish truths of great value.

—Isaac Asimov

Smoke me a kipper, I'll be back for breakfast!

—Ace Rimmer, *Red Dwarf*

28.1 Introduction

One of the technical reasons that COM has fallen out of favor in this millennium is the extreme verbosity of code in C++. Two more are the complexity of reference-count management and the relative ease with which users can mismanage resource ownership, both of which can lead to leaks and/or double deletion. *COM enumerators* and *COM collections* are two of the major problem areas. This is compounded by the fact that, because they are language-independent, COM interface methods may not throw exceptions, which, as we well know, C++ functions and methods may and do.

In this chapter, we're going to look at COM enumerators, as represented by the ***IEnumXXXX*** protocol, and discuss the **COMSTL** `enumerator_sequence` class template, which nicely encapsulates all the negative aspects of COM enumerators in an *STL collection* (Section 2.2). We'll see how COM's enumerators naturally support neither *input* nor *forward iterator* categories (Section 1.3) and look at techniques for reliably and meaningfully supporting input or forward iterator semantics, as directed by the user. We'll look at the dangers of caching elements in iterators. And we'll see how value type policies can be used to tailor sequences for manipulation of types with a variety of initialization, copying, and destruction semantics.

28.2 Motivation

In customary style, we begin with a comparison of using the underlying collection API in its raw form and in the form of an STL collection, contrasting the costs and benefits of the two approaches. For our example we'll use the COM wrapper of the **recls** library, which allows you to conduct recursive file system searches from any COM-enabled language. The **recls**/**COM** enumerator interface is `IEnumFileEntry`, with which you enumerate file system entries (represented by the `IFileEntry` interface).

28.2.1 Longhand Version

For the sake of brevity, we'll assume we have a function open_search(), used to conduct searches, declared as follows:

```
HRESULT open_search(char const*       directory
                  , char const*       patterns
                  , long              flags
                  , IEnumFileEntry**  ppenum) throw();
```

The function takes three search parameters and returns to the caller an instance of an enumerator object from which the search results may be elicited. The semantics of the directory + patterns + flags are the same for all **recls** language mappings: directory is the root directory of the search; patterns is one or more search patterns, separated by the pipe character, for example, "*.cpp|*.hpp|?akefile"; and flags is a combination of search flags, requesting files and/or directories, specifying recursive action, and so on. The function returns the standard COM error type HRESULT, which is a 32-bit signed integer whose bits are divided into blocks representing success/failure, the error code group, and the error code. COM defines the macros SUCCEEDED() and FAILED(), with which you may interpret the success/failure represented by an HRESULT without more detailed treatment.

Now let's say that we want to use **recls**/**COM** to look for all the C++ source files in the current directory. Listing 28.1 shows how we might achieve this in regular C++. (Note the use of comstl::propget_cast, which helps in further reducing the boilerplate. This function invokes the given method and returns the retrieved value of the specializing type.)

Listing 28.1 Use of a COM Enumerator via Raw Interfaces

```
IEnumFileEntry* pe;
HRESULT         hr   = open_search("."
                       , "*.c|*.cpp|*.h|*.hpp|*.d|*.rb|*.java|*.cs|*.py"
                       , RECLS_F_FILES | RECLS_F_RECURSIVE
                       , &pe);
if(SUCCEEDED(hr))
{
  IFileEntry* entries[5];
  ULONG       numRetrieved = 0;
  for(pe->Reset(); SUCCEEDED(pe->Next(STLSOFT_NUM_ELEMENTS(entries)
                                    , &entries[0], &numRetrieved)); )
  {
    if(0 == numRetrieved)
    {
      break;
    }
    for(ULONG i = 0; i < numRetrieved; ++i)
    {
      std::wcout << comstl::propget_cast<bstr>(entries[i]
                               , &IFileEntry::get_Path) << std::endl;
      entries[i]->Release();
    }
  }
  pe->Release();
}
```

We'll look in more detail at what's going on here later. For the moment, I will comment that, in addition to a lack of transparency, this code is fragile and not exception-safe in two areas.

28.2.2 Shorthand Version

Listing 28.2 is the shorthand version, using the `comstl::enumerator_sequence` class template and the `comstl::interface_policy` template.

Listing 28.2 Use of a COM Enumerator via `enumerator_sequence`

```
IEnumFileEntry* pe;
HRESULT         hr  = open_search("."
                    , "*.c|*.cpp|*.h|*.hpp|*.d|*.rb|*.java|*.cs|*.py"
                    , RECLS_F_FILES | RECLS_F_RECURSIVE
                    , &pe);
if(SUCCEEDED(hr))
{
  typedef comstl::enumerator_sequence<IEnumFileEntry
                                   , IFileEntry*
                                   , comstl::interface_policy<IFileEntry>
                                   >          enum_t;
  enum_t  entries(pe, false, 5); // Eat the reference; batches of 5
  for(enum_t::iterator b = entries.begin(); entries.end() != b; ++b)
  {
   std::wcout << comstl::propget_cast<bstr>(*b,&IFileEntry::get_Path)
              << std::endl;
  }
}
```

In this case, the entry enumeration is clearer, all resources are nicely encapsulated, and the code is entirely exception-safe. The cost is in understanding the `enumerator_sequence` template specialization, which is somewhat low on the discoverability scale. (And we've still got a manual loop. In Section 36.1 we'll see how `enumerator_sequence` can be used in combination with the `transform_iterator` iterator adaptor and a custom function object to enable this to be more succinctly expressed using the `std::copy()` algorithm.)

28.3 COM Enumerators

Before we discuss these two versions, it's probably wise to do a quick refresher on COM enumerators. All COM enumerators have four methods: `Next()`, `Skip()`, `Reset()`, and `Clone()`. The definition of `IEnumFileEntry` is, in C++, as follows:

```
struct IEnumFileEntry
{
  virtual HRESULT Next( unsigned long     numRequested
                      , IFileEntry**      items
                      , unsigned long*    numRetrieved) = 0;
  virtual HRESULT Skip( unsigned long     numToSkip) = 0;
  virtual HRESULT Reset() = 0;
  virtual HRESULT Clone(IEnumFileEntry**  ppenum) = 0;
};
```

28.3.1 `IEnumXXXX::Next()`

This method is the way in which elements are retrieved and, simultaneously, by which the enumeration point is advanced. Because COM objects may be remote—whether in a different thread, process, host, or network—the method allows a number of elements to be retrieved at once to facilitate performance optimization by balancing round-trip time with per-element retrieval time. The `numRequested` parameter stipulates how many are requested, and therefore the extent of the array specified to the `items` parameter, and `*numRetrieved` receives the number actually retrieved and returned to the caller. As a concession to usability, `numRetrieved` may be *`NULL`* if and only if `numRequested` is *`1`*. If the function successfully retrieves the number requested, the standard COM success return code *`S_OK`* (*`0`*) is returned. If the function succeeds but retrieves fewer than the number requested, *including* `0`, a specific success code *`S_FALSE`* (*`1`*) is returned. (All failures return one of myriad error codes, with values < `0`, discriminated by the `FAILED()` macro). Due to issues related to marshalling—which is *way* beyond the scope of this book—you must not rely on the return value and should instead rely on the value of `*numRetrieved`.

28.3.2 `IEnumXXXX::Skip()`

This method advances the enumeration point without retrieving elements. The return code semantics are similar to those of `Next()`: If the enumeration point was moved `numToSkip` places, the return is *`S_OK`*; if the endpoint was reached prior to `numToSkip` moves, the return is *`S_FALSE`*. Otherwise, an error code is returned.

28.3.3 `IEnumXXXX::Reset()`

This method resets the enumeration point to the beginning. Note that COM does not require that an enumerator can be reset or that a subsequent enumeration after a successful reset will yield the same elements in the same order as the previous one.

28.3.4 `IEnumXXXX::Clone()`

The `Clone()` method is used to provide to the caller an enumerator instance separate from the one provided to the callee. The two enumerators hold ostensibly the same enumeration position but have separate enumeration state and act independently. This is to allow a user to record the current enumeration state at a particular point and return to that state after independent processing of the (remaining) elements. Note that COM does not require that an enumerator will support this method or that the sets of elements subsequently returned by the clone and the cloned instance will be the same. Remember this point—it holds important ramifications for adaptation to STL iterator concepts, which we'll cover in detail in Sections 28.8 and 28.9.

28.3.5 Different Value Types

Earlier I referred to ***IEnumXXXX*** as a protocol. Because COM is language-independent, it does not support templates, à la C++. Rather, it defines a protocol—`Next()`, `Skip()`, `Reset()`, and `Clone()`—that all enumerator interfaces must support. The actual interfaces are not related by type (other than that they all inherit from `IUnknown`, the root COM interface). But they do share a common structure. The `Clone()` method differs in that the parameter always corresponds to the interface itself. The `Skip()` and `Reset()` methods are identical across all interfaces. The `Next()` method differs in the type of elements returned to the caller, which we may reasonably call the enumerator's value type. The range of value types is unbounded. Well-known enumerator interfaces and their value types are listed in Table 28.1

Table 28.1 Well-Known Enumerator Interfaces and Corresponding Value Types

Interface	Value Type	Resource Release
`IEnumBSTR`	`BSTR`	Call `SysFreeString()`
`IEnumGUID`	`GUID`	—
`IEnumString`	`LPOLESTR (== wchar_t*)`	Call `CoTaskMemFree()`
`IEnumUnknown`	`IUnknown*`	Call `interface's Release()`
`IEnumVARIANT`	`VARIANT`	Call `VariantClear()`

It's important to remember that all values returned via calls to an enumerator interface's `Next()` method become the property of the caller. This is a nonissue with something like a `GUID`, which is just a 128-bit integer structure, but is very important when the value type is a resource. An interface pointer must be released via its `Release()` method:

```
LPUNKNOWN     punk;
IEnumUnknown* pen = . . .
for(pen->Reset(); S_OK == pen->Next(1, &punk, NULL); )
{
  punk->Release();
}
```

A `BSTR` instance must be passed to the `SysFreeString()` function:

```
BSTR         bstr;
IEnumBSTR*   pen = . . .
for(pen->Reset(); S_OK == pen->Next(1, &bstr, NULL); )
{
  ::SysFreeString(bstr);
}
```

An LPOLESTR instance must be deallocated by the CoTaskMemFree() function:

```
LPOLESTR      str;
IEnumString*  pen = . . .
for(pen->Reset(); S_OK == pen->Next(1, &str, NULL); )
{
  ::CoTaskMemFree(str);
}
```

A VARIANT instance must be passed to the VariantClear() function:

```
VARIANT       var;
IEnumVARIANT* pen = . . .
for(pen->Reset(); S_OK == pen->Next(1, &var, NULL); )
{
  ::VariantClear(&var);
}
```

Any class that abstracts the enumerator interfaces must handle such different resource types.

28.4 Decomposition of the Longhand Version

Armed with this COM lore, we can now look in detail at the longhand version shown earlier in Listing 28.1.

Line **8**: We declare an array of five IFileEntry pointers, so that we can retrieve en bloc from Next(). Although I can tell you that **recls/COM** search objects are created in the thread that calls them, in general you cannot make such assumptions. As such, retrieving, say, five to ten elements at a time is a compromise between apparent responsiveness of your program (as dictated by granularity of retrieval) and overall performance (by reducing round-trip times where information is retrieved extrathread).

Line **10**: Reset the enumerator. Although this is unnecessary in this specific case, in general it's good practice to ensure we're starting at the beginning.

Lines **10–11**: Invoke Next(), requesting as many elements as we have in the entries array. This is determined via the STLSOFT_NUM_ELEMENTS() macro (Section 5.1), supporting *DRY SPOT* (Chapter 5) to guard against destructive maintenance changes.

Tip: Specify the sizes of arrays using size-determining constructs, such as STLSOFT_NUM_ELEMENTS(), to avoid the dangers of magic numbers.

Lines **13–16**: If the call to Next() has succeeded, it is still possible to receive zero items, in which case we must break the loop.

Lines **17–22**: Each returned entry (between one and five) is processed.

Lines **19–20**: The path is elicited from the entry via the IFileEntry::get_Path() method and passed to the stream insertion operator. If the insertion operator throws an exception, the entry pointer(s) will not be released. Indeed, if, say, four elements have been returned, and the stream insertion throws on the second time around the inner loop, three objects will not be released.

Line **21**: The entry is released, via its `IUnknown::Release()` method.

Line **24**: The enumerator is released, via its `IUnknown::Release()` method. If any exception is thrown between the return from `open_search()` and this call, the enumerator object will not be released.

That's a lot of bark for not much bite. And there are several places where we could have taken a misstep. There are at least four common mistakes in code like this:

1. Testing whether the return value is equal to *`S_OK`*, rather than whether `SUCCEEDED()` evaluates *`true`*; if there were thirteen files, the return value would be *`S_FALSE`*`(`*`1`*`)`, in which case we'd retrieve only the first ten and we'd leak the other three reference counts.
2. Passing *`NULL`* for numRetrieved when numRequested is not `1`.
3. Forgetting to check whether the number retrieved is zero, resulting in an infinite loop.
4. Forgetting to release the returned resource, in this case to call `IUnknown::Release()` on each entry pointer.

The significant problem with all these mistakes is that they do not show up until runtime, and some of them might escape detection for a long time. Even done correctly, as shown, the code remains exception-unsafe. And of course, the use of enumerators requires handwritten loops, which counts out the use of algorithms.

28.5 comstl::enumerator_sequence

The **COMSTL** `enumerator_sequence` class template handles all these issues for us. It is constructed from an enumerator interface, from which it retrieves elements, providing an STL collection interface to those elements. It handles the resources, including the enumerator interface pointer, ensuring all are properly released upon destruction, and is exception-safe.

28.5.1 Public Interface

The public interface for `enumerator_sequence` is shown in Listing 28.3.

Listing 28.3 Definition of `enumerator_sequence`

```
// In namespace comstl
template< typename I                           // Enumerator interface
        , typename V                           // Value type
        , typename VP                          // Value policy type
        , typename R  = V const&               // Reference type
        , typename CP = input_cloning_policy<I> // Cloning policy type
        , size_t   Q  = 10                     // Quanta
        >
```

```
class enumerator_sequence
{
public: // Member Types
  typedef I                                           interface_type;
  typedef V                                           value_type;
  typedef value_policy_adaptor<VP>                    value_policy_type;
  typedef R                                           reference;
  typedef ????                                        pointer;
  typedef CP                                          cloning_policy_type;
  typedef typename CP::iterator_tag_type              iterator_tag_type;
  enum                                              { retrievalQuanta = Q };
  typedef enumerator_sequence<I, V, VP, R, CP, Q> class_type;
  typedef class_type                                  sequence_type;
  typedef size_t                                      size_type;
  class                                               iterator;
public: // Construction
  enumerator_sequence(interface_type* i
                    , bool            bAddRef
                    , size_type       quanta = 0
                    , bool            bReset = true);
  ~enumerator_sequence() throw();
public: // Iteration
  iterator begin() const;
  iterator end() const;
};
```

The first thing to note is the template parameter list with its six parameters. As discussed in Chapter 14, *Henney's Hypothesis* is always watching, and it is a harsh judge. Thankfully, the last three parameters are defaulted to types/values appropriate to most uses to which you might wish to put the sequence; we'll discuss these later. The three nondefault parameters are the enumerator interface (`I`), the value type (`V`), and the value type policy (`VP`).

28.5.2 Member Types and Constants

As is my wont, the type template parameters are used to define member types, `interface_type`, `value_type`, `reference`, and `cloning_policy_type`. The value policy parameter is used to specialize the `value_policy_adaptor`, which is explained shortly, to define the member type `value_policy_type`. The `pointer` type has a peculiar definition, which is discussed in the intermezzo following this chapter. All the remaining member types are standard fare, except `iterator_tag_type`, which is derived from the same named member type of the cloning policy and is used later in the declaration of the iterator class. The member constant `retrievalQuanta` is an enumerator assigned the value of the quanta template parameter (`Q`). This makes the value specified to the template available to client code.

28.5.3 Value Policies

Because different value types have different resource management requirements, the sequence uses a policy for determining how the value types should be manipulated. Each policy defines three static functions, `init()`, `copy()`, and `clear()`, as can be seen in Listing 28.4, which shows the definitions for three stock policies. All stock policies are defined within the `comstl` namespace. Note that the `init()` and `clear()` methods are required to *not* throw exceptions, whereas `copy()` must be able to do so; we'll see why later.

Listing 28.4 Stock Value Policies Provided by COMSTL

```
// In namespace comstl
struct GUID_policy
{
public: // Member Types
  typedef GUID        value_type;
public: // Operations
  static void init(value_type*) throw()
  {}
  static void copy(value_type* dest, value_type const* src)
  {
    *dest = *src;
  }
  static void clear(value_type*) throw()
  {}
};
struct VARIANT_policy
{
public: // Member Types
  typedef VARIANT value_type;
public: // Operations
  static void init(value_type* p) throw()
  {
    ::VariantInit(p);
  }
  static void copy(value_type* dest, value_type const* src)
  {
    HRESULT hr = ::VariantCopy(dest, const_cast<VARIANT*>(src));
    if(FAILED(hr))
    {
      throw com_exception("failed to copy VARIANT", hr);
    }
  }
  static void clear(value_type* p) throw()
  {
    ::VariantClear(p);
  }
};
template <typename I>
```

```
struct interface_policy
{
public: // Member Types
  typedef I               interface_type;
  typedef interface_type* value_type;
public: // Operations
  static void init(value_type* p) throw()
  {
    *p = NULL;
  }
  static void copy(value_type* dest, value_type const* src)
  {
    *dest = *src;
    if(NULL != *dest)
    {
      (*dest)->AddRef();
    }
  }
  static void clear(value_type* p) throw()
  {
    if(NULL != *p)
    {
      (*p)->Release();
      *p = NULL;
    }
  }
};
```

For the simple structure `GUID`, which does not handle resources, there is nothing required of the `init()` and `clear()` methods, and copying is carried out via assignment, overwriting any previous value. Contrast this with the case of `VARIANT_policy`, where the memory it will occupy must be initialized with `VariantInit()` and cleaned up with `VariantClear()`, and copying must be via `VariantCopy()`. The third policy, `interface_policy`, is a class template that is parameterized with the given interface, wherein instances are copied via `AddRef()` and released via `Release()`.

You might be wondering why these policies are adapted via the `value_policy_adaptor`. The reason becomes clear when we look at the implementation of the nested class `enumerator_sequence::iterator::enumeration_context`. Two methods, `init_elements_()` and `clear_elements_()`, use the standard algorithm `for_each()` to process the elements en bloc, as shown in Listing 28.5.

Listing 28.5 `enumeration_context` **Methods**

```
  . . .
  struct enumeration_context
  {
    . . .
    void init_elements_(size_type n) throw()
    {
      COMSTL_ASSERT(n <= STLSOFT_NUM_ELEMENTS(m_values));
      std::for_each(&m_values[0], &m_values[0] + m_acquired
                  , typename value_policy_type::init_element());
    }
    void clear_elements_() throw()
    {
      COMSTL_ASSERT(m_acquired <= STLSOFT_NUM_ELEMENTS(m_values));
      std::for_each(&m_values[0], &m_values[0] + m_acquired
                  , typename value_policy_type::clear_element());
    }
```

In order to do this, a function object is required that can take an argument of type `value_type&`. But the policy functions take pointers. The original implementation of the `enumerator_sequence` didn't use algorithms; rather, it invoked the policy function directly in manual loops. Once this component and the policies were released, changing the policy interfaces to mandate the use of references rather than pointers would have broken existing code that used user-defined policies; hence the need for the adaptors. And there's another reason at play here: I tend to think that passing a pointer to a policy is a clearer indication that raw memory is lurking, needing to be initialized, whereas passing a reference can indicate access to an object that is already initialized. Consequently, I am content with the way these components have evolved.

Tip: Avoid passing references to uninitialized memory since they intimate valid objects to the user.

The definition of `value_policy_adaptor` could hardly be simpler (Listing 28.6).

Listing 28.6 Definition of `value_policy_adaptor`

```
template <typename P>
struct value_policy_adaptor
  : public P
{
public: // Member Types
  typedef typename P::value_type    value_type;
public: // Operations
  struct init_element
  {
    void operator ()(value_type& v) const
    {
      P::init(&v);
    }
  };
  struct copy_element;  // Invokes P::copy()
  struct clear_element; // Invokes P::clear()
};
```

28.5.4 Member Variables

The enumerator sequence maintains only two member variables: the interface pointer passed to the constructor and the retrieval quanta (Listing 28.7).

Listing 28.7 Member Variables

```
. . .
private: // Member Variables
  interface_type* m_root;
  const size_type m_quanta;
private: // Not to be implemented
  enumerator_sequence(class_type const&);
  class_type& operator =(class_type const&);
};
```

Note the proscription of the copy constructor and copy assignment operator: `enumerator_sequence` instances are neither *CopyConstructible* nor *Assignable*. This is partly to avoid complexity in the implementation that would probably never be used, but mainly because the rules of COM enumerators mean that you cannot be guaranteed that cloning an enumerator always yields an instance that will behave identically to its source instance. This behavior has significant consequences for iterator semantics, as we will see.

28.5.5 Construction

Given the member variables, it should be very clear what the constructor and destructor do, as shown in Listing 28.8.

Listing 28.8 Construction Methods

```
public: // Construction
  enumerator_sequence(interface_type* i
                    , bool            bAddRef
                    , size_type       quanta  = 0
                    , bool            bReset  = true)
    : m_root(i)
    , m_quanta(validate_quanta_(quanta))
  {
    COMSTL_MESSAGE_ASSERT("Precondition violation: interface cannot be
NULL!", NULL != i);
    if(bAddRef)
    {
      m_root->AddRef();
    }
    if(bReset)
    {
      m_root->Reset();
    }
    COMSTL_ASSERT(is_valid());
  }
  ~enumerator_sequence() throw()
  {
    COMSTL_ASSERT(is_valid());
    m_root->Release();
  }
. . .
```

The constructor holds a copy of the interface pointer, optionally adding a reference if the `bAddRef` parameter is *`true`*. Note that this parameter is not defaulted: I *never* default such parameters in any reference-counting classes because it leads to reference-counting errors. It's always better to be explicit.

Tip: Make all reference-counting classes require explicit direction from the user as to whether a reference is to be taken (reference is borrowed) or not (reference is consumed) when passed a raw pointer to a reference-counted class (or interface).

By default, the enumerator is reset to its notional start via a call to `IEnum`*`XXXX`*`::Reset()`.

The precondition enforcement in the constructor ensures that the interface pointer is non-*`NULL`*. Thus, `enumerator_sequence` exhibits *externally initialized immutable RAII* (Chapter 11), which means that the destructor implementation does not need to do a runtime check before calling `Release()` on the managed enumerator interface pointer. Note that (absent precondition failure) the constructor will not throw an exception. We'll see why this is important in Section 30.3.4.

The third constructor parameter, `quanta`, specifies how many values the user wants the given instance to retrieve at a time. This can be any value between *`1`* and the quanta value (`Q`) used to specialize the template. If it's *`0`*, the value is set to the maximum available by the parameter validating worker function `validate_quanta_()`, shown in Listing 28.9.

Listing 28.9 Implementation of the `validate_quanta_()` Worker Method

```
private: // Implementation
  static size_type validate_quanta_(size_type quanta)
  {
    COMSTL_MESSAGE_ASSERT("Cannot set a quantum that exceeds the value
specified in the template specialisation", quanta <= retrievalQuanta);
    if( 0 == quanta ||
        quanta > retrievalQuanta)
    {
      quanta = retrievalQuanta;
    }
     return quanta;
  }
```

Note that the `if` statement takes into account a precondition validating value of `quanta`. Thus it appears to correct a precondition violation. There is no meaningful way in which contract violations can be corrected, even those of preconditions, which necessarily predate the actual epoch of process-entering-undefined-behavior. So what is this code doing? Well, in this case, the assertion is not a precondition per se; rather, it is a debug-time mnemonic for users to avoid *DRY SPOT* violations. A user may change the value of `Q` in a specialization of the template without affecting its apparent semantics, merely its performance. Say the user had a `Q` of *`10`* and specified *`8`* to the constructor of an instance of the specialization. If he or she subsequently changed `Q` to be *`5`* but forgot to change the *`8`* in the constructor call, it would then be incorrect and would not represent any meaningful design decision in the use of the sequence. The assertion will help the user track such omissions, although the code will still perform semantically correctly in builds in which the precondition test is elided.

28.5.6 Iteration Methods

COM enumerators do not provide any direct facility for retrieving the total number of elements, or elements by index, so the only two remaining methods of `enumerator_sequence` are thus `begin()` and `end()`, which simply return suitably constructed instances of the `iterator` class, as shown in Listing 28.10. Because the iterator type is not *bidirectional* (or higher) (Section 1.3), the endpoint iterator is simply a default-constructed instance.

Listing 28.10 Enumeration Methods

```
iterator enumerator_sequence<. . .>::begin() const
{
  COMSTL_ASSERT(is_valid());
  return iterator(m_root, m_quanta);
}
iterator enumerator_sequence<. . .>::end() const
{
  COMSTL_ASSERT(is_valid());
  return iterator();
}
```

28.5.7 `const`-Incorrect Iterator Methods?

You may be looking at the collection interface and pondering that the `begin()` and `end()` methods are nonmutating (`const`), and yet they return mutating iterator instances. This contradicts the usual convention where nonmutating `begin()/end()` methods return instances of nonmutating iterator types and, optionally, mutating overloads return instances of mutating iterator types. This inconsistency accounts for the fact that there is a logical and a physical `const`-incorrectness in COM enumerator interface definitions (and in COM as a whole, as it happens). Because COM is language-independent, the C++ `const` qualifier is not used, either for parameters or for member functions. Thus a COM interface method that does not change the state of the object on which it is expressed is still a mutating (non-`const`) method as far as C++ is concerned. Further, the return of resource-bound values, such as `VARIANT`, must be done via mutating (non-`const`) pointer arguments because COM methods must (in almost all cases) return `HRESULT`.

When providing access to an enumerated value, we must facilitate manipulation in just the same manner as is available when manually enumerating. If, for example, our value type is `IFileEntry`, we want to be able to invoke the (non-`const`, yet still "nonmutating") methods on it. This means that any iterator whose value we wish to access—via `operator *()` or via operator ->()—must have mutating (non-const) pointer and reference members. Given that, we are really looking at an iterator, and not a const_iterator: When users are writing client code, they are forced to use the iterator member type to declare iterator variables, rather than const_iterator (which does not exist). This unambiguously indicates to them that they're dealing with something that is physically (even if not always logically) mutating. And given *that*, we might opt to provide begin() and end() as only mutating (non-const) members. Unfortunately, this would mean that the const-incorrectness would propagate out to any function to which you might wish to pass a reference to an instance of the sequence type. Notwithstanding the potential hazards when logical const-ness and physical const-ness are in contradiction, this is where I decided to draw the line. You might draw it otherwise. (Providing both mutating *and* nonmutating iterator methods, all returning iterator instances, would just be a pointless waste of key presses.)

28.5.8 Breaking Value Semantics?

One important feature of enumerator_sequence is that its value types may not necessarily have value semantics. This is because many COM types do not have value semantics, the most obvious ones being BSTR and VARIANT. The only way to make the sequence follow this rule would be to anticipate *all* the enumerator interfaces (and their value types) with which it might ever be specialized and to stipulate the sequence value type for each one; for example, the value type for IEnumVARIANT would be comstl::variant, that for IEnumBSTR would be comstl::bstr, and so on. Naturally, this is impossible, so the best that can be done is to provide the opportunity to change the reference type (which defaults to V const&) to one with value semantics. For example, imagine that we were using enumerator_sequence with the IEnumBSTR interface. The specialization would be as follows:

```
typedef comstl::enumerator_sequence<IEnumBSTR
                                  , BSTR
                                  , comstl::BSTR_policy
                                  >     enum_t;
```

We can make it follow the value type rule by specifying a type with value semantics, for example, comstl::bstr, as the fourth parameter:

```
typedef comstl::enumerator_sequence<IEnumBSTR
                                  , BSTR
                                  , mystuff::BSTR_to_BStr_policy
                                  , comstl::bstr
                                  >     enum_t;
```

Now we can do something like the following and have it well defined and leak-free:

```
comstl::bstr  str = *enum_t(pEnum, true).begin();
```

28.6 `comstl::enumerator_sequence::iterator`

Given the succinct definition of `enumerator_sequence`, it's obvious that the bulk of the work is to be done in the nested `iterator` class. Its declaration is shown in Listing 28.11. The most obvious part of this declaration is the use of yet another nested type, `enumeration_context`, to which `iterator` defers some of its behavior. Before we investigate how and why this is needed, let's look at `iterator`'s methods.

Listing 28.11 Definition of `enumerator_sequence::iterator`

```
template <. . .>
class enumerator_sequence::iterator
  : std::iterator<iterator_tag_type
                 , value_type, ptrdiff_t
                 , pointer, reference
                 >
{
public: // Member Types
  typedef iterator  class_type;
  . . .
private:
  struct enumeration_context
  {
  public: // Member Types
    typedef enumeration_context   class_type;
    . . .
  private: // Construction
    enumeration_context(interface_type* i, class_type const& rhs);
  public:
    enumeration_context(interface_type* i, size_type quanta);
    ~enumeration_context() throw();
    void AddRef();
    void Release();
    static class_type*  make_clone(class_type* ctxt);
  public: // Iteration
    void        advance() throw();
    value_type& current() throw();
    size_type   index() const throw();
    bool        empty() const throw();
  public: // Invariant
    bool        is_valid() const;
    . . .
  }; // End of enumeration_context
private: // Construction
  friend class enumerator_sequence<I, V, VP, R, CP, Q>;
  iterator(interface_type* i, size_type quanta);
public:
  iterator();
  iterator(class_type const& rhs);
  class_type& operator =(class_type const& rhs);
public: // Forward Iterator Methods
  class_type& operator ++();
  class_type  operator ++(int);
  reference   operator *();
  pointer     operator ->();
  bool        equal(class_type const& rhs) const;
private: // Invariant
  bool is_valid() const;
private: // Implementation
  bool is_end_point() const;
  static bool equal_( class_type const& lhs
                    , class_type const& rhs
                    , std::input_iterator_tag);
  static bool equal_( class_type const& lhs
                    , class_type const& rhs
                    , std::forward_iterator_tag);
private: // Member Variables
  enumeration_context*  m_ctxt;
};
```

28.6.1 Construction

The `iterator` construction methods (Listing 28.12) are all reasonably straightforward. The conversion constructor creates an `enumeration_context` instance. The default constructor, which yields endpoint instances, has a *NULL* context. This tallies with what we know regarding the lack of need for logical state in input and forward iterator endpoint instances. The copy constructor and copy assignment operators get a clone of the enumeration context held by the copied instance, via the mysterious `enumeration_context::make_clone()` method. The destructor releases the context.

Listing 28.12 Construction Methods

```
private: // Construction
  iterator(interface_type* i, size_type quanta)
    : m_ctxt(new enumeration_context(i, quanta))
  {
    COMSTL_ASSERT(is_valid());
  }
public:
  iterator()
    : m_ctxt(NULL)
  {
    COMSTL_ASSERT(is_valid());
  }
  iterator(class_type const& rhs)
    : m_ctxt(enumeration_context::make_clone(rhs.m_ctxt))
  {
    COMSTL_ASSERT(is_valid());
  }
  ~iterator() throw()
  {
    COMSTL_ASSERT(is_valid());
    if(NULL != m_ctxt)
    {
      m_ctxt->Release();
    }
  }
  class_type& operator =(class_type const& rhs)
  {
    enumeration_context*
                  newCtxt = enumeration_context::make_clone(rhs.m_ctxt);
    if(NULL != m_ctxt)
    {
      m_ctxt->Release();
    }
    m_ctxt = newCtxt;
    return *this;
  }
  . . .
```

The `iterator` class uses reference counting to control the lifetime of its `m_ctxt` instance. Note that in the copy assignment operator `m_ctxt` is released *after* the call to `make_clone()`, since `make_clone()` might throw an exception.

28.6.2 Iteration Methods

The iteration methods are all simple delegations to `enumeration_context`, as shown in Listing 28.13. Again, there's little to remark on; `iterator` clearly abrogates almost all responsibility to the nested class.

Listing 28.13 Increment, Dereference, and Member Selection Operators

```
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    COMSTL_ASSERT(is_valid());
    m_ctxt->advance();
    COMSTL_ASSERT(is_valid());
    return *this;
  }
  class_type operator ++(int); // Canonical implementation
  reference operator *()
  {
    COMSTL_ASSERT(is_valid());
    COMSTL_MESSAGE_ASSERT("Attempting to dereference an invalid
iterator", NULL != m_ctxt && !m_ctxt->empty());
```

```
    return m_ctxt->current();
  }
  pointer operator ->()
  {
    COMSTL_ASSERT(is_valid());
    COMSTL_MESSAGE_ASSERT("Attempting to dereference an invalid
iterator", NULL != m_ctxt && !m_ctxt->empty());
    return &m_ctxt->current();
  }
  . . .
```

28.6.3 equal()

Since I mentioned in the introduction that `enumerator_sequence` can support both input and forward iterators, you may have wondered how iterator comparison can be supported for both types. The answer lies in the compile-time selection of the appropriate overload of the worker method `equal_()`, based on the iterator category, as shown in Listing 28.14. (I've left the comments in to assist you in understanding the strategy.)

Listing 28.14 Members of `enumerator_sequence` **for Evaluating Equality**

```
private: // Implementation
  bool is_end_point() const
  {
    return NULL == m_ctxt || m_ctxt->empty();
  }
  static bool equal_( class_type const& lhs
                    , class_type const& rhs
                    , std::input_iterator_tag)
  {
    // Only equal if both at endpoint.
    return lhs.is_end_point() && rhs.is_end_point();
  }
  static bool equal_( class_type const& lhs
                     , class_type const& rhs
                     , std::forward_iterator_tag)
  {
    // The iterators are equal under two conditions:
    // 1. Both are at end (as in case for input iterators)
    // 2. Both have context, and index of both contexts are same
    // Otherwise:
    // 3. They're not equal
    if(lhs.is_end_point())
    {
      return rhs.is_end_point(); // 1 or 3
    }
    else if(rhs.is_end_point())
    {
      return false; // 3
    }
    else
    {
      COMSTL_ASSERT(NULL != lhs.m_ctxt);
      COMSTL_ASSERT(NULL != rhs.m_ctxt);
      return lhs.m_ctxt->index() == rhs.m_ctxt->index(); // 2 or 3
    }
  }
public: // Comparison
  bool equal(class_type const& rhs) const
  {
    COMSTL_ASSERT(is_valid());
    return class_type::equal_(*this, rhs, iterator_tag_type());
  }
  . . .
```

The `is_end_point()` method is used to identify whether an iterator instance is at the endpoint. This can be when there is no context or when the context is empty: They're logically equivalent. When the iterator category is input iterator, the first overload of the `equal_()` worker method will be selected within the `equal()` method. In this case, the iterators can be equal only if they're both endpoint iterators. When the iterator category is forward iterator, the second `equal_()` overload is selected, in which case equality also incorporates the comparison of the current iteration position, as indicated by the `index()` method of `enumeration_context`, between two non-endpoint instances.

28.7 `comstl::enumerator_sequence::iterator::enumeration_context`

As you may have guessed, the nested class `enumerator_sequence::iterator::enumeration_context` is a reference-counted cache of values representing a particular enumeration point. It serves equally whatever iterator category is exhibited by a given specialization of the sequence; it is shared by input iterator instances and exclusively owned by forward iterator instances. It initializes, acquires, and releases elements in batches, according to the quanta value of the sequence class. It maintains state information about the iterator position, in the form of a count of the enumerated items from the logical start of the enumeration, to support comparison of forward iterator instances.

28.7.1 Why an Enumeration Context?

Before we get into the how, it behooves me to explain why an enumeration context is needed. The original version of the sequence did not have it, and all the functionality of iteration was maintained within the `iterator` class itself. But this was quite inefficient when the specialization supported input iteration: Each time an instance of an iterator was copied, all the contents of the source instance had to be copied into the receiving instance. And that's just half the sorry tale. Since input iterators must support only single-pass algorithms, the source instance must then be put into a state whereby it cannot purport to be the sole representative of the iteration. The standard

(C++-03: 24.1.1;3) states that "algorithms on input iterators should never attempt to pass through" (that's increment to you and me) the same iterator twice.

This raises the question of whether a source instance should be dereferenceable after an iterator copy operation is completed. The standard states that one is a copy of the other. It also says that after a subsequent increment, other copies are not required to be dereferenceable. We can infer that two copies of an input iterator, where one is copied from the other, may be dereferenceable until such time as one of them is incremented. Throw into the mix the fact that each iterator instance can contain a cache of `Q` elements, representing the current value and a number of subsequent values, and you can imagine the logical conundrum in trying to get meaningful semantics when each instance has one or more cached enumerated values to play with. Without shared state, it's almost impossible to achieve meaningful behavior when copying input iterators. And I quickly reached the point where I didn't want to continue to try. (An illustration of the problem is provided in an extra section for this chapter on the CD.)

So, `enumeration_context` was born. With one notable exception, which we address at the end of this chapter, using `enumeration_context` solves all such problems and allows the design of `enumerator_sequence` to be relative simple and straightforward.

28.7.2 Class Definition

The remainder of the class definition of `enumeration_context` is shown in Listing 28.15, adding in those elements not shown earlier.

Listing 28.15 Definition of `enumerator_context`

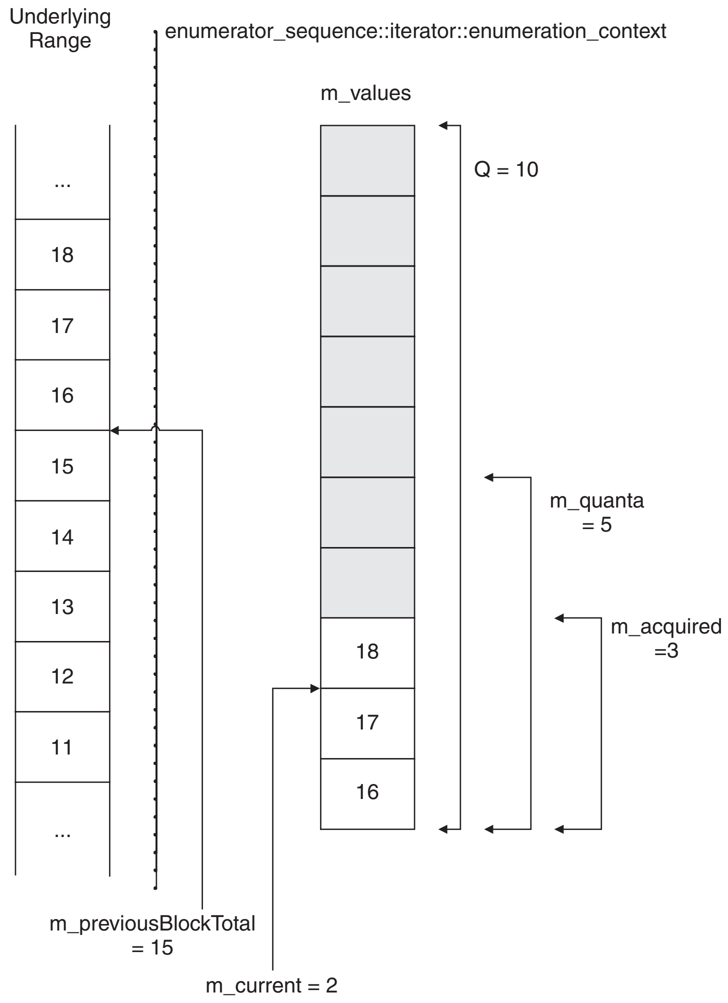
```
struct enumerator_sequence<. . .>::iterator::enumeration_context
{
. . . // Member Types, Construction, Invariant, and Iteration as earlier
private: // Implementation
  void acquire_next_() throw();
  void clear_elements_() throw();
  void init_elements_(size_type n) throw();
private: // Member Variables
  interface_type* m_enumerator;
  size_type       m_acquired;
  size_type       m_current;
  size_type const m_quanta;
  value_type      m_values[retrievalQuanta];
  long            m_refCount;
  size_type       m_previousBlockTotal;
private: // Not to be implemented
  enumeration_context(class_type const&);
  class_type& operator =(class_type const&);
};
```

There are seven member variables. `m_enumerator` is a pointer to the underlying enumerator instance, to which `enumeration_context` holds a reference. `m_refCount` is the reference count of the `enumeration_context` instance (controlled via its `AddRef()` and

`Release()` methods). The five remaining members, `m_values`, `m_quanta`, `m_acquired`, `m_current`, and `m_previousBlockTotal`, represent the cached enumerated values and the enumeration state. Figure 28.1 shows the relationship of these members, in respect to a notional enumerated range.

In the state shown in the figure, the user specified a `Q` of *`10`* in the specialization of the sequence class, so `m_values` is an array of ten elements. The constructor received a value of *5* for `quanta`; hence `m_quanta` is *5*. The `enumeration_context` instance has been retrieving elements in batches of five from `IEnum`*`XXXX`*`::Next()`. Three such retrievals have been made previously; hence `m_previousBlockTotal` is *`15`*. The fourth retrieval yielded only three elements; hence `m_acquired` is *3*. Seventeen elements have been iterated, via the `advance()` method (called by `iterator::operator ++()`); hence `m_current` is *2*.

Figure 28.1 Iterator member variables and enumeration state



28.7.3 Construction

There are two constructors for enumeration_context, a copying constructor and a sharing constructor. The copying constructor, shown in Listing 28.16, takes an enumerator interface pointer and a source instance whose values and enumeration state will be copied. It copies the contents of all currently cached values and also copies the m_acquired, m_current, and m_previousBlockTotal members, which means it adopts the identical logical enumeration point as the source instance. This constructor is used to create copies of forward iterator instances.

Listing 28.16 enumeration_context **Copying Constructor**

```
enumeration_context(interface_type* i, class_type const& rhs)
  : m_enumerator(i)
  , m_acquired(rhs.m_acquired)
  , m_current(rhs.m_current)
  , m_quanta(rhs.m_quanta)
  , m_refCount(1)
  , m_previousBlockTotal(rhs.m_previousBlockTotal)
{
  COMSTL_ASSERT(rhs.m_acquired <= m_quanta);
  init_elements_(m_quanta);
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
  try
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
  {
    value_type*       begin     = &m_values[0];
    value_type*       end       = &m_values[0] + m_quanta;
    value_type const* src_begin = &rhs.m_values[0];
    value_type const* src_end   = &rhs.m_values[0] + rhs.m_acquired;
    for(; src_begin != src_end; ++begin, ++src_begin)
    {
      value_policy_type::copy(begin, src_begin);
    }
    COMSTL_ASSERT(begin <= end);
  }
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
  catch(...)
  {
    clear_elements_();
    throw;
  }
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
  COMSTL_ASSERT(is_valid());
 COMSTL_ASSERT(this->index() == rhs.index());
}
```

The sharing constructor starts a new enumeration from the current position of an enumerator (Listing 28.17). It is called by the conversion constructor of the iterator class, itself called from the `begin()` method of the sequence. It sets all its elements to their nominal default values and then invokes the private worker method `acquire_next_()` (described in Section 28.7.4).

Listing 28.17 `enumeration_context` **Sharing Constructor**

```
enumeration_context(interface_type* i, size_type quanta)
  : m_enumerator(cloning_policy_type::share(i))
  , m_acquired(0)
  , m_current(0)
  , m_quanta(quanta)
  , m_refCount(1)
  , m_previousBlockTotal(0)
{
  COMSTL_ASSERT(quanta <= STLSOFT_NUM_ELEMENTS(m_values));
  init_elements_(m_quanta);
  acquire_next_();
  COMSTL_ASSERT(is_valid());
}
```

The destructor releases any resources, via a call to `clear_elements_()`, and then releases the enumerator (Listing 28.18). Note the local increment and decrement around the invariant test, in order to avoid precipitating an inappropriate detection of an erroneous reference count.

Listing 28.18 Destructor and Lifetime Control Methods of `enumeration_context`

```
~enumeration_context()
{
  COMSTL_ASSERT(0 == m_refCount);
  ++m_refCount; // Bump so is_valid() is valid
  COMSTL_ASSERT(is_valid());
  --m_refCount; // Drop after is_valid()
  clear_elements_();
  if(NULL == m_enumerator)
  {
    m_enumerator->Release();
  }
}
void AddRef()
{
  ++m_refCount;
}
void Release()
{
  if(0 == --m_refCount)
  {
    delete this;
  }
}
```

The make_clone() method has a degree of complexity reflecting its duty of balancing the assumptions about an enumerator's capabilities specified at compile time with the actuality of its capabilities as detected at runtime. I'll defer a discussion of the wider ramifications of that balance until Section 28.8 and just explain what make_clone() does. Its implementation is shown in Listing 28.19.

Listing 28.19 make_clone() **Method of** enumeration_context

```
static class_type* make_clone(class_type* ctxt)
{
  if(NULL == ctxt)
  {
    return NULL;
  }
  else
  {
    COMSTL_ASSERT(NULL != ctxt->m_enumerator);
    interface_type* copy;
    if(!cloning_policy_type::clone(ctxt->m_enumerator, &copy))
    {
      COMSTL_ASSERT(NULL == copy);
      ctxt->AddRef();
      return ctxt;
    }
    else
    {
      COMSTL_ASSERT(NULL != copy);
      class_type* newCtxt;
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
      try
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
      {
        newCtxt = new class_type(copy, *ctxt);
        if(NULL == newCtxt)
        {
          copy->Release();
        }
      }
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
      catch(...)
      {
        copy->Release();
        throw;
      }
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
      return newCtxt;
    }
  }
}
```

First, if the source context pointer is *NULL*, then *NULL* is returned. This affords us simplicity in iterator's copy assignment operator without requiring special-case testing of empty instances. If the source context pointer is non-*NULL*, we invoke the cloning policy's clone() method on it to retrieve a copy. The precise functioning of this method is, obviously, policy-dependent, but its role can be summarized thus.

- If the policy stipulates support for cloning (via IEnum*XXXX*::Clone()), then clone and return *true*, or throw an exception (clone_failure) if cloning fails.
- If the policy does not stipulate support for cloning, then return *false*. In this case, make_clone() simply adds a reference to the source enumeration_context instance and returns it.

In the first case, a successful clone yields a new enumerator, so the copying constructor is invoked and the new enumeration_context instance, which now owns the cloned enumerator, is returned. The explicit try-catch is to ensure that the enumerator instance is released if the enumeration_context construction fails. And since this component must be compilable absent exception support, there's also a test for *NULL* on the new operator return. (It's not unusual for COM components to be built without exception-handling support. As pioneered by **ATL**, COM components can be drastically reduced in size when the runtime library, which includes support for exception throwing and handling, is avoided.)

28.7.4 Iterator Support Methods

The implementations of the iterator support methods are pretty straightforward, as shown in Listing 28.20. The advance() method does a whole lot of sanity checking via precondition enforcements. These are telling vestiges of the challenging evolution of the component, which could be viewed as an embarrassing history of the missteps. However, it's far better to keep the glass half full about such things and instead view them as important checks against any retrogradation of the implementation. The functionality of advance() is simple: Bump m_current while there are more cached elements; otherwise, clear the current elements and invoke acquire_next_() to get the next batch.

Listing 28.20 Iterator Support Methods

```
void advance() throw()
{
  COMSTL_ASSERT(NULL != m_enumerator);
  COMSTL_ASSERT(0 < m_refCount);
  COMSTL_ASSERT(0 != m_acquired);
  COMSTL_ASSERT(m_current < m_acquired);
  COMSTL_ASSERT(m_acquired <= m_quanta);
  COMSTL_ASSERT(m_quanta <= STLSOFT_NUM_ELEMENTS(m_values));
  if(++m_current < m_acquired)
  {
    // Nothing to do; iteration pt refers to next entry in m_values
  }
  else
```

```
      {
        COMSTL_ASSERT(NULL != m_enumerator);
        clear_elements_();
        m_current = 0;
        acquire_next_();
      }
    }
    value_type& current() throw()
    {
      COMSTL_ASSERT(!empty());
      return m_values[m_current];
    }
    size_type index() const throw()
    {
      return m_previousBlockTotal + m_current;
    }
    bool empty() const throw()
    {
      return 0 == m_acquired;
    }
  private: // Implementation
    void acquire_next_() throw()
    {
      COMSTL_ASSERT(0 == m_current);
      ULONG   cFetched = 0;
      m_enumerator->Next(m_quanta, &m_values[0], &cFetched);
      m_acquired            =   cFetched;
      m_previousBlockTotal  +=  cFetched;
    }
```

current() returns a reference to the current item; the instance must not be empty. empty() is self-explanatory. And index() returns the index of the enumeration point, relative to the start of the enumeration. Because m_previousBlockTotal is copied in the copying constructor (Section 28.7.3), this is meaningful between cloned contexts (i.e., between copied forward iterators) and thereby supports the iterator equality comparison.

28.7.5 Invariant

With logic as complex as this, it's very important to have a well-defined notion of the class invariant and for that notion to be expressed in code that is frequently verified. The is_valid() method reflects this, as shown in Listing 28.21. (For brevity I have elided all but one of the output statements that are compiled when building for **STLSoft**'s unit-testing framework. You can see the full implementation in the **STLSoft** libraries, included on the CD.)

Listing 28.21 `enumeration_context` **Class Invariant**

```
bool is_valid() const
{
  if(m_refCount < 1)
  {
#ifdef STLSOFT_UNITTEST
    unittest::fprintf(unittest::err, "invalid ref count (%ld) \n"
                    , m_refCount);
#endif /* STLSOFT_UNITTEST */
    return false;
  }
  if( NULL == m_enumerator &&
      0 == m_quanta)
  {
    if(0 != m_acquired)
    {
      . . . // Report and return false
    }
    if(0 != m_current)
    {
      . . . // Report and return false
    }
    if(0 != m_quanta)
    {
      . . . // Report and return false
    }
  }
  else
  {
    if(m_acquired < m_current)
    {
      . . . // Report and return false
    }
    if(m_quanta < m_current)
    {
      . . . // Report and return false
    }
    if(m_quanta < m_acquired)
    {
      . . . // Report and return false
    }
  }
  return true;
}
```

28.8 Iterator Cloning Policies

Now it's time to discuss the cloning policies. Before we go any further, I must come clean and admit that I'm holding onto a *final* version until Section 28.9. What I've presented thus far, and will continue to present in this section, represents a correct and workable solution, but it has one suboptimal behavioral characteristic. Before addressing that issue, we need to cover cloning policies since they're pivotal to understanding the final solution. The cloning policy conforms to the interface shown in Listing 28.22.

Listing 28.22 Cloning Policy Interface

```
template<typename I>
struct XYZ_cloning_policy
{
public: // Member Types
  typedef I                       interface_type;
  typedef std::input_iterator_tag iterator_tag_type;
public: // Operations
  . . .
  static interface_type* share(interface_type* src);
  static bool clone(interface_type* src, interface_type** pdest);
};
```

The policy has two responsibilities. First, it must define the iterator tag that will be used to specialize `std::iterator` for the `iterator` class. Second, it must define the `share()` and `clone()` methods that define the runtime behavior of the policy.

The `share()` method must return a suitable copy of the given interface; the definition of "suitable" is policy-dependent. It is used in the sharing constructor (refer back to Listing 28.17) of `enumeration_context`.

The `clone()` method attempts to create a genuine clone of the enumerator represented by `src` and to store it in `*pdest`. It is used in the `make_clone()` method of `enumeration_context`. It must return *`true`* if it succeeds in creating a clone, or *`false`* if the policy does not create clones; it must throw an instance of `clone_failure` if the policy invokes `IEnum`*`XXXX`*`::Clone()` on `src` and fails.

There are three cloning policies provided with **COMSTL**: `input_cloning_policy`, `forward_cloning_policy`, and `cloneable_cloning_policy`, which vary in the type of `iterator_tag_type` and the behavior of `share()` and `clone()`. Listing 28.3 earlier in the chapter showed `input_cloning_policy` as the one used by default.

28.8.1 `comstl::input_cloning_policy`

This policy assumes that the enumerator cannot support forward iterator semantics, and the sequence's iterators should exhibit input iterator semantics. That is to say, it assumes neither that the enumerator can be cloned nor that its clone is repeatable. We know COM does not require either of these of an arbitrary enumerator, which is why `input_cloning_policy` is the chosen default. Its definition is shown in Listing 28.23.

Listing 28.23 Definition of `input_cloning_policy`

```
// In namespace comstl
template<typename I>
struct input_cloning_policy
{
public: // Member Types
  typedef I                          interface_type;
  typedef std::input_iterator_tag   iterator_tag_type;
public: // Operations
  . . .
  static interface_type* share(interface_type* src)
  {
    COMSTL_ASSERT(NULL != src);
    src->AddRef();
    return src;
  }
  static bool clone(interface_type* src, interface_type** pdest)
  {
    COMSTL_ASSERT(NULL != src);
    COMSTL_ASSERT(NULL != pdest);
    *pdest = NULL;
    return false;
  }
};
```

The policy defines the iterator tag as `std::input_iterator_tag`. Algorithms will not (and users should not) assume that the specialization provides multipass iterators. The `share()` method calls `AddRef()` on the given instance, which means that all iterators will share the same enumerator. The `clone()` method simply sets the returned pointer to *`NULL`* and returns *`false`*, causing the `make_clone()` method to return a reference to the "copied" `enumeration_context` instance, thereby sharing the underlying state, as is appropriate for input iterators. Note that using `input_cloning_policy` means that a sequence can provide only *one* iterable range, regardless of whether the iterators are derived from one or several different calls to `begin()`, as illustrated in the following code:

```
typedef enumerator_sequence<IEnumXYZ
                           , input_cloning_policy<. . .>
                           , . . .
                           >  es_t;
IEnumXYZ*       xyz = . . .
es_t            es(xyz,  . . . );
es_t::iterator  b1 = es.begin();
es_t::iterator  e1 = es.end();

for(; b1 != e1; ++b1)
{}
```

```
es_t::iterator  b2 = es.begin();
es_t::iterator  e2 = es.end();

assert(b2 == e2); // Enumeration of b1 always exhausts sequence
```

Indeed, iterators derived from separate `enumerator_sequence` instances may still cover only one overarching range, as illustrated in the following code:

```
IEnumXYZ*       xyz = . . .
es_t            es1(xyz,  . . . );
es_t::iterator  b1 = es1.begin();
es_t::iterator  e1 = es1.end();

for(; b1 != e1; ++b1)
{}

es_t            es2(xyz,  . . . );
es_t::iterator  b2 = es2.begin();
es_t::iterator  e2 = es2.end();

assert(b2 == e2); // Enumeration of b1 always exhausts all sequences
```

As long as users understand this characteristic of specializations involving `input_cloning_policy`, that's an acceptable semantic. However, it is something that many users would probably be surprised by; we'll come back and address this in the final implementation to follow.

28.8.2 `comstl::forward_cloning_policy`

The second policy, `forward_cloning_policy`, allows users to specify that they *know* that a given enumerator supports forward iterator semantics. As shown in Listing 28.24, it defines the `iterator_tag_type` member type as `std::forward_iterator_tag` and implements the `share()` and `clone()` methods accordingly.

Listing 28.24 Definition of `forward_cloning_policy`

```
// In namespace comstl
template<typename I>
struct forward_cloning_policy
{
public: // Member Types
  typedef I                          interface_type;
  typedef std::forward_iterator_tag  iterator_tag_type;
public: // Operations
  . . .
  static interface_type* share(interface_type const* src)
  {
    COMSTL_ASSERT(NULL != src);
    interface_type* ret;
    HRESULT         hr = const_cast<interface_type*>(src)->Clone(&ret);
    if(FAILED(hr))
    {
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
      throw clone_failure(hr);
#else /* ? STLSOFT_CF_EXCEPTION_SUPPORT */
      ret = NULL;
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
    }
    return ret;
  }
  static bool clone(interface_type* src, interface_type** pdest)
  {
    COMSTL_ASSERT(NULL != src);
    COMSTL_ASSERT(NULL != pdest);
    *pdest = share(src);
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
    COMSTL_ASSERT(NULL != *pdest);
    return true;
#else /* ? STLSOFT_CF_EXCEPTION_SUPPORT */
    return NULL != *pdest;
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
  }
};
```

`share()` is where the action happens. The `IEnum`*`XXXX`*`::Clone()` member is invoked. If successful, the new enumerator instance is returned. If it fails, an instance of the exception class `comstl::clone_failure` is thrown or, if compiled absent exception support, *`NULL`* is returned. The second overload is merely implemented in terms of the first. It can only return *`false`* if compiled absent exception support and the `Clone()` method fails, in which case the iterators will behave like input iterators. (Obviously, in some cases this will affect the semantics of the client code. This is a case of caveat emptor. If you want to use the sequence absent exception support, them's yer apples.)

28.8.3 `comstl::cloneable_cloning_policy`

There is a halfway house between the input and forward cloning policies: the `cloneable_cloning_policy`, shown in Listing 28.25. (Yes, it's another graduate from the Wilson Academy of Disobfuscatory Nomenclatural Disambiguation.) It exhibits the `input_iterator_tag` à la `input_cloning_policy` but has exactly the same functionality as the `forward_cloning_policy`. The user selects this when he or she knows that the enumerator supports cloning but does not necessarily provide a repeatable range.

Listing 28.25 Definition of `cloneable_cloning_policy`

```
// In namespace comstl
template<typename I>
struct cloneable_cloning_policy
{
public: // Member Types
  typedef I                          interface_type;
  typedef std::input_iterator_tag    iterator_tag_type;
public: // Operations
  . . .
  static interface_type* share(interface_type const* src)
  {
    . . . // Same as forward_cloning_policy<I>::share()
  }
  static bool clone(interface_type* src, interface_type** pdest)
  {
    . . . // Same as forward_cloning_policy<I>::clone()
  }
};
```

28.9 Choosing a Default Cloning Policy: Applying the *Principle of Least Surprise*

The *Principle of Least Surprise* is one of the most important principles when implementing discoverable libraries. Applying it in this case, however, presents an interesting challenge. We have three policies to choose from at compile time, and we need to match these to enumerators whose adherence to a chosen policy may not be determinable until runtime. This is a serious disconnect.

Because we must attempt to ameliorate the ramifications of *Henney's Hypothesis* (Chapter 14), we need to default as many template parameters as is sensible, including the cloning policy (`CP`). But which to pick? It's fair to say that we can discount from consideration the `forward_iterator_policy` because it assumes the most exacting, and hardest to verify, behavior of an enumerator. But how do we choose between the other two? Using the `input_cloning_policy` essentially means that we can have at most one concurrent pass of the enumerator, irrespective of the apparent independence of invocations of `begin()` on a given sequence or of separate sequence instances. As I commented earlier, this is somewhat surprising.

Alternatively, using the `cloneable_cloning_policy` means that an enumerator must be cloneable. If `IEnum`*XXXX*`::Clone()` fails, construction of the iterator to be returned by `begin()` will fail, even for the first time. You would pass a viable but noncloneable enumerator instance to the sequence constructor, and you would receive an exception for your troubles, even though you wanted to do only a single pass! This too would be surprising.

With `input_cloning_policy`, one option is to monitor how many iterators are alive and invoke `Reset()` on a call to `enumerator_sequence::begin()` when there are no extant iterator instances. But this is not viable because it's possible that another sequence instance is being used on the same enumerator interface pointer. Whichever way you cut it, this abstraction leaks big style!

The approach I've chosen is a workable, and least-surprising, compromise, albeit one involving a little hack. The default policy is changed to `cloneable_cloning_policy`, but a clone is retrieved only on second and subsequent invocations of `begin()`. If the second (or subsequent) invocation fails with a `clone_failure` exception, that's in keeping with reasonable expectations a user might have of an enumerator, rather than being a surprise. The modifications required to effect this changed behavior are relatively simple, though they do involve all the classes. Listing 28.26 shows the changes.

Listing 28.26 Required Changes to `enumerator_sequence`

```
template< typename I                            // Enumerator interface
        , typename V                            // Value type
        , typename VP                           // Value policy type
        , typename R   = V const&               // Reference type
        , typename CP  = cloneable_cloning_policy<I> // Cloning policy
        , size_t   Q   = 10                     // Quanta
        >
class enumerator_sequence
{
  . . .
public: // Construction
  enumerator_sequence(interface_type* i
                    , bool            bAddRef
                    , size_type       quanta = 0)
    : m_root(i)
    , m_enumerator(NULL)
    , m_quanta(validate_quanta_(quanta))
    , m_bFirst(true)
  {
    . . . // As original version
    if(bReset)
    {
      m_root->Reset();
    }
    m_enumerator = cloning_policy_type::get_working_instance(m_root);
    if(NULL != m_enumerator)
    {
      m_bFirst = false;
    }
    COMSTL_ASSERT(is_valid());
  }
  . . .
public: // Iteration
  iterator begin() const
  {
    COMSTL_ASSERT(is_valid());
    interface_type* en;
    if(NULL != m_enumerator)
    {
      en = m_enumerator;
    }
    else
    {
      if(!m_bFirst)
      {
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
        throw clone_failure(E_NOTIMPL);
#else /* ? STLSOFT_CF_EXCEPTION_SUPPORT */
        return end();
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
      }
      en = m_root;
    }
    COMSTL_ASSERT(NULL != en);
    return iterator(m_root, m_quanta, m_bFirst);
  }
  . . .
private: // Member Variables
  interface_type*   m_root;
  interface_type*   m_enumerator;
  size_type const   m_quanta;
  mutable   bool    m_bFirst;
  . . .
};
```

The first change is that the default cloning policy is now `cloneable_cloning_policy`. This reflects the fact that many, if not most, enumerators are cloneable. The remainder of the changes to all three classes are concerned with making the first iteration viable even in the case where the enumerator is not actually cloneable. Two new members are added to the sequence class, `m_enumerator` and `m_bFirst`. Together these two can inform the implementation of `begin()` whether it is being called the first time and, if not, whether the enumerator is cloneable. If it's not, subsequent invocations of `begin()` will result in the `clone_failure` exception being thrown. The one remaining significant change is the invocation of the new `get_working_instance()` method on the cloning policy in the sequence constructor. Essentially, the function of this method is to have a first go at `share()`, but with execution remaining with the caller on failure-to-clone as appropriate to the cloning policy.

The necessary changes to `iterator::iterator()` are shown in Listing 28.27. `bFirst` is reset in the constructor body because the construction of `enumeration_context` can throw an exception, in which case we don't want to modify `bFirst`.

Listing 28.27 Required Changes to the `iterator` **Constructor**

```
private: // Construction
  iterator(interface_type* i, size_type quanta, bool& bFirst)
    : m_ctxt(new enumeration_context(i, quanta, bFirst))
  {
    bFirst = false;
    COMSTL_ASSERT(is_valid());
  }
  . . .
```

The change to the `enumeration_context` class is shown in Listing 28.28. `AddRef()` is used instead of `share()` when `bFirst` is *`true`*. (The bracing prevents the compiler from becoming confused by the sequence iterator into thinking we're presenting multiple arguments to the constructor of a pointer. That's a set of error messages you can live without, let me tell you!)

Listing 28.28 Required Changes to the `enumeration_context` **Constructor**

```
enumeration_context(interface_type* i, size_type quanta)

  enumeration_context(interface_type* i, size_type quanta, bool bFirst)
    : m_enumerator(bFirst ? (i->AddRef(), i)
                          : cloning_policy_type::share(i))
    , m_acquired(0)
    , m_current(0)
    , m_quanta(quanta)
    , m_refCount(1)
    , m_previousBlockTotal(0)
  {
    . . .
```

The change to the policies is the addition of the `get_working_instance()` method, as shown in Listing 28.29. For `input_cloning_policy`, this is identical to `share()` in that a reference is taken and the same instance returned.

Listing 28.29 Required Changes to Policies

```
template<typename I>
struct input_cloning_policy
{
  . . .
  static interface_type* get_working_instance(interface_type* root)
  {
    COMSTL_ASSERT(NULL != root);
    root->AddRef();
    return root;
  }
  . . .
};
```

```
template<typename I>
struct forward_cloning_policy
{
  . . .
  static interface_type* get_working_instance(interface_type* root)
  {
    COMSTL_ASSERT(NULL != root);
    interface_type* ret;
    HRESULT         hr = const_cast<interface_type*>(root)->Clone(&ret);
    if(FAILED(hr))
    {
#ifdef STLSOFT_CF_EXCEPTION_SUPPORT
      throw clone_failure(hr);
#else /* ? STLSOFT_CF_EXCEPTION_SUPPORT */
      ret = NULL;
#endif /* STLSOFT_CF_EXCEPTION_SUPPORT */
    }
    return ret;
  }
  . . .
};
template<typename I>
struct cloneable_cloning_policy
{
  . . .
  static interface_type* get_working_instance(interface_type* root)
  {
    COMSTL_ASSERT(NULL != root);
    interface_type* ret;
    HRESULT         hr = const_cast<interface_type*>(root)->Clone(&ret);
    if(FAILED(hr))
    {
      ret = NULL;
    }
    return ret;
  }
  . . .
};
```

Similarly, `forward_cloning_policy::get_working_instance()` is identical to `forward_cloning_policy::share()`, so failure to clone is detected in the constructor of `enumerator_sequence`. This is a deliberate design decision. Users who specify `forward_cloning_policy` are making a bold and definite statement about the capabilities of their enumerator and should receive the earliest possible notification if they are wrong.

It's in the definition of `cloneable_cloning_policy` that the magic clicks in. `get_working_instance()` attempts to get the clone, but if that fails, the function returns *NULL*, so `enumerator_sequence::m_enumerator` will be assigned *NULL*. This means that the first invocation of `begin()` will succeed, but subsequent invocations will fail with `clone_failure`. We have, at last, achieved least surprise. *QED*.

I grant that these changes feel a little like a hack, perhaps because they were added after the main structure evolved. But they actually bring a refreshing clarity to the functionality of the default specializations of the sequence; I've never had a problem using this final form, and no users have complained, so I'm pretty happy with it. To be sure, we might have arrived at a different, and possibly more direct, form had all the idiosyncrasies of the COM enumerator and STL iterator concepts been clear and to the fore at the time of the original design. But that's rarely the way with software, especially long-lived libraries.

28.9.1 `empty()`

One last note: Now we can see why there's no `empty()` method provided in `enumerator_sequence`. This would potentially steal the one and only good interface for an enumerator that is not cloneable when `cloneable_cloning_policy` is used. Perverse? Indeed. But sensible? Most assuredly.

28.10 Summary

I've no more rabbits to extract from my hat or doves from my sleeves. That really is the final implementation. If you've had a bit of a slog over the last few sections, I ask you to have a quick skim back to the motivating example at the start of the chapter. `enumerator_sequence` solves *all* the problems commonly associated with the use of COM enumerators in C++; is flexible in allowing different references, cloning policies, and quanta; is discoverable in use (though I'll concede not transparent in implementation); and adheres to the *Principle of Least Surprise*. But in the interest of balance, it's appropriate to quickly explore a few alternative approaches.

28.10.1 Why Not Default to Forward Iterators?

In my experience, the majority of COM enumerators are cloneable and repeatable. One option would be to assume that for all cases and to throw exceptions when the enumerator fails to live up to expectations. On the positive side, this would certainly have made for a simpler implementation and would also allow users to always "know" that they're dealing with forward iterators, which, as we well know, gentle readers, are considerably simpler to grok than input iterators. Unfortunately, in the general case, enumerator capabilities are determinable only at runtime. A user might craft an enumerator-manipulating component using `enumerator_sequence` into a binary component and then ship that off to customers or make it available for free download. Naturally, someone somewhere will use that with noncloneable/nonrepeatable enumerators, and things will be unpleasant.

28.10.2 Why Not Default to Input Iterators?

The alternative would be to assume input iterators only. Although this would never suffer runtime disappointment as in the previous case, users of the sequence would be denied the more sophisticated semantics afforded by the forward iterator category. That's perfectly fine with enumerators like those provided by **recls/COM**, but consider the case where you're writing a driver program for, say, Microsoft Word, perhaps because you're writing a book on STL programming and need to automatically generate phrases for the index, and you require forward iterator semantics on the collections exhibited by the Word object model. You see how awful that could be?

28.10.3 Why Not Have a Fixed Quanta of 1?

Again, this alternative would have saved implementation effort at the cost of a suboptimal solution. Those cases where enumerators operate outside their own thread context are rare, but they exist, and it's too important to deny them the facility of batch retrieval.

28.10.4 Why Not Use a Standard Container?

This probably seems like the most obvious alternative, but it's also the most logically flawed. It's a simple one to answer: COM enumerators are not guaranteed to be exhaustible, so if you try to load all the elements into a container, you might be waiting a very long time. Further, although most COM enumerators do operate over a finite set of elements, some of these sets are very large. For example, just searching for C and C++ header files on the main work drive on my Windows machine yielded 47,000+ files and took 200 seconds to do. Imagine using a component that must experience such a latency to populate a container, when all you're after is maybe one or two of the files, which could have been encountered early in the search.

28.11 Coming Next

In Chapter 30, we'll look at the other half of the collection/iterator picture in COM, when we look at an adaptor for the COM collections protocol. But before that there's a little intermezzo wherein I clear up the issue of the `pointer` member type definition and concede to another piece of design that is lacking in prescience.

CHAPTER 29

Intermezzo: Correcting Minor Design Omissions with Member Type Inference

Those people who think they know everything are a great annoyance to those of us who do.

—Isaac Asimov

In the last chapter, I mentioned that the `pointer` member type of `enumerator_sequence` had a somewhat peculiar definition. The reason this is an issue at all is that the initial version of `enumerator_sequence` did not provide member selection operators and therefore didn't define a `pointer` member. Because the `enumerator_sequence` iterators are *transient* (Section 3.3.4), they can support the member selection operator and were updated to do so. The problem is that the template parameter list was already set, and there's no reasonable way to allow the user to stipulate the pointer type without breaking backward compatibility. Let's see how `pointer` is *actually* defined:

```
  . . .
#ifdef STLSOFT_META_HAS_SELECT_FIRST_TYPE_IF
    typedef typename select_first_type_if<value_type const*
                                         , value_type*
                                         , base_type_traits<R>::is_const
                                         >::type          pointer;
#else /* ? STLSOFT_META_HAS_SELECT_FIRST_TYPE_IF */
    typedef value_type*                                   pointer;
#endif /* STLSOFT_META_HAS_SELECT_FIRST_TYPE_IF */
  . . .
```

The pointer has two definitions, depending on whether the compiler supports the use of the metaprogramming IF (`select_first_type_if`; Section 13.4.1), which is largely tied in with whether the compiler supports partial template specialization. Since the standard requires this support, and all modern compilers provide it, we'll focus on that first. In that case, the mutability of the reference (template parameter `R`) is determined using `base_type_traits` (Section 12.1.1). Type selection (Section 13.4.1) is then used to select the appropriate mutability of `pointer`. Simple.

Without support for meta IF, the choices are simple: Make `pointer` mutating or nonmutating. Making it nonmutating would be fine for most cases but would mean that the user would be denied the wish to use the member selection operator in some cases. Since COM interfaces do not concern themselves with `const`-ness, instead putting the burden of doing the right thing on the user, I think it's appropriate in this case to follow suit. Since the `pointer` member type affects only the return type of the member selection operator, which can be invoked only on a class type, this is largely a minor issue.

CHAPTER 30

Adapting COM Collections

Age and treachery will overcome youth and skill.
—Fausto Coppi

Stoke me a clipper, I'll be back for Christmas!
—Arnold J. Rimmer, *Red Dwarf*

30.1 Introduction

The *COM collection* model represents collections with a loosely defined interface containing some or all of the following members: the methods `Add()`, `Clear()`, and `Remove()` and the properties `Count` and `Item`. (When expressed in C/C++, COM properties are prefixed with `get_`, e.g., `get_Count()`.) The one mandatory member is `_NewEnum`, which returns an enumerator in the form of a COM enumerator interface (Section 28.3). Just for grins, `_NewEnum` may be either a method or a property. By convention COM collections provide enumerators with the `IEnumVARIANT` interface to be compatible with late-binding code.

This simple and powerful protocol, although cumbersome to work with from C and C++, has been tried and tested over the past decade or so, and it underpins an enormous number of COM applications. If you have customized just about *any* Microsoft application—Word, Excel, Visual Studio—you've used collection objects and enumerated their contents.

30.2 Motivation

30.2.1 Longhand Version

Say we want to use **recls/COM** to look for all the C++ source files in the current directory and print their relative paths to the screen. We'll assume, as we did in Section 28.2, the existence of a function `open_search()`, this time defined as follows:

```
HRESULT open_search(char const*          directory
                  , char const*          patterns
                  , long                 flags
                  , ISearchCollection** ppcoll) throw();
```

`ISearchCollection` is the main collection interface in **recls/COM**, defining just the property `_NewEnum`. Listing 30.1 shows its definition in the Microsoft variant of Interface

Definition Language (IDL), which is used to define language-independent interface definitions. Note that, by convention, its return value is `IUnknown*` (also known by the typedef `LPUNKNOWN`) rather than any specific enumerator interface; the caller must use `IUnknown::QueryInterface()` to elicit the required enumerator interface.

Listing 30.1 IDL Definition of `ISearchCollection`

```
[ object, dual,
  uuid(2CCEE26C-B94B-4352-A269-A4EE84908367),
  pointer_default(unique)
]
interface ISearchCollection
  : IDispatch
{
  [propget, id(DISPID_NEWENUM), restricted, hidden]
  HRESULT _NewEnum([out, retval] IUnknown** pVal);
};
```

In C++, it has the more comprehensible form shown here:

```
struct ISearchCollection
  : public IDispatch
{
  virtual HRESULT get__NewEnum(IUnknown** pVal) = 0;
};
```

Listing 30.2 shows how we would use this in regular C++.

Listing 30.2 Use of a COM Collection via Raw Interfaces

```
ISearchCollection* pSrchColl;
HRESULT            hr = open_search("."
                              , "*.c|*.cpp|*.h|*.hpp"
                              , RECLS_F_FILES | RECLS_F_RECURSIVE
                              , &pSrchColl);
if(SUCCEEDED(hr))
{
  LPUNKNOWN   punkEnum;
  hr = pSrchColl->get__NewEnum(&punkEnum); // Get enumerator object
  if(SUCCEEDED(hr))
  {
    IEnumVARIANT* pEnVar;
    hr = punkEnum->QueryInterface(IID_IEnumVARIANT  // Get enumerator
                                , reinterpret_cast<void**>(&pEnVar));
    if(SUCCEEDED(hr))
    {
      VARIANT entries[5];
      ULONG   cFetched;
      for(pEnVar->Reset();
20           SUCCEEDED(pEnVar->Next( STLSOFT_NUM_ELEMENTS(entries)
21                                 , &entries[0], &cFetched));
22          )
23       {
24         if(0 == cFetched)
25         {
26           break;
27         }
28         else
29         {
30           for(ULONG i = 0; i < cFetched; ++i)
31           {
32             if( VT_UNKNOWN == entries[i].vt ||
33                 VT_DISPATCH == entries[i].vt)
34             {
35               IFileEntry* pFe;
36               hr = entries[i].punkVal->QueryInterface(IID_IFileEntry
37                                     , reinterpret_cast<void**>(&pFe));
38               if(SUCCEEDED(hr))
39               {
40                 std::wcout << comstl::propget_cast<comstl::bstr>(pFe
41                                            , &IFileEntry::get_Path)
42                            << std::endl;
43                 pFe->Release();
44               }
45             }
46             ::VariantClear(&entries[i]);
47           }
48         }
49       }
50       pEnVar->Release();
51     }
52     punkEnum->Release();
53   }
54   pSrchColl->Release();
55 }
```

That's a whole lot of code, even with error handling elided. Lines 17–50 are similar to the example from the chapter on COM enumerators (Listing 28.1), and all the same problems outlined in that case apply here: The code is verbose, it's low on transparency, it's fragile, and it's not exception-safe. Note that the enumerator interface is `IEnumVARIANT`, which leads to the additional effort to retrieve the required type from the `VARIANT` instances it retrieves in lines 32–38. Further, there appears to be a lot of monkey business to get an enumerator object from the collection.

Actually, there's too much monkey business. When I wrote this program while preparing this chapter, I fluffed the code. What are now lines 13–14 was previously:

```
13     hr = pSrchColl->QueryInterface(IID_IEnumFileEntry
14                                  , reinterpret_cast<void**>(&punkEnum));
```

This failed at runtime and had me debugging right into the internals of **recls/COM**, when perhaps what I should have been doing was checking my work at least once before running it. But I digress. The point is that the compiler is of little help when using COM, and this mistake illustrates that quite conclusively.

30.2.2 Shorthand Version

Of course, the answer lies in the use of a wrapper, or ***Façade***, in this case `comstl::collection_sequence`, which answers *all* the problems attendant in the longhand version. Listing 30.3 shows `collection_sequence` in action.

Listing 30.3 Use of a COM Collection via `collection_sequence`

```
ISearchCollection* pSrchColl;
HRESULT            hr = open_search("."
                                  , "*.c|*.cpp|*.h|*.hpp"
                                  , RECLS_F_FILES | RECLS_F_RECURSIVE
                                  , &pSrchColl);
if(SUCCEEDED(hr))
{
  typedef comstl::collection_sequence<ISearchCollection
                                  , IEnumFileEntry
                                  , IFileEntry*
                                  , comstl::interface_policy<IFileEntry>
                                  >     collection_t;
  collection_t  en(pSrchColl, false, 5); // Eat ref; can't throw
  { for(collection_t::iterator b = en.begin(); en.end() != b; ++b)
  {
    std::wcout << comstl::propget_cast<bstr>(en[i]
                                  , &IFileEntry::get_Path) << std::endl;
  }}
}
```

Just as was the case with `enumerator_sequence` (Section 28.5), the only downside to using this code over the longhand form is understanding the specialization of the `collection_sequence` template. All the hassles with manipulating the interfaces and values that occupied lines 8–39 and 43–54 have essentially been boiled down to constructing the collection object and enumerating its contents via the iterators returned by `begin()` and `end()`.

You might be wondering why we should bother with collection interfaces when we can use the (somewhat) more succinct enumerator interface. The reason is precisely the same as why we don't just have iterators without containers. And following the collection model has another advantage: Your COM libraries are seamlessly available to any automation language, including implicit use of enumerators in language-specific constructs such as Ruby's `each`.

30.3 comstl::collection_sequence

The **COMSTL** `collection_sequence` class template abstracts the COM collection interface as an STL collection and the associated enumerator interface as its iterator pairs. It handles elicitation of the enumerator, manages all reference counts safely, and provides input or forward iterators with transient element reference category semantics.

30.3.1 Public Interface

The definition of `comstl::collection_sequence` is shown in Listing 30.4. As it has much in common with `enumerator_sequence` (see Listings 28.3 and 28.26), I have highlighted the salient differences.

Listing 30.4 Definition of `collection_sequence`

```
// In namespace comstl
template< typename CI                                 // Collection interface
        , typename EI                                 // Enumerator interface
        , typename V                                  // Value type
        , typename VP                                 // Value policy type
        , typename R   = V const&                     // Reference type
        , typename CP  = input_cloning_policy<EI> // Cloning policy
        , size_t   Q   = 8                            // Quanta
        , typename EAP = new_enum_property_policy<CI> // Enumerator-
                                                      // acquisition policy
        >
class collection_sequence
{
private: // Member Types
  typedef enumerator_sequence<EI, V, VP, R, CP, Q>  enum_seq_type;
public:
  typedef collection_sequence<CI, EI, V, VP, R, CP, Q, EAP> class_type;
  typedef CI                            collection_interface_type;
  typedef EAP                           enumerator_acquisition_policy_type;
  typedef typename enum_seq_type::value_type   value_type;
  . . .  // And enumerator_interface_type, value_policy_type, reference,
pointer, iterator, cloning_policy_type, iterator_tag_type, size_type,
and difference_type
  enum     { retrievalQuanta = enum_seq_type::retrievalQuanta };
public: // Construction
  collection_sequence(collection_interface_type*  i
                     , bool                        bAddRef
                     , size_type                   quanta = 0);
  ~collection_sequence() throw();
public: // Iteration
  iterator begin() const;
  iterator end() const;
public: // Size
  size_type size() const;
private: // Invariant
  bool is_valid() const;
private: // Implementation
  static size_type validate_quanta_(size_type quanta);
private: // Member Variables
  collection_interface_type*  m_root;
  size_type const             m_quanta;
private: // Not to be implemented
  collection_sequence(class_type const&);
  class_type const& operator =(class_type const&);
};
```

The main thing to note from the interface is how similar it is to that of `enumerator_sequence`. The major difference is that there are two additional template parameters, taking the total to a somewhat alarming eight. In its (and my) defense, four are defaulted. But *Henney's Hypothesis* (Chapter 14) is hanging over this class template like a five-year-old boy over a bowl of ice cream. As with `enumerator_sequence`, the justification or amelioration is that the overblown template parameter list is its only flaw; in all other aspects, `collection_sequence` makes using COM collections in C++ a breeze. But I would say that, wouldn't I?

30.3.2 Member Types and Constants

The member types and constants are very similar to those of `enumerator_sequence`. Indeed, the private member type `enum_seq_type` is a specialization of `enumerator_sequence` using the given parameters, which is then used to define the `collection_sequence`'s member types and constant. There are only two distinctly new member types, corresponding to the two additional template parameters. The first, `collection_interface_type`, is the type of the collection interface, for example, `ISearchCollection`. The second, `enumerator_acquisition_policy_type`, has a less obvious purpose, which we'll deal with in Section 30.4. (If you like challenging pedagogical digressions, you might postulate what purpose this policy serves before reading on. Hint: There's a strong clue in the first paragraph of this chapter.)

Because the iterator type is that of the corresponding `enumerator_sequence`, the iterator category is also the same. Once more, it's incumbent on the user to *know* that the collection and its enumerator support *forward* iterator semantics should he or she elect to explicitly stipulate a cloning policy; if not, the specialization will support *input* iterators only and will always be quite safe and correct in doing so.

30.3.3 Construction

The constructor and destructor (Listing 30.5) have virtually identical definitions to those of `enumerator_sequence`. The only noteworthy difference is that there's no `bReset` parameter since a collection interface does not have a `Reset()` method: Enumerators are always elicited in their initial position.

Listing 30.5 Constructor and Destructor

```
  . . .
public: // Construction
  collection_sequence(collection_interface_type*  i
                    , bool                        bAddRef
                    , size_type                   quanta = 0)
    : m_root(i)
    , m_quanta(validate_quanta_(quanta))
  {
    COMSTL_ASSERT(NULL != i);
    if(bAddRef)
    {
      m_root->AddRef();
    }
    COMSTL_ASSERT(is_valid());
  }
  ~collection_sequence() throw()
  {
    COMSTL_ASSERT(is_valid());
    m_root->Release();
  }
  . . .
```

30.3.4 Iteration: Using a Dirty Trick Cleanly

The original version of `collection_sequence` had a full implementation that was identical to that of `enumerator_sequence` in large part; the copy-and-paste demon was dancing that day. Naturally, such a thing is a maintenance nightmare, and eventually I bit the bullet and started working on refactoring the common features into a shared component. But I detest sharing iterator types between components when the iterator is managing a resource because it means either declaring public what should be a `private` (conversion) constructor or declaring friendship across header files; neither of these sits easily with my C++onscience. (Not that either of these is as bad as having independent copies of code as complex as the implementation of `enumerator_sequence::iterator`, of course.)

Thankfully, a decision deferred often yields a superior answer. In this case, it dawned on me that I don't need to go to any of these unpleasant measures. (That's doublespeak for "I went to bed tired, and I woke up lazy.") What I can do *in this case* is use the ordinarily dirty trick of using the iterator instances returned from a temporary collection instance, as shown in Listing 30.6 (with error handling elided for brevity).

Listing 30.6 Iteration Methods

```
collection_sequence<. . .>::iterator begin() const
{
  COMSTL_ASSERT(is_valid());
  LPUNKNOWN punkEnum;
  HRESULT   hr = enumerator_acquisition_policy_type::acquire(m_root
                                                         , &punkEnum);
  if(SUCCEEDED(hr))
  {
    enumerator_interface_type*  ei;
    hr = punkEnum->QueryInterface(
                            IID_traits<enumerator_interface_type>::iid()
                          , reinterpret_cast<void**>(&ei));
    punkEnum->Release();
    if(SUCCEEDED(hr))
    {
      COMSTL_ASSERT(is_valid());
      return enum_seq_type(ei, false, m_quanta).begin(); // Temporary!
    }
    else
      . . .
}
collection_sequence<. . .>::iterator end() const
{
  COMSTL_ASSERT(is_valid());
  return iterator();
}
```

Why is this valid? Well, the prosaic reason is that I wrote both STL collections, and I just know that it is. However, that's not of great help in the grand scheme of things. The *official* reason that it's okay is that the documentation for `enumerator_sequence` contains the following clause:

```
/// \note The iterators returned by begin()/end() are valid outside
/// the lifetime of the collection instance from which they're obtained
```

There is not (yet) a mechanism for codifying this concept—where a collection's iterator instances are valid outside the collection instance's lifetime—so for the moment such documentation must suffice. There is, however, a workable mechanism for codifying a check on the concept for iterators of class type. The collection class can maintain an internal list of extant iterators and can assert or log upon its destruction if the list is not empty; naturally such a mechanism would have to be optional and would likely be used only in debug/test builds.

In a sense this is just formalizing a leaky abstraction (Chapter 6); our dirty trick is actually clean because we are using a documented feature of the `enumerator_sequence` collection. But it's fundamentally no different from, say, knowing that `std::vector`'s iterator is contiguous whereas that of `std::deque` is "merely" random access.

You might wonder whether the line incorporating the temporary `enumerator_sequence` could lead to a leak of the reference count associated with `ei`. Thankfully, that's not the case. As noted in Section 28.5.5, the `enumerator_sequence` constructor guarantees not to throw an exception. If `begin()` throws an exception, the `enumerator_sequence` instance will be destroyed since C++ guarantees to destroy all fully constructed objects when processing an exception. So it's perfectly proper, even though it may look a little odd. (In case you're wondering, I use the single-statement form because it unequivocally states that we are retrieving an iterator from a temporary instance.)

30.3.5 `size()`?

The `enumerator_sequence`, like many other sequences we've looked at, was not able to provide an implementation of a `size()` method because the complexity of doing so would be decidedly *not* constant-time and because COM enumerators are not required to give out the same, or even any, elements on a second traversal of their underlying elements.

Remember that I said at the start of the chapter that all members of COM collections, other than `_NewEnum`, are optional. Also remember—if it's not already burned into your brain—that C++ compilers instantiate only what's used. Thus, it should be possible, in principle, to provide implementations for methods corresponding to the optional COM collection methods, without causing any problems with any particular collection interface that does not support them (unless we try to use them, of course). The problem here is that there's too much variation of the definitions of the methods between different collections. For example, just a quick grep through the Microsoft Platform SDK headers reveals the following signature differences for just the first few interfaces found.

- `Add()` methods take a `long` and return an `IBodyPart*`, or take no parameters and return an `IDispatch*`, or take a `VARIANT* + BSTR` and return nothing, or take a `BSTR + VARIANT + VARIANT` and return `SnapIn**`.
- `Item()` methods take a `long` and return an `IMessage*`, or take a `BSTR` and return a `VARIANT`, or take a `VARIANT` and return an `IDispatch*`.
- `Remove()` methods take a `long`, or a `BSTR`, or a `VARIANT`.

The `Add()` method is definitely in the too-hard basket. And handling `Item()` and `Remove()` would be quite challenging and would not likely have a payoff worth the effort: I leave them as an exercise for the reader. The only members that, where present, have a consistent signature are the `Clear()` method and the `Count` property, as follows:

```
struct ISomeCollection
  : public IDispatch
{
  . . .
  virtual HRESULT Clear() = 0;
  virtual HRESULT get_Count(long* pCount) = 0;
  . . .
```

I think it would be churlish to provide a `clear()` method for a collection to which no additive methods were also available. However, there is good value in making use of the `Count` property, so `collection_sequence` defines the `size()` method, as shown in Listing 30.7 (which includes its auto-documentation comment).

Listing 30.7 `size()` **Method**

```
  . . .
public: // Size
  /// Returns the number of items in the collection
  ///
```

```
/// \note This method will not compile for collection interfaces
/// that do not contain the get_Count() method
size_type size() const
{
  COMSTL_ASSERT(is_valid());
  ULONG   count;
  HRESULT hr = m_root->get_Count(&count);
  COMSTL_ASSERT(is_valid());
  return SUCCEEDED(hr) ? count : 0;
}
. . .
```

If a collection interface defines a `Count` property, you can call `size()` to retrieve advanced notice of how many items you may be enumerating via `begin()` and `end()`. If it does not, you'll receive a compilation error if you try.

30.4 Enumerator Acquisition Policies

As I mentioned at the start of the chapter, collections require only the `_NewEnum` member, and this member may be either a method or a property. In other words, its signature in C++ may be either:

```
interface ISomeCollectionOrOther
  : public IDispatch
{
  virtual HRESULT get__NewEnum(LPUNKNOWN* ppunk) = 0;
}
```

or:

```
interface ISomeCollectionOrOther
  : public IDispatch
{
  virtual HRESULT _NewEnum(LPUNKNOWN* ppunk) = 0;
}
```

The reason this inconsistency was allowed to develop is that COM collections started out as supporting only the `IDispatch` interface, rather than what is known as dual interfaces. These are interfaces that derive from `IDispatch` and therefore have its support for late (i.e., runtime) binding, but also define explicitly the (COM) methods and (COM) properties as (C++) methods of the derived class, for the convenience of any languages (such as C++) that support early (i.e., compile-time) binding. Both versions of `ISomeCollectionOrOther` shown above are dual interfaces. In addition to the direct method calls shown, the `Invoke()` method of the `IDispatch` parent interface can be used to invoke the enumerator property/method, as long as the dispatch identifier (DispId) for that member is known. Thankfully, the DispId for `_NewEnum` is a standard value, `DISPID_NEWENUM` (`-4`).

"So what?" you may be wondering. Well, this is where the final template parameter comes in: the enumerator acquisition policy. Three policies are defined in the **COMSTL** libraries. The `new_enum_property_policy` assumes that `_NewEnum` is defined as a property (in IDL) and therefore calls the (C++) method `get__NewEnum()`, as shown in Listing 30.8. The vast majority of COM collection interfaces I've encountered define `_NewEnum` as a property, including most of those defined in the Microsoft Platform SDK, so this is the default policy. I've left the comments in as this is what a user will be transported by their IDE to see if they specify an interface that does not have `_NewEnum` as a property.

Listing 30.8 Definition of `new_enum_property_policy`

```
template <typename CI>
struct new_enum_property_policy
{
  static HRESULT acquire(CI* pcoll, LPUNKNOWN* ppunkEnum)
  {
    COMSTL_ASSERT(NULL != pcoll);
    COMSTL_ASSERT(NULL != ppunkEnum);
    // If the compiler complains here that your interface does not
    // have the get__NewEnum method, then:
    //  - you're passing a pure IDispatch interface, so you need to use
    //    new_enum_by_dispid_policy, or
    //  - you're passing a collection interface that defines _NeWEnum as
    //   a method, so you need to use new_enum_method_policy, or
    //  - you're passing the wrong interface. Check your code to ensure
    //    you've not used the wrong interface to specialize
    //    comstl::collection_sequence.
    return pcoll->get__NewEnum(ppunkEnum);
  }
};
```

The alternate case, where `_NewEnum` is a method, is handled by `new_enum_method_policy`, shown in Listing 30.9.

Listing 30.9 Definition of `new_enum_method_policy`

```
template <typename CI>
struct new_enum_method_policy
{
  static HRESULT acquire(CI* pcoll, LPUNKNOWN* ppunkEnum)
  {
    COMSTL_ASSERT(NULL != pcoll);
    COMSTL_ASSERT(NULL != ppunkEnum);
    return pcoll->_NewEnum(ppunkEnum);
  }
};
```

If you don't know which type you're working with, or you're working with a pure `IDispatch` collection, you can use `new_enum_by_dispid_policy`, shown in Listing 30.10. This one is a bit more involved than the other two since it needs to query for the

IDispatch interface, invoke the intensely verbose Invoke() method, and elicit the returned enumerator interface pointer from the VARIANT result.

Listing 30.10 Definition of new_enum_by_dispid_policy

```
template <typename CI>
struct new_enum_by_dispid_policy
{
  static HRESULT acquire(CI* pcoll, LPUNKNOWN* ppunkEnum)
  {
    COMSTL_ASSERT(NULL != pcoll);
    COMSTL_ASSERT(NULL != ppunkEnum);
    LPDISPATCH  pdisp;
    HRESULT     hr   = pcoll->QueryInterface(IID_IDispatch
                          , reinterpret_cast<void**>(&pdisp));
    if(SUCCEEDED(hr))
    {
      DISPPARAMS  params;
      UINT        argErrIndex;
      VARIANT     result;
      ::memset(&params, 0, sizeof(params));
      ::VariantInit(&result);
      hr = pdisp->Invoke( DISPID_NEWENUM, IID_NULL
                        , LOCALE_USER_DEFAULT
                        , DISPATCH_METHOD | DISPATCH_PROPERTYGET
                        , &params, &result
                        , NULL, &argErrIndex);
      pdisp->Release();
      if(SUCCEEDED(hr))
      {
        hr = ::VariantChangeType(&result, &result, 0, VT_UNKNOWN);
        if(SUCCEEDED(hr))
        {
          if(NULL == result.punkVal)
          {
            hr = E_UNEXPECTED;
          }
          else
          {
            *ppunkEnum = result.punkVal;
            (*ppunkEnum)->AddRef();
          }
        }
        ::VariantClear(&result);
      }
    }
    return hr;
  }
};
```

This version doesn't care whether the collection has defined `_NewEnum` as a method or a property. It just invokes it based on its DispId. But—there's always a but—don't think this is a panacea. If your type library does not contain the collection interface and your component uses type library marshalling (which automation components usually do), the `Invoke()` invocation will fail with the code *`TYPE_E_ELEMENTNOTFOUND`*, whose human-readable explanation gives the effusive *`Element not found.`* (You can make the library expose the interface by including an `interface IMyInterface;` statement *inside* the `library` block in the IDL.)

30.5 Summary

If you're not, as I clearly am, a fan of COM—and we must all acknowledge that only a mother could love its naked verbosity and fragility—you'll be relieved to know that this chapter wraps up all the COM discussion you're going to get in this book. But even if you don't appreciate COM, I hope you've taken the lessons of the various twists and turns as substance for generally applicable techniques. We've seen many ideas.

- You can use a policy-based class template to tame an archaic, verbose, and exception-unsafety-prone programming model, providing safe and succinct client code. The discoverability of the component is largely a win: The class template interface is not as discoverable as you would like, but the class interface is eminently so. And I would argue that the cost in understanding the policies for the class template is less arduous than is the effort of correct and exception-safe coding using raw COM.
- Where the abstraction leaks sufficiently to make such a determination, it can be appropriate to derive one STL collection's iterators from temporary instances of another STL collection—something that's not generally valid except in limited cases.
- You can handle syntactic inconsistencies in interfaces in a generic component based on compile-time and/or runtime measures, albeit with the help of the dynamic typing facilities of `IDispatch::Invoke()`.

(Assuming I haven't put you off using COM from C++ for life, you might want to check out the **VOLE** library, included on the CD, which allows you to drive COM Automaton servers in a robust and succinct manner. It's extremely easy to use and hides almost all the guff of COM Automation from you.)

CHAPTER 31

Gathering Scattered I/O

Power sits poorly on those who grow up in it.

—John O'Halloran

Tell an Australian he is not allowed to tap-dance and he will rush out to buy tap shoes.

—Simon Briggs

31.1 Introduction

Don't ever let anyone tell you that STL is nice and abstract and all that, but it just doesn't perform well. I poke holes in that myth in several places in this book (Sections 17.1.1, 19.1.1, 27.9, 36.2.2, and 38.2.1)—but this chapter blows it clean out of the water.

The subject matter of this chapter is Scatter/Gather I/O (also known as Scatter I/O), which means the exchange of data between application code and (usually kernel) I/O routines in the form of multiple blocks per action. Its primary intent is to allow client code to manipulate application-level packet headers separately from the payloads, but it can also be turned to other cunning ends and can yield considerable performance benefits. The cost is that it complicates the manipulation of data when logically contiguous data is spread over physically discontiguous blocks. To aid in handling such cases, we may use classes that (re)linearize the data back to an acceptably manipulable abstraction. Since this is a book about extending STL, the abstraction we will seek is that of an *STL collection* (Section 2.2) and its associated iterator ranges. But as we will see, there is a cost in such abstractions, so we will go further and examine how we might optimize the transfer of information without diluting the power of the abstraction. This will lead us into looking into the rules regarding overriding functions in the `std` namespace and how we may accommodate one with the other.

31.2 Scatter/Gather I/O

Scatter/Gather I/O involves the exchange of information between an I/O API and client code in a physically discontiguous form. In all cases I've come across, this discontiguous form involves a number of separate memory blocks. For example, the UNIX `readv()` and `writev()` functions act like their `read()` and `write()` siblings, but, rather than a pointer to a single area of memory and its size, they are passed an array of `iovec` structures:

```
struct iovec
{
  void*   iov_base;
  size_t  iov_len;
};
ssize_t readv(int fd, const struct iovec* vector, int count);
ssize_t writev(int fd, const struct iovec* vector, int count);
```

The Windows **Sockets** API has an analogous structure and corresponding functions:

```
struct WSABUF
{
  u_long  len;
  char*   buf;
};
int WSARecv(SOCKET  s
          , WSABUF* lpBuffers
          , DWORD   dwBufferCount
          , . . . // And 4 more parameters);
int WSASend(SOCKET  s
          , WSABUF* lpBuffers
          , DWORD   dwBufferCount
          , . . . // And 4 more parameters);
int WSARecvFrom(SOCKET  s
              , WSABUF* lpBuffers
              , DWORD   dwBufferCount
              , . . . // And 6 more parameters);
int WSASendTo(  SOCKET  s
              , WSABUF* lpBuffers
              , DWORD   dwBufferCount
              , . . . // And 6 more parameters);
```

You might wonder why people would want to perform I/O in such a fashion, given the obvious complication to client code. Well, if your file or network data has a fixed format, you can read one or more records/packets in or out without any need to move, reformat, or coalesce them. This can be quite a convenience. Similarly, if your records/packets have variable format but a fixed-size header, you can read/write the header directly to/from a matching structure and treat the rest as an opaque variable-size blob. And there's a third reason: performance. I once created a network server architecture using Scatter/Gather I/O that used a multithreaded nonlocking memory allocation scheme. (Suffice to say, it was rather nippy.)

But however much Scatter/Gather I/O may help in terms of performance, when dealing with variable-length records/packets, or those whose payloads contain elements that are variable-length, the client code is complicated, usually low on transparency, and bug-prone. An efficient abstraction is needed.

31.3 Scatter/Gather I/O APIs

31.3.1 Linearizing with COM Streams

The challenge with Scatter/Gather I/O is that using memory scattered over multiple blocks is not a trivial matter. On projects (on Windows platforms) in the 1990s, I tended to use a custom COM stream implementation from my company's proprietary libraries, which was implemented for a different task some years previously. Permit me to talk about the COM stream architecture for a moment. (I know I promised in the last chapter there would be no more COM, but there is a point to this, even for UNIX diehards. Trust me, I'm a doctor!)

A COM stream is an abstraction over an underlying storage medium having much in common with the file abstractions we're used to. Essentially, it has access to the underlying medium and defines a current point within its logical extent. A stream object exhibits the `IStream` interface (shown in abbreviated form in Listing 31.1), which contains a number of methods, including `Seek()`, `SetSize()`, `Stat()`, and `Clone()`. There are also methods for acquiring exclusive access to regions of the underlying medium. The `IStream` interface derives from `ISequentialStream` (also shown in Listing 31.1), which defines the two methods `Read()` and `Write()`.You can implement a stream for a particular underlying medium directly by deriving from `IStream` and providing suitable definitions for its methods.

Listing 31.1 Definition of the `ISequentialStream` and `IStream` Interfaces

```
interface ISequentialStream
  : public IUnknown
{
  virtual HRESULT Read(void* p, ULONG n, ULONG* numRead) = 0;
  virtual HRESULT Write(void const* p, ULONG n, ULONG* numWritten) = 0;
};
interface IStream
  : public ISequentialStream
{
  virtual HRESULT Seek(. . .) = 0;
  virtual HRESULT SetSize(. . .) = 0;
  virtual HRESULT CopyTo(. . .) = 0;
  virtual HRESULT Commit(. . .) = 0;
  virtual HRESULT Revert(. . .) = 0;
  virtual HRESULT LockRegion(. . .) = 0;
  virtual HRESULT UnlockRegion(. . .) = 0;
  virtual HRESULT Stat(. . .) = 0;
  virtual HRESULT Clone(. . .) = 0;
};
```

COM defines another stream-related abstraction, in the form of the `ILockBytes` interface (shown in abbreviated form in Listing 31.2). It abstracts arbitrary underlying mediums as a logically contiguous array of bytes. It does not maintain any positional state. Hence, it has `ReadAt()` and `WriteAt()` methods rather than `Read()` and `Write()`.

Listing 31.2 Definition of the `ILockBytes` Interface

```
interface ILockBytes
  : public IUnknown
{
  virtual HRESULT ReadAt( ULARGE_INTEGER pos, void* p
                        , ULONG n, ULONG* numRead) = 0;
  virtual HRESULT WriteAt(ULARGE_INTEGER pos, void const* p
                        , ULONG n, ULONG* numWritten) = 0;
  virtual HRESULT Flush() = 0;
  virtual HRESULT SetSize(. . .) = 0;
  virtual HRESULT LockRegion(. . .) = 0;
  virtual HRESULT UnlockRegion(. . .) = 0;
  virtual HRESULT Stat(. . .) = 0;
};
```

It is a relatively simple matter to implement a COM stream in terms of (an object that exhibits) the `ILockBytes` interface. All that's required is an `ILockBytes*` and a position. My company has just such an entity, accessible via the `CreateStreamOnLockBytes()` function:

```
HRESULT CreateStreamOnLockBytes(ILockBytes* plb, unsigned flags
                              , IStream** ppstm);
```

Obviously, the next question is, "How do we get hold of an `ILockBytes` object?" Again, there's a function for that, `CreateLockBytesOnMemory()`:

```
HRESULT CreateLockBytesOnMemory(void*         pv
                              , size_t        si
                              , unsigned      flags
                              , void*         arena
                              , ILockBytes**  pplb);
```

This supports a whole host of memory scenarios, including using a fixed buffer, using Windows "global" memory, using a COM allocator (`IAllocator`), and so on. One of the many flags is *SYCLBOMF_FIXED_ARRAY*, which indicates that `pv` points to an array of `MemLocBytesBlock` structures:

```
struct MemLockBytesBlock
{
  size_t  cb;
  void*   pv;
};
```

I'm not going to bang on about this much more, as hindsight is a harsh judge of things such as opaque pointers whose meanings are moderated by flags. The point I want to get across about this stuff is that I was able to take a set of memory blocks containing the scattered packet contents and get back an `IStream` pointer from which the packet information can be extracted in a logical and linear manner. Such code takes the following simple and reasonably transparent form. (Error handling is elided for brevity.) The `ref_ptr` instances are used to ensure that the reference counts are managed irrespective of any early returns and/or exceptions.

```
std::vector<WSABUF>   blocks      = . . .
size_t                payloadSize = . . .
ILockBytes*           plb;
IStream*              pstm;

SynesisCom::CreateLockBytesOnMemory(&blocks[1], payloadSize
                            , SYCLBOMF_FIXED_ARRAY | . . ., NULL, &plb);
stlsoft::ref_ptr<ILockBytes>  lb(pbl, false); // false "eats" the ref

SynesisCom::CreateStreamOnLockBytes(plb, 0, &pstm);
stlsoft::ref_ptr<IStream>     stm(pstm, false); // false "eats" the ref

. . . // Pass off stm to higher-layer processing
```

The stream can then be wrapped by a byte-order-aware instance adaptor class that works in partnership with a message object factory, to complete the mechanism for efficient translation from TCP packet stream segments to instances of higher-level protocol (C++) objects. The high efficiencies obtainable by such a scheme result from there being no allocations of, and no copying into, memory that does not constitute part of the final translated message object instances.

This is a powerful basis for a communications server model, one that I've used several times, albeit in different guises. In the case described earlier, a number of characteristics of the approach might incline you to search, as I have done, for better, less technology-specific solutions. First, the major downside of the described mechanism is that, being COM, the server code is effectively Windows-specific. Second, many developers (incorrectly) consider COM, as they (equally incorrectly) do C++ and STL, to be intrinsically inefficient, and it can be hard to disabuse them of that notion even with hard facts. Finally, add in the type-unsafe opaque pointers and the fact that the stream and lock-bytes classes were hidden proprietary implementations, and it all leaves something to be desired.

31.3.2 `platformstl::scatter_slice_sequence`—A Teaser Trailer

An alternate representation is to be found in a new, and still evolving, component in the **PlatformSTL** subproject: `scatter_slice_sequence`. This ***Façade*** class template maintains an array of slice structures describing a set of I/O buffers and provides methods for invoking native read/write functions on the set of buffers, in addition to providing STL collection access (in the form of `begin()` and `end()` methods). The class works with both `iovec` and `WSABUF` by abstracting their features with attribute shims (Section 9.2.1) **get_scatter_slice_size**, **get_scatter_slice_ptr**, and **get_scatter_slice_size_member_ptr**, shown in Listing 31.3.

Listing 31.3 Attribute Shims for the `iovec` and `WSABUF` Structures

```
#if defined(PLATFORMSTL_OS_IS_UNIX)

inline void* const get_scatter_slice_ptr(struct iovec const& ss)
{
  return ss.iov_base;
}
inline void*& get_scatter_slice_ptr(struct iovec& ss);

inline size_t get_scatter_slice_size(struct iovec const& ss)
{
  return static_cast<size_t>(ss.iov_len);
}
inline size_t& get_scatter_slice_size(struct iovec& ss);

inline size_t iovec::*
 get_scatter_slice_size_member_ptr(struct iovec const*)
{
  return &iovec::iov_len;
}
#elif defined(PLATFORMSTL_OS_IS_WIN32)
inline void const* get_scatter_slice_ptr(WSABUF const& ss)
{
  return ss.buf;
}
inline void*& get_scatter_slice_ptr(WSABUF& ss);

inline size_t get_scatter_slice_size(WSABUF const& ss)
{
  return static_cast<size_t>(ss.len);
}
inline size_t& get_scatter_slice_size(WSABUF& ss);

inline u_long WSABUF::* get_scatter_slice_size_member_ptr(WSABUF const*)
{
  return &WSABUF::len;
}
#endif /* operating system */
```

`scatter_slice_sequence` currently provides for `readv()`/`writev()` on UNIX and `WSARecv()`/`WSASend()` and `WSARecvFrom()`/`WSASendTo()` on Windows. Listing 31.4 shows an example that uses an `iovec` specialization of the class template to read the contents from one file descriptor into a number of buffers, processes the content in an STL kind of way, and then writes the converted contents to another file descriptor.

Listing 31.4 Example Use of `scatter_slice_sequence` with `readv()` and `writev()`

```
int fs  = . . . // Opened for read
int fd  = . . . // Opened for write
for(;;)
{
  const size_t  BUFF_SIZE = 100;
  const size_t  MAX_BUFFS = 10;
  char          buffers[MAX_BUFFS][BUFF_SIZE];
  const size_t  numBuffers  = rand() % MAX_BUFFS;

  // Declare an instance with arity of numBuffers
  platformstl::scatter_slice_sequence<iovec>  sss(numBuffers);

  // Set up each slice in the sequence, which may be of
  // different sizes in reality
  { for(size_t i = 0; i < numBuffers; ++i)
  {
    sss.set_slice(i, &buffers[i][0], sizeof(buffers[i]));
  }}
  if(0 != numBuffers) // In real scenario, might get 0 buffers
  {
    size_t  n = sss.read(::readv, fs); // Read from fs using ::readv()
    if(0 == n)
    {
      break;
    }
    // "Process" the contents
    std::transform( sss.payload().begin(), sss.payload().begin() + n
                  , sss.payload().begin(), ::toupper);
    sss.write(::writev, fd, n); // Write n to fd using ::writev()
  }
}
```

Obviously this example is very stripped down, but I trust your abilities to imagine that `fs` and `fd` might represent sockets, that the buffers shown here would be obtained from a shared memory arena (which may not have any to spare at a given time), and that the "processing" would be something less trivial than setting the contents to uppercase before (re)transmission.

The sequence's payload (available via `payload()`) provides random access iterators over the contents of its memory blocks. Just as with `std::deque`, it's important to realize that these iterators are *not* contiguous (Section 2.3.6)! Pointer arithmetic on the iterators is a constant-time operation, but iterating the range is not a linear-time operation. The `scatter_slice_sequence` is still a work in progress and its interface might evolve further before it's released into the **PlatformSTL** subproject proper. (It is on the CD.) But what it clearly provides is the ability to represent a given set of data blocks as an STL sequence (Section 2.2), along with adaptor methods `read()` and `write()` that take a file/socket handle and a Scatter/Gather I/O function and apply them to the blocks. This is the logical equivalent of the COM stream object created via

`CreateLockBytesOnMemory()` + *`SYCLBOMF_FIXED_ARRAY`* and `CreateStreamOnLockBytes()`. The one apparent disadvantage is that its contents have to be traversed one element at a time, something that may have performance costs. (Hint: This is a clue about something interesting to follow. . . .)

31.4 Adapting `ACE_Message_Queue`

The main subject of this chapter covers my efforts to adapt the memory queues of the **Adaptive Communications Environment** (**ACE**) to the STL collection concept, to serve the requirements of one of my recent commercial networking projects, a middleware routing service. To use the **ACE *Reactor*** framework, you derive event handler classes from `ACE_Event_Handler` (overriding the requisite I/O event handler methods) and register instances of them with the program's reactor singleton. When the reactor encounters an I/O event of a type for which an instance is registered, it invokes the appropriate callback method on the handler. When used with TCP, the Internet's stream-oriented transport protocol, the common idiom is to handle received data into instances of `ACE_Message_Block` and queue them in an instance of (a specialization of) the class template `ACE_Message_Queue`, as shown (with error handling omitted for brevity) in Listing 31.5.

Listing 31.5 A Simple Event Handler for the ACE Reactor Framework

```
class SimpleTCPReceiver
  : public ACE_Event_Handler
{
  . . .
  virtual int handle_input(ACE_HANDLE h)
  {
    const size_t        BLOCK_SIZE = 1024;
    ACE_Message_Block*  mb   = new ACE_Message_Block(BLOCK_SIZE);
    ssize_t             n    = m_peer.recv(mb->base(), mb->size());
    mb->wr_ptr(n);
    m_mq.enqueue_tail(mb);
    return 0;
  }
  . . .
private: // Member Variables
  ACE_SOCK_Stream                   m_peer; // Connection socket
  ACE_Message_Queue<ACE_SYNCH_USE>  m_mq;   // Message queue
};
```

The `ACE_Message_Queue` class acts as an ordered repository for all blocks, thereby faithfully representing the data stream. But `ACE_Message_Queue` is strictly a container of blocks; it does not attempt to provide any kind of abstracted access to the *contents* of the blocks. To access the contents of a message queue, you can use the associated class template, `ACE_Message_Queue_Iterator`, to iterate the blocks, as shown in Listing 31.6. The `ACE_Message_Queue_Iterator::next()` method returns a nonzero result and sets the given pointer reference to the block if a next block is available; otherwise, it returns *0*. The `advance()` method moves the current enumeration point to the next block (if any).

Listing 31.6 Example Code That Uses `ACE_Message_Queue_Iterator`

```
void SimpleTCPReceiver::ProcessQueue()
{
  ACE_Message_Queue_Iterator<ACE_NULL_SYNCH>  mqi(m_mq);
  ACE_Message_Block*                          mb;
  for(; mqi.next(mb); mqi.advance())
  {
    { for(size_t i = 0; i < mb->length(); ++i)
    {
       printf("%c", i[mb->rd_ptr()];
    }}
    mb->rd_ptr(mb->length()); // Advance read ptr to "exhaust" block
  }
}
```

Obviously, if you want to process a set of blocks as a logically contiguous single block, it's going to be a bit messy. We need a sequence to flatten the stream for STL manipulation.

31.4.1 `acestl::message_queue_sequence`, Version 1

The **ACESTL** subproject contains a number of components for adapting **ACE** to **STL** (and for making **ACE** components easier to use). `acestl::message_queue_sequence` is a class template that acts as an ***Instance Adaptor*** for the `ACE_Message_Queue`. Since this component's got quite a kick, I'm going to play my usual author's dirty trick of presenting you with a progression of implementations. Thankfully, unlike some material covered in other chapters, the changes between the versions are entirely additive, which should help keep me under 40 pages for this topic. Listing 31.7 shows the definition of the first version.

Listing 31.7 Definition of `message_queue_sequence`

```
// In namespace acestl
template <ACE_SYNCH_DECL>
class message_queue_sequence
{
public: // Member Types
  typedef char                                    value_type;
  typedef ACE_Message_Queue<ACE_SYNCH_USE>        sequence_type;
  typedef message_queue_sequence<ACE_SYNCH_USE> class_type;
  typedef size_t                                  size_type;
  class                                           iterator;
public: // Construction
  explicit message_queue_sequence(sequence_type& mq);
public: // Iteration
  iterator begin();
  iterator end();
public: // Attributes
  size_type  size() const;
  bool       empty() const;
private: // Member Variables
  sequence_type&  m_mq;
private: // Not to be implemented
  message_queue_sequence(class_type const&);
  class_type& operator =(class_type const&);
};
```

Given what we've seen with previous sequences, there's little here that needs to be remarked on; the interesting stuff will be in the `iterator` class. Note that the value type is `char`, meaning that `size()` returns the number of bytes in the queue, and [`begin()`, `end()`) defines the range of bytes. No methods pertain to message *blocks*.

31.4.2 acestl::message_queue_sequence::iterator

Listing 31.8 shows the definition of the `acestl::message_queue_sequence::iterator` class. Again, a lot here should be familiar based on prior experience. (I hope by now you're building a familiarity with these techniques, recognizing their similarities and identifying the differences between the different cases of their application. Naturally, it's my hope that this stands you in great stead for writing your own STL extensions.) The iterator category is *input iterator* (Section 1.3.1). The element reference category (Section 3.3) is *transient* or higher; in fact, it's *fixed*, with the caveat that no other code, within or without the defining thread, changes the contents of the underlying message queue or its blocks (in which case it would be *invalidatable*). The iterator is implemented in terms of a `shared_handle`, discussed shortly. I've not shown the canonical manipulation of the `shared_handle` in the construction methods since we've seen it before in other sequences (Sections 19.3 and 20.5).

Listing 31.8 Definition of `message_queue_sequence::iterator`

```
class message_queue_sequence<. . .>::iterator
  : public std::iterator<std::input_iterator_tag
                        , char, ptrdiff_t
                        , char*, char&
                        >
{
private: // Member Types
  friend class message_queue_sequence<ACE_SYNCH_USE>;
  typedef ACE_Message_Queue_Iterator<ACE_SYNCH_USE>   mq_iterator_type;
  struct                                              shared_handle;
public:
  typedef iterator                                    class_type;
  typedef char                                        value_type;
private: // Construction
  iterator(sequence_type& mq)
    : m_handle(new shared_handle(mq))
  {}
public:
  iterator()
    : m_handle(NULL)
  {}
  iterator(class_type const& rhs); // Share handle via AddRef() (+)
  ~iterator() throw(); // Call Release() (-) if non-NULL
  class_type& operator =(class_type const& rhs); // (+) new; (-) old
public: // Input Iteration
  class_type& operator ++()
  {
    ACESTL_ASSERT(NULL != m_handle);
    if(!m_handle->advance())
    {
      m_handle->Release();
      m_handle = NULL;
    }
    return *this;
  }
  class_type operator ++(int); // Canonical implementation
  value_type& operator *()
  {
    ACESTL_ASSERT(NULL != m_handle);
    return m_handle->current();
  }
  value_type operator *() const
  {
    ACESTL_ASSERT(NULL != m_handle);
    return m_handle->current();
  }
  bool equal(class_type const& rhs) const
  {
    return lhs.is_end_point() == rhs.is_end_point();
  }
private: // Implementation
  bool is_end_point() const
  {
    return NULL == m_handle || m_handle->is_end_point();
  }
private: // Member Variables
  shared_handle*  m_handle;
};
```

The iteration methods are implemented in terms of the methods of `shared_handle`. The endpoint state is identified by either a *NULL* handle or a handle that identifies itself as being at the endpoint. The preincrement operator advances by calling `shared_handle::advance()` and releases the handle when `advance()` returns *false*. The dereference operator overloads are implemented in terms of the `current()` overloads of `shared_handle`. Note that the mutating (non-`const`) overload returns a mutating reference, whereas the nonmutating (`const`) overload returns a `char` by value.

The main action lies in the `shared_handle`. Listing 31.9 shows its implementation. I'm going to invoke another low author tactic now and not explain the fine detail of the algorithm. I'll leave it as an exercise for you to figure out. To be fair, though, I will note that it skips empty `ACE_Message_Block` instances, which is how its endpoint condition can be so simple.

Listing 31.9 Definition of `shared_handle`

```
struct message_queue_sequence<. . .>::iterator::shared_handle
{
public: // Member Types
  typedef shared_handle   class_type;
public: // Member Variables
  mq_iterator_type    m_mqi;
  ACE_Message_Block*  m_entry;
  size_t              m_entryLength;
  size_t              m_entryIndex;
private:
  sint32_t            m_refCount;
public: // Construction
  explicit shared_handle(sequence_type& mq)
    : m_mqi(mq)
    , m_entry(NULL)
    , m_entryLength(0)
    , m_entryIndex(0)
    , m_refCount(1)
  {
    if(m_mqi.next(m_entry))
    {
      for(;;)
      {
        if(0 != (m_entryLength = m_entry->length()))
        {
          break;
        }
        else if(NULL == (m_entry = nextEntry()))
        {
          break;
        }
      }
    }
  }
private:
  ~shared_handle() throw()
  {
    ACESTL_MESSAGE_ASSERT("Shared handle destroyed with outstanding
references!", 0 == m_refCount);
```

```
  }
public:
  sint32_t AddRef();  // Canonical implementation
  sint32_t Release(); // Canonical implementation
public: // Iteration Methods
  bool is_end_point() const
  {
    return m_entryIndex == m_entryLength;
  }
  char&   current()
  {
    ACESTL_ASSERT(NULL != m_entry);
    ACESTL_ASSERT(m_entryIndex != m_entryLength);
    return m_entryIndex[m_entry->rd_ptr()];
  }
  char    current() const
  {
    ACESTL_ASSERT(NULL != m_entry);
    ACESTL_ASSERT(m_entryIndex != m_entryLength);
    return m_entryIndex[m_entry->rd_ptr()];
  }
  bool   advance()
  {
    ACESTL_MESSAGE_ASSERT("Invalid index", m_entryIndex <
m_entryLength);
    if(++m_entryIndex == m_entryLength)
    {
      m_entryIndex = 0;
      for(;;)
      {
        if(NULL == (m_entry = nextEntry()))
        {
          return false;
        }
        else if(0 != (m_entryLength = m_entry->length()))
        {
          break;
        }
      }
    }
    return true;
  }
private: // Implementation
  ACE_Message_Block* nextEntry()
  {
    ACE_Message_Block* entry = NULL;
    return m_mqi.advance() ? (m_mqi.next(entry), entry) : NULL;
  }
private: // Not to be implemented
  shared_handle(class_type const&);
  class_type& operator =(class_type const&);
};
```

31.5 Time for Some Cake

I hope you'll agree that being able to treat a set of ACE_Message_Block communications stream fragments as a logically contiguous stream eases considerably the task of dealing with such streams. In our middleware project, this enabled us to unmarshal the high-level protocol messages as objects, by the combination of another ***Instance Adaptor*** and a message ***Factory***. Permit me a small digression about the protocol. Standards Australia defines a protocol for the exchange of electronic payment implementation, called AS2805. This is a very flexible protocol, with a significant drawback. The messages do not contain any message-size information in their fixed-format header, and each message can contain a variable number of fields, some of which are of variable size. This means that you can't know whether all of a message has been received from the peer until the message has been fully parsed. Consequently, being able to easily and *efficiently* deconstruct a message is critical.

This was achieved by applying another instance adaptor to the acestl::message_queue_sequence instance to make it be treated as a streamable object, similar to how the logically contiguous ILockBytes instance was turned into a streamable object with CreateStreamFromLockBytes(). The streamable object is used by the message factory, which understands how to read the message type from the packet header and then uses that type to dispatch the appropriate unmarshaling function to read the remainder of the message contents and create a corresponding message instance. If insufficient data is available, the factory invocation fails benignly and the queue contents remain unchanged until the next I/O event. Only when a full message is retrieved is the requisite portion of the head of the queue removed and released back to the memory cache. If the message parsing fails for bad contents, the peer has sent bad data and the connection is torn down.

31.5.1 But Captain, I Canna' Mek Her Goo Any Fastah!

So, we've got some nice abstractions going—some of which have seen service in large-scale deployments, which is never to be sniffed at—but you may have been receiving portents from your skeptical subconsciousness. We're manipulating a number of blocks of contiguous memory, but the nature of the acestl::message_queue_sequence::iterator means that each byte in those blocks must be processed one at a time. This has to have an efficiency cost. And it does.

Before we proceed, I want to trot out the received wisdom on performance, namely, to avoid premature optimization. More precisely, although it's something of a simplification, you've usually got only one bottleneck in a system at a time. Inefficiencies usually make themselves felt only when a greater inefficiency has been resolved. In none of the applications in which I've used message_queue_sequence has it been associated with the bottleneck. However, I tend to be a little efficiency obsessed—What's that? You've noticed?—and since **STLSoft** is an open-source publicly available library, the message_queue_sequence component might find itself being a bottleneck in someone else's project, and that would never do. So I want to show you how to have your cake and eat it too, that is, how to linearize data block contents in an STL sequence and yet have block-like efficiency.

31.5.2 `acestl::message_queue_sequence`, Version 2

First, we need to identify when a block transfer is valid. The **ACE** libraries define opaque memory in terms of `char`, that is, the pointers are either `char const*` or `char*`, presumably to make pointer arithmetic straightforward. I don't hold with this strategy, but that's irrelevant; it is what it is. When transferring contents between STL iterators of type `char*` or `char const*` and `acestl::message_queue_sequence::iterator`, we want the sequence's contents to be block transferred. In other words, the following code should result in 2 calls to `memcpy()`, rather than 120 calls to `shared_handle::advance()`:

```
ACE_Message_Queue<ACE_NULL_SYNCH>   mq; // 2 message blocks, 120 bytes
char                                results[120];
acestl::message_queue_sequence<ACE_NULL_SYNCH>  mqs(mq);

std::copy(mqs.begin(), mqs.end(), &results[120]);
```

We want the same efficiency when transferring from contiguous memory into the message queue sequence, as in the following:

```
std::copy(&results[0], &results[0] + STLSOFT_NUM_ELEMENTS(results)
        , mqs.begin());
```

The first thing we need for this is to define block copy operations for `message_queue_sequence`. Listing 31.10 shows the definition of two new static methods for the sequence class, overloads named `fast_copy()`.

Listing 31.10 Definition of the `message_queue_sequence` Algorithm Worker Methods

```
template <ACE_SYNCH_DECL>
class message_queue_sequence
{
  . . .
  static char* fast_copy(iterator from, iterator to, char* o)
  {
#if defined(ACESTL_MQS_NO_FAST_COPY_TO)
    for(; from != to; ++from, ++o)
    {
      *o = *from;
    }
#else /* ? ACESTL_MQS_NO_FAST_COPY_TO */
    from.fast_copy(to, o);
#endif /* ACESTL_MQS_NO_FAST_COPY_TO */
    return o;
  }
  static iterator fast_copy(char const* from, char const* to
                          , iterator o)
  {
#if defined(ACESTL_MQS_NO_FAST_COPY_FROM)
    for(;from != to; ++from, ++o)
    {
      *o = *from;
    }
#else /* ? ACESTL_MQS_NO_FAST_COPY_FROM */
    o.fast_copy(from, to);
#endif /* ACESTL_MQS_NO_FAST_COPY_FROM */
    return o;
  }
  . . .
```

I've deliberately left in the `#defines` that suppress the block operations, just to illustrate in code what the alternative, default, behavior is. These `#defines` also facilitate tests with and without block copying enabled. (Anyone sniff a performance test in the near future?) The block mode code uses new `iterator::fast_copy()` methods, shown in Listing 31.11.

Listing 31.11 Definition of the `iterator` Algorithm Worker Methods

```
class message_queue_sequence<. . .>::iterator
{
  . . .
  void fast_copy(char const* from, char const* to)
  {
    if(from != to)
    {
      ACESTL_ASSERT(NULL != m_handle);
      m_handle->fast_copy(from, to, static_cast<size_type>(to - from));
    }
  }
  void fast_copy(class_type const& to, char* o)
  {
    if(*this != to)
    {
      ACESTL_ASSERT(NULL != m_handle);
      m_handle->fast_copy(to.m_handle, o);
    }
  }
```

Tantalizingly, these do very little beyond invoking the same-named new methods of the `shared_handle` class, shown in Listing 31.12. For both in and out transfers, these methods calculate the appropriate portion of each block to be read/written and effect the transfer with `memcpy()`.

Listing 31.12 Definition of the `shared_handle` Algorithm Worker Methods

```
struct message_queue_sequence<. . .>::iterator::shared_handle
{
  . . .
  void fast_copy(char const* from, char const* to, size_type n)
  {
    ACESTL_ASSERT(0 != n);
    ACESTL_ASSERT(from != to);
    if(0 != n)
    {
      size_type n1 = m_entryLength - m_entryIndex;
      if(n <= n1)
      {
        ::memcpy(&m_entryIndex[m_entry->rd_ptr()], from, n);
      }
      else
      {
        ::memcpy(&m_entryIndex[m_entry->rd_ptr()], from, n1);
        from += n1;
        m_entry = nextEntry();
        ACESTL_ASSERT(NULL != m_entry);
        fast_copy(from, to, n - n1);
      }
    }
  }
  void fast_copy(class_type const* to, char* o)
  {
    size_type n1 = m_entryLength - m_entryIndex;
    if( NULL != to &&
      m_entry == to ->m_entry)
    {
      ::memcpy(o, &m_entryIndex[m_entry->rd_ptr()], n1);
    }
    else
    {
      ::memcpy(o, &m_entryIndex[m_entry->rd_ptr()], n1);
      o += n1;
      m_entry = nextEntry();
      if(NULL != m_entry)
      {
        fast_copy(to, o);
      }
    }
  }
  . . .
```

31.5.3 Specializing the Standard Library

So far so good, but no one wants to write client code such as the following:

```
ACE_Message_Queue<ACE_NULL_SYNCH>   mq; // 2 locks; total 120 bytes
char                                results[120];
acestl::message_queue_sequence<ACE_NULL_SYNCH>  mqs(mq);

acestl::message_queue_sequence<ACE_NULL_SYNCH>::fast_copy(mqs.begin()
                                                , mqs.end(), &results[120]);
```

We want the invocation `std::copy()` to pick up our fast version automatically when the other iterator type is `char (const)*`. For this we need to specialize `std::copy()`.

For a number of reasons, defining partial template specializations in the `std` namespace is prohibited. This proves inconvenient in two ways. First, and most important, because `message_queue_sequence` is a template, we want to cater to all its specializations and so would want to do something like that shown in Listing 31.13. (For brevity I'm omitting the namespace qualification `acestl` from each `message_queue_sequence<S>::iterator` shown in this listing and the next.)

Listing 31.13 Illegal Specializations of `std::copy()`

```
// In namespace std
template <typename S>
char* copy( typename message_queue_sequence<S>::iterator from
          , typename message_queue_sequence<S>::iterator to, char* o)
{
  return message_queue_sequence<S>::fast_copy(from, to, o);
}
template <typename S>
typename message_queue_sequence<S>::iterator copy(char* from, char* to
                     , typename message_queue_sequence<S>::iterator  o)
{
  return message_queue_sequence<S>::fast_copy(from, to, o);
}
```

Since we may not do this, we are forced to anticipate the specializations of `message_queue_sequence` and (fully) specialize `std::copy()` accordingly, as in Listing 31.14. Note that separate `char*` and `char const*` specializations are required for the `char`-pointer-to-iterator block transfer, to ensure that copying from `char*` *and* `char const*` uses the optimization.

Listing 31.14 Legal Specializations of `std::copy()`

```
// In namespace std
template <>
char*
 copy( typename message_queue_sequence<ACE_NULL_SYNCH>::iterator from
     , typename message_queue_sequence<ACE_NULL_SYNCH>::iterator to
     , char*                                                     o)
{
  return message_queue_sequence<ACE_NULL_SYNCH>::fast_copy(from, to, o);
}
 . . . // Same as above, but for ACE_MT_SYNCH

template <>
typename message_queue_sequence<ACE_NULL_SYNCH>::iterator
 copy(char*                                                     from
    , char*                                                     to
    , typename message_queue_sequence<ACE_NULL_SYNCH>::iterator o)
{
  return message_queue_sequence<ACE_NULL_SYNCH>::fast_copy(from, to, o);
}
. . . // Same as above, but for ACE_MT_SYNCH

template <>
typename message_queue_sequence<ACE_NULL_SYNCH>::iterator
 copy(char const*                                               from
    , char const*                                               to
    , typename message_queue_sequence<ACE_NULL_SYNCH>::iterator o)
{
  return message_queue_sequence<ACE_NULL_SYNCH>::fast_copy(from, to, o);
}
. . . // Same as above, but for ACE_MT_SYNCH
```

Fortunately, **ACE** offers only two specializations, in the form of *`ACE_NULL_SYNCH`* (a `#define` for `ACE_Null_Mutex`, `ACE_Null_Condition`) and *`ACE_MT_SYNCH`* (a `#define` for `ACE_Thread_Mutex`, `ACE_Condition_Thread_Mutex`), yielding only six specializations.

But there's more. If, like me, you avoid like the plague the use of `char` as a substitute for C++'s missing `byte` type, you probably instead use `signed char` or `unsigned char`, both of which are distinct types from `char` when it comes to overload resolution (and template resolution). Passing these to an invocation of `std::copy()` will *not* succeed in invoking the optimized transfer methods. So, with heads bowed low, we need to provide another six specializations for `signed char` and six for `unsigned char`, yielding a total of eighteen specializations, for what we'd like to have been two, or at most three, were we able to partially specialize in the `std` namespace.

Thankfully, all this effort is worth the payoff. Before we look at that, I just want to answer one question you might be pondering: Why only `std::copy()`? In principle there is no reason to not specialize all possible standard algorithms. The reason I've not done so is twofold. First, the sheer effort in doing so would be onerous, to say the least; to avoid a lot of manual plugging around we'd be pragmatically bound to use macros and who likes macros? The second reason is more, well, reasoned. The whole reason for this optimization is to facilitate high-speed interpretation of data in its original memory block and high-speed exchange of data into new storage. In my experience, both of these involve `std::copy()`. I should admit one exception to this in our middleware project that required `copy_n()`. The `copy_n()` algorithm was overlooked for incorporation into the C++-98 standard (but will be included in the next version) and so appears in **STLSoft**. There are specializations of it, this time in the `stlsoft` namespace, in the same fashion as for `std::copy()`. Hence, there are a total of 36 function specializations in the *<acestl/collections/message_queue_sequence.hpp>* header file.

31.5.4 Performance

Now that we've examined the optimization mechanism, we'd better make sure it's worth the not inconsiderable effort. In order to demonstrate the differences in performance between the optimized block copying version and the original version, I used a test program that creates an `ACE_Message_Queue` instance to which it adds a number of blocks, copies the contents from a `char` array using `std::copy()`, copies them back to another `char` array (again with `std::copy()`), and verifies that the contents of the two arrays are identical. The number of blocks ranged from 1 to 10. The block size ranged from 10 to 10,000. The times for copying from the source `char` array to the sequence, and from the sequence to the destination `char` array, were taken separately, using the **PlatformSTL** component `performance_counter`. Each copying operation was repeated 20,000 times, in order to obtain measurement resolution in milliseconds. The code is shown in the extra material for this chapter on the CD.

Table 31.1 shows a representative sample of the results, expressed as percentages of the time (in milliseconds) taken by the equivalent nonoptimized version. As we might expect, with the very small block size of 10, the difference is negligible. For a buffer size of 100, there's an advantage with the optimized form, but it's not stunning. However, when we get to the more realistic buffer sizes of 1,000 and 10,000, there's no competition—the optimized form is 40–50 times faster.

Table 31.1 Relative Performance of Block Copy Operations

		Array to Iterator			Iterator to Array		
Number of Blocks	**Block Size**	**Nonoptimized**	**Block**	**%**	**Nonoptimized**	**Block**	**%**
1	10	7	6	85.7%	7	9	128.6%
2	10	9	8	88.9%	9	7	77.8%
5	10	16	10	62.5%	16	10	62.5%
10	10	27	14	51.9%	26	14	53.8%
1	100	25	7	28.0%	23	7	30.4%
2	100	46	9	19.6%	42	9	21.4%
5	100	108	14	13.0%	99	14	14.1%
10	100	211	23	10.9%	188	23	12.2%
1	1,000	207	10	4.8%	184	11	6.0%
2	1,000	416	16	3.8%	391	17	4.3%
5	1,000	1,025	32	3.1%	898	32	3.6%
10	1,000	2,042	60	2.9%	1,793	61	3.4%
1	10,000	2,038	29	1.4%	1,786	29	1.6%
2	10,000	4,100	101	2.5%	3,570	101	2.8%
5	10,000	4,143	109	2.6%	3,606	101	2.8%
10	10,000	4,103	102	2.5%	3,573	102	2.9%

31.6 Summary

This chapter has looked at the features of Scatter/Gather I/O, whose APIs present considerable challenges to STL adaptation. We've examined an adaptation, in the form of the `scatter_slice_sequence` component, and have seen that such sequences must have genuinely random access iterators (i.e., not contiguous iterators), for which the identity `&*it + 2 == &*(it + 2)` does *not* hold (see Section 2.3.6). Notwithstanding, we've seen how we can take advantage of their partial contiguity in order to effect significant performance improvements, something that is particularly important given their use in file and/or socket I/O. With minimal sacrifice of the *Principle of Transparency*, we've made big gains in the *Principle of Composition* (and also served the *Principle of Diversity* along the way).

CHAPTER 32

Argument-Dependent Return-Type Variance

Show me the man that's never been jealous.
—Lou Reed

There's nothing sexy about a guy in a funny hat, or a programmer who knows Fortran.
—George Frazier

32.1 Introduction

Q: How do you double a function's potential return value semantics?
A: *ARV it!*

This chapter looks at how other languages—in this case, Ruby—can influence the idioms of C++, specifically with regard to how a collection may act as both an array and an associative array (also known as a map, hash, or dictionary). This requires that different functions of the same name return different types based on the types of their arguments. Now you're probably thinking, "Well duh! That's just overloading." And so it is. But there's more to it than that; hence our story.

32.2 Borrowing a Jewel from Ruby

Consider the following Ruby code, which uses the **Open-RJ**/**Ruby** mapping:

```
# Open the database in the given file
db = OpenRJ::FileDatabase('pets.orj', OpenRJ::ELIDE_BLANK_RECORDS)
# Access the first record
rec = db[0]
# Print the fields in this record
(0 ... rec.numFields).each \
{ |i|
    fld = rec[i]
    puts "Field#{i}: name=#{fld.name}; value=#{fld.value}"
}
```

That's pretty regular Ruby and typical of code using the **Open-RJ**/**Ruby** mapping. (I could have more easily used `each_with_index`, but this way suits my pedagogical purposes a little better.) Using the Pets sample database that comes with the **Open-RJ** distribution, this prints out the following:

```
Field0: name=Name; value=Barney
Field1: name=Species; value=Dog
Field2: name=Breed; value=Bichon Frise
```

Consider now that rather than accessing the fields by index, we access them by name:

```
# Print the fields in this record
puts "Name="    + name=rec["Name"]
puts "Species=" + name=rec["Species"]
puts "Breed="   + name=rec["Breed"] if rec.include?("Breed")
```

This style is more useful when you have an expectation as to the structure of the data, verified by throwing an exception if either the *Name* or *Species* field is missing. This prints out as follows:

```
Name=Barney
Species=Dog
Breed=Bichon Frise
```

Look again carefully at the two uses of the subscript operator. In the first case the argument is an integer, and in the second case it is a string. Note the return types associated with these different calls. With an integral argument, the return value is a `Field` instance. The record has acted like an array. With a string argument, the return value is a string (representing the `value` member of the named `Field`). Now the record has acted like an associative array. When appropriate, this duality is a remarkably useful facility. It's appropriate in the case of **Open-RJ** records because **Open-RJ** database contents are immutable, fields are represented as *Name*+*Value* string pairs, and arrays of pointers to field structures are maintained in each record structure.

This functionality is implemented in the **Open-RJ**/**Ruby** mapping (written in C) via the `Record_subscript()` function and its two worker functions, `Record_subscript_string()` and `Record_subscript_fixnum()` (Listing 32.1). If the `index` argument is a string (*T_STRING*), `Record_subscript_string()` is invoked, returning a string representing the value of the named field. If `index` is an integer (*T_FIXNUM*), an instance of the field at the given index is returned. If the index is not known, is not in range, or is of the wrong type, an exception is raised to the caller.

Listing 32.1 C Implementation of Open-RJ/Ruby

```
static VALUE Record_subscript(VALUE self, VALUE index)
{
  switch(rb_type(index))
  {
    case T_STRING:
      return Record_subscript_string(self, StringValuePtr(index));
    case T_FIXNUM:
      return Record_subscript_fixnum(self, FIX2INT(index));
    default:
      rb_raise(rb_eTypeError, "field id must be integer or string");
    }
}
static VALUE Record_subscript_string( VALUE self, char const* index)
{
  ORJRecord const* record = Record_get_record_(self);
  ORJFieldA const* field  = ORJ_Record_FindFieldByNameA(record, index
                                                       , NULL);
  if(NULL == field)
  {
    rb_raise(cFieldNameError, "field not found: %s", index);
  }
  return rb_str_from_ORJStringA(&field->value);
}
static VALUE Record_subscript_fixnum(VALUE self, int index)
{
  ORJRecord const* record  = Record_get_record_(self);
  size_t           cFields = ORJ_Record_GetNumFieldsA(record);
  if( 0 <= index &&
      index < cFields)
  {
    VALUE  database_  = rb_iv_get(self, "@database_");
    return Field_create_(database_, self, &record->fields[index]);
  }
  else
  {
    rb_raise(rb_eIndexError, "field index out of range", index);
  }
}
```

32.3 Dual-Semantic Subscripting in C++

I wanted to emulate dual-semantic subscripting in the **Open-RJ**/**C++** mapping. A simplistic form of this would be as follows:

```
// In namespace openrj::stl
class Record
{
public: // Element Access
  const Field  operator [](size_t index) const;
  const String operator [](char const* name) const;
  . . .
```

This code works well, up to a point:

```
Record r;

r["Species"]; // Returns value (a String) of field named "Species"
r[1];         // Returns second field instance (a Field)
```

Alas, there are several drawbacks to this approach. First, consider what happens in the following case:

```
r[0]; // Compile error!
```

The problem is that literal *0* is equally convertible to an integral type that is not `int` as it is to a pointer type. We might solve this by changing the integral form to use `int`, but then we have the possibility of negative indices, which are not meaningful with an **Open-RJ** record.

```
class Record
{
public: // Element Access
  const Field  operator [](int index) const;
  const String operator [](char const* name) const;
  . . .
```

In any case, there's a much bigger issue here. The only string type with which the other overload is compatible is a C-style string (`char const*`). By now you're more than familiar with my predilection for generalized programming by manipulation of types by what they *do*, rather than what they strictly *are*, so you won't be surprised that I find this not the least satisfying. Maximally flexible classes should work with all string types, not just `char const*` and/or `std::string const&`.

32.4 Generalized Compatibility via String Access Shims

We can overload the named subscript operator of `Record` to work with *any* type for which the **c_str_ptr** string access shim (Section 9.3.1) is defined:

```
class Record
{
public: // Element Access
  const Field  operator [](size_t index) const;
  const String operator [](char const* name) const;
  template <typename S>
  const String operator [](S const& name) const
  {
    return operator [](stlsoft::c_str_ptr(name));
  }
  . . .
```

We can also access named field values with a multitude of types, as in the following:

```
std::string             s1("Name");
ACE_CString             s2("Species");
stlsoft::simple_string  s3("Breed");
```

```
r[s1];
r[s2];
r[s3];
```

32.5 A Fly in the `int`-ment

Alas, the problem is not quite solved. Looking again at the definition of `Record`, we see that we have three overloads of the subscript operator. With an argument of type `char const*` or `size_t` (or `int`, had we elected to use that form), the requisite non-template overload would be selected. With an argument of *any other type*, the function template overload would be selected and would attempt to apply the **c_str_ptr** shim to the argument. This would be quite a problem if the argument is a different integer type:

```
long n = 1;

r[n]; // Error: no c_str_ptr() overload matches 'long'
```

Naturally, having a `long` interpreted as something convertible to a field name string is quite against the intent of the `Record` subscript operators. One way to fix this is to define non-template overloads for *all* integral types:

```
class Record
{
public: // Element Access
  const Field  operator [](unsigned char index) const
  {
    return operator [](static_cast<unsigned int>(index));
  }
  const Field  operator [](signed char index) const;
  const Field  operator [](unsigned short index) const;
  const Field  operator [](signed short index) const;
  const Field  operator [](unsigned int index) const;
  const Field  operator [](signed int index) const;
  const Field  operator [](unsigned long index) const; // Aka size_t
  const Field  operator [](signed long index) const;
#if Visual C++ 6 or Intel compiler in VC6-compatibility mode
  const Field  operator [](unsigned __int32 index) const;
  const Field  operator [](signed __int32 index) const;
#endif
#if 64-bit integer supported?
  const Field  operator [](uint64_t const& index) const;
  const Field  operator [](sint64_t const& index) const;
#endif
  const String operator [](char const* name) const;
  . . .
```

Not exactly a pretty picture, is it? Lots of repeated code, and ugly preprocessor discrimination to boot. There should be a better way, and there is. We need a way for the compiler to react to an integral argument type by selecting the integral indexing operator and to use the string lookup operator for everything else.

We can't simply add a member function template for handling integer types:

```
public: // Element Access
  . . .
  template <typename I>
  const Field operator [](I const& index) const
  {
    return operator [](static_cast<unsigned int>(index));
  }
```

We already have a subscript operator template for use with strings, and the compiler would be understandably confused. We need to combine the behavior in one. This would be straightforward if the two methods had the same return type, but since they don't, a dash of cunning template metaprogramming is called for.

32.6 Selecting Return Type and Overload

We need to select the right overload and return type, which is achieved by combining two metaprogramming techniques: *type detection* (Section 13.4.2) and *type selection* (Section 13.4.1). Type detection—determining whether something is an integer—is performed by the `is_integral_type` template (Section 12.1.4). Type selection is performed by using the `select_first_type_if` template (Section 13.4.1). Hence the return type selection looks like this:

```
public: // Element Access
  . . .
  template <typename T>
  typename select_first_type_if<Field, String
                               , is_integral_type<T>::value
                               >::type
```

Overloads of a private worker method, `subscript_operator_()`, are defined as follows:

```
  template <typename S>
  String subscript_operator_(S const& name, no_type) const
  {
    return operator [](stlsoft::c_str_ptr(name));
  }
  template <typename I>
  Field subscript_operator_(I const& index, yes_type) const
  {
    return operator [](static_cast<size_t>(index));
  }
```

The correct overload is selected within the implementation of the subscript member function template via a temporary instance of the `type` member type of `is_integral_type`:

```
template <typename T>
typename select_first_type_if<Field
                              , String
                              , is_integral_type<T>::value
                              >::type operator [](S const& name) const
{
  typedef typename is_integral_type<T>::type   yesno_type;
  return subscript_operator_(name, yesno_type());
}
```

Note the unavoidable *DRY SPOT* (Chapter 5) violation in the duplication of the `is_integral_type` specialization, in the method signature for deducing return type and in the method body for selecting the worker function overload.

32.6.1 Proscribing Signed Subscript Indexes

For extra discipline we could add a constraint to force the integral to be unsigned, using the `is_signed_type` template (Section 12.1.5), as in the following:

```
template <typename I>
Field subscript_operator_(I const& index, yes_type) const
{
  STLSOFT_STATIC_ASSERT(0 == is_signed_type<I>::value);
  return operator [](static_cast<size_t>(index));
}
```

Such constraints are applied *after* the type has been detected to be integral, so there's no chance of it falling through into the string side of things.

32.7 Summary

And that's it! I call the technique *argument-dependent return-type variance* (ARV) since the return type of a function is selected not by the author of the code, but rather by the compiler on behalf of the user, based on the type of the argument. (Actually, it was my friend Bjorn Karlsson, a cunning linguist if ever there was one, who suggested the name.) The features of ARV are the following:

- Avoidance of fatuous language and warty ambiguities
- The ability to truly overload on concept without having to write a large number of identical overload method bodies
- The ability to deduce a return type of a member function template based on argument type

To be sure, there's a bit of metaprogramming to write or digest, but it leads to extremely efficient, flexible, and capable library code, supporting a minimum of fuss in client code, so it is worth the complexity. There are no runtime costs. It upholds the principles of *Composition* and *Least Surprise* with a minimal (in my opinion) transgression of *Transparency*.

CHAPTER 33

External Iterator Invalidation

Correctness must be a local property.

—Neils Ferguson and Bruce Schneier

It pays to be obvious, especially if you have a reputation for subtlety.

—Isaac Asimov

33.1 Element-Interface Coherence

As described in Section 1.2, the different standard containers have various rules for when and how their iterators (and pointers and references) may be invalidated in response to mutating changes on them. For example, in the following code, both `b` and `e` are invalidated:

```
std::vector<int>            ints;
ints.push_back(1);
ints.push_back(2);
std::vector<int>::iterator  b = ints.begin();
std::vector<int>::iterator  e = ints.end();
ints.erase(b); // b and e are invalidated
```

That's because the standard dictates that all iterators of an instance of `std::vector` after the point of erasure are invalidated (C++-03: 23.2.4.3). Conversely, in the following code, only `b` is invalidated:

```
std::list<int>            ints;
ints.push_back(1);
ints.push_back(2);
std::list<int>::iterator  b = ints.begin();
std::list<int>::iterator  e = ints.end();
ints.erase(b); // Only b is invalidated
```

That's because `std::list::erase()` invalidates only the iterators to the erased elements (C++ -03: 23.2.2.3). These rules can be defined so precisely and stipulated so unequivocally for one simple reason: Containers own their elements. The way in which containers and their elements are related and interact, their *element-interface coherence*, is total. Thus, as long as you follow the rules of the STL containers and don't attempt to use invalidated iterators, you won't get any nasty surprises.

With STL collections, however, things are less clear-cut: The coherence may necessarily be less than 100%. Consequently, there are no hard-and-fast rules, as we've seen in the examples from the previous chapters in this part.

- `glob_sequence` (Chapter 17) has exclusive ownership of the `glob_t` instance and is nonmutating. `glob()` supplies all matching file entries to the caller as a single snapshot. Thus, there are no issues of iterator invalidation. The process enumeration sequences `pid_sequence` and `process_module_sequence` (Chapter 22) operate in the same way, retrieving a complete snapshot of all available entries at a given epoch.
- `readdir_sequence` (Chapter 19) and `findfile_sequence` (Chapter 20) elicit entries from the underlying file system one at a time. However, the file system APIs insulate calling code from any "invalidation" of the current enumeration point. It may well be that actions by other threads or processes, or the operating system itself, modify an entry whose information is just about to be enumerated to the caller, but this does not interrupt the correct functioning of the API function(s). We don't have to know or care how this is achieved—the system may cache a snapshot of the entries at the time of enumeration, or it may serialize access to the *inode* list—we just know that we're going to get a full enumeration whose contents were logically valid at a given point in time. As users of a file system enumeration API, or an STL collection that wraps it, we know that the view we've been given is subject to change, so there is no problem and no likelihood of our collections going "bad" on us while we're using them.
- Although they have very different syntax, the collections wrapping COM enumerators (Chapter 28) and COM collections (Chapter 30) share the same insulation, provided by the COM enumerator instances, from changes to the underlying elements.
- The issue is moot with the `Fibonacci_sequence` (Chapter 23) and `string_tokeniser` (Chapter 27) collections because they generate their values themselves. These values do not have other physical existence anywhere, merely a notional existence insofar as they're interpreted by the collections.

None of these collections are subject to iterator invalidation by virtue of incomplete element-interface coherence, and therefore they don't suffer from what I call *external iterator invalidation*. However, we've seen cases where the boundaries between the underlying elements and the STL concepts have been a lot more malleable. For example, the `scatter_slice_sequence` and `message_queue_sequence` (Chapter 31) collections are arrangement-immutable (Section 2.2.1) but maintain coherence only when the client thread has exclusive access to the I/O slices.

Two collections that have no choice but to accept reduced coherence are the `environment_map` (Chapter 25) and Z-order sequences (Chapter 26). Indeed, each is subject to highly changeable underlying elements. `environment_map` uses the standard `getenv()` function for named-element retrieval and reference-counted, short-lived caches for iteration. The Z-order sequences eschew indexed retrieval and exhibition of the *bidirectional iterator* category and use a custom self-reversing iterator class template. In the case of `environment_map`, the incoherence comes about as a consequence of actions of other threads in the same process and of the possible legal side effects of the standard (`getenv()`) and nonstandard (`putenv()`, `setenv()`, `environ`) parts of the environment API. We explicitly don't cater to the former but cannot ignore the latter, which is the first kind of external invalidation: *invalidation by intrathread API side effects*.

In the case of the Z-order sequences, the incoherence stems from the fact that the `GetWindow()` API function elicits a sampling of one particular relation between two windows, in a set of windows that may be changed at any time by the actions of the user or of any other process. This is the second kind of external invalidation: *invalidation by extrathread action*.

Along with these two, there's also a third situation worth discussing. When using a collection that acts as a ***Façade*** over an existing API, it's possible to effect changes to the underlying elements by using the API independent of the collection. Ideally, we should refrain from doing so. But there are circumstances where the complexity of application code may be such that you can inadvertently alter the underlying elements without realizing you are doing so. This is the third kind of external invalidation: *invalidation by intrathread application side effects*. (This is still external invalidation because it's external to the collection.) Unlike the other two types, there is a reasonable degree of equivocation on whether this is a valid concern. Certainly, it can be argued that a greater degree of diligence and thoroughness can be applied to such problems, and failure to do so should be viewed as bad programming. I can see the sense in that point of view, but it fails to apply in practice, precisely because the potential for external manipulation and the likelihood of the programmer being unaware of it (at compile time) rise with the complexity of the application code and the level of abstraction of the (STL) adaptation. This is certainly something that must be judged on a case-by-case basis.

The `environment_map` class adequately demonstrated amelioration and obviation of invalidation by intrathread API side effects. But that's not always possible or desirable. In this chapter we'll examine components that handle these conditions actively: They must detect invalidation and report it to their client code. First (Section 33.2), we look at some STL extensions that act as ***Façades*** for Windows controls, that handle invalidation by intrathread API side effects. Then, in the main part of this chapter (Section 33.3), we'll tackle invalidation by extrathread action, illustrated by the much-criticized Windows **Registry** API and its corresponding STL extension ***Façades*** from the **WinSTL Registry** library. (In the additional material for this chapter included on the CD, we'll look at why some STL extensions for XML must, due to the complexity of the underlying XML libraries, handle invalidation by intrathread application side effects. Such components can employ the techniques described in Sections 33.2 and 33.3 to detect and report invalidation.)

33.2 Windows ListBox and ComboBox Controls

Window handles are global objects in the Windows operating systems. The text of a window, along with many of its other attributes, can be elicited and altered by threads other than the one that created and owns it. We won't dwell on the general objections to such a design, as I'd like to finish this book before I retire. We're just going to focus on one of the ramifications of this design as it pertains to the standard ListBox control.

The ListBox, like all other Windows controls, is manipulated by sending it messages via the `SendMessage()` family of functions. Items may be inserted into a ListBox by the *`LB_ADDSTRING`* and *`LB_INSERTSTRING`* messages and may be removed by the *`LB_DELETESTRING`* and *`LB_RESETCONTENT`* messages. *`LB_GETCOUNT`* returns the number of elements in the ListBox. Listing 33.1 illustrates their use. (Note that the third and fourth parameters to `SendMessage()`, known as `wParam` and `lParam`, are [32-bit on Win32] integers used to carry a variety of types, whether as integers or as cast pointers. Where a particular message does not use one or the other of the parameters, client code should pass a *0*.)

Listing 33.1 Manipulation of the ListBox Control Using Windows API Functions

```
HWND hwndListBox = ::CreateWindow("LISTBOX", "", LBS_SORT, . . . );

// Insert strings at their sorted position
::SendMessage(hwndListBox, LB_ADDSTRING, 0, (LPARAM)"String 3");
::SendMessage(hwndListBox, LB_ADDSTRING, 0, (LPARAM)"String 1");

// At this point ListBox contains 2 strings: "String 1" and "String 3"
assert(2 == ::SendMessage(hwndListBox, LB_GETCOUNT, 0, 0));

// Insert a string at the given position
::SendMessage(hwndListBox, LB_INSERTSTRING, 1, (LPARAM)"String 2");

// At this point ListBox contains 3 strings: "String 1", "String 2",
// and "String 3"
assert(3 == ::SendMessage(hwndListBox, LB_GETCOUNT, 0, 0));

// Delete the string at index 1
::SendMessage(hwndListBox, LB_DELETESTRING, 1, 0);

// At this point ListBox contains 2 strings: "String 1" and "String 3"
assert(2 == ::SendMessage(hwndListBox, LB_GETCOUNT, 0, 0));

// Delete all strings
::SendMessage(hwndListBox, LB_RESETCONTENT, 0, 0);
assert(0 == ::SendMessage(hwndListBox, LB_GETCOUNT, 0, 0));
```

It's important for authors of STL collections that will wrap the Windows ListBox (and ComboBox) controls to note that there is no *LB_UPDATESTRING* or *LB_REPLACESTRING* message. In other words, though you can add or delete a string, you cannot change it. This basically means that providing a meaningful mutable collection is not possible: Collections must be immutable.

33.2.1 Retrieval Races?

Retrieving the entries from a ListBox is achieved via the *LB_GETTEXT* message, as shown in the following code. (*LB_GETTEXT* is actually a `#define` for either *LB_GETTEXTA* or *LB_GETTEXTW*, depending on whether compiling for ANSI/multibyte or Unicode/wide string, that is, absent or present the definition of the `UNICODE` preprocessor symbol.)

```
char  str[100];

// Get the text for the element at index 1, with 100 byte guess
::SendMessage(hwndListBox, LB_GETTEXT, 1, (LPARAM)&str[0]);
```

Alas, this is a buffer overrun waiting to happen. What if the element at index 1 contains a string with more than 100 characters? We've not communicated the buffer length with the message, so the handler for that message cannot know the size of the buffer. The answer is that we must first use the associated message *LB_GETTEXTLEN* to get the length (in characters, not including the nul-terminator) of the element and ensure we have a buffer of the appropriate length, as shown in Listing 33.2. Note that this code tests the return value for *LB_ERR* (*-1*), which is returned if the index specified is not valid (i.e., out of range of the current contents).

Listing 33.2 Elicitation of Text Length from a ListBox Control

```
int r = ::SendMessage(hwndListBox, LB_GETTEXTLEN, 1, 0);
if(LB_ERR != r)
{
  stlsoft::auto_buffer<char>  str(1 + static_cast<size_t>(r));

  // Get the text for the element at index 1, with sufficient space
  ::SendMessage(hwndListBox, LB_GETTEXT, 1, (LPARAM)&str[0]);
}
```

Multitasking aficionados may now be scratching their heads and wondering whether they've spotted a potential race condition. Since window handles are shared among all threads on the system, surely it's possible for a list element whose length is greater than the current one at index `1` to be added or inserted such that its index is now 1, and for that to happen after the first `SendMessage()` call and before the second?

This was not a problem in 16-bit Windows because it was a cooperative multitasking system. A process would yield processing to another only by explicit action (which was usually when it went to the message queue to get the next message). But 32-bit Windows is a preemptive multitasking operating system, meaning that one thread can interrupt another at any time.

The migration from 16-bit to 32-bit Windows had to be as simple as possible (otherwise, people would be disinclined to make the jump), so the windowing APIs needed to stay as unchanged as possible. Creditably, Microsoft puts huge amounts of effort into assuring backward compatibility between different operating system releases, which makes a lot of business sense. How to solve this conundrum? Well, if you create two Win32 programs, one that creates and changes the elements in a ListBox, and one that reads the elements from the ListBox, you'll find that this race condition never eventuates. What's happening is that access to windows owned by another thread is serialized by a synchronization object (presumably a mutex). In other words, Win32 is acting like 16-bit Windows. How d'ya like them apples?

Whatever you may think of an operating system that shares user interface objects between all processes on the system, we must concede that use of the poorly designed 16-bit message APIs on Win32 is safe in all but the most perverse and unrealistic scenarios. (I've included an example of just such a scenario on the CD, and it is perverse and unrealistic.)

Nonetheless, since it's *possible* to encounter the race condition, our contract programming principles (Chapter 7) dictate that we must treat invalidation by extrathread action as a bona fide runtime case. In any case, it's certainly possible that invalidation by intrathread application side effects can occur. So a wrapper class should account for changes to the ListBox contents. There are only two ways to detect that the elements have changed. First, we can use *LB_GETCOUNT* to remember the number of elements in the ListBox and then check this every time we mutate the iterator. This is the more pure option since it detects iterator invalidation at the earliest point. However, as you may have noticed, by working in terms of indexes, the ListBox message API actually supports *random access* iteration. There are several ways in which an iterator can be mutated, for example, through ++, --, +=, and -=, in addition to the ability to offset a given iterator using pointer arithmetic. Since external iterator invalidation of any sort is something that we can expect to be rare, I chose instead to detect it when eliciting the value (string) of the element in the dereference and subscript operators.

33.2.2 WinSTL `listbox_sequence` and `combobox_sequence` Classes

Let me briefly introduce you to the `winstl::listbox_sequence` class template and its iterator class `winstl::listbox_const_iterator`. One thing Windows programmers will know is that the ListBox and ComboBox controls have very similar message APIs: The latter has *CB_GETLBTEXT*, *CB_GETLBTEXTLEN*, *CB_GETCOUNT*, and *CB_ADDSTRING*, with semantics analogous to those of their *LB_** counterparts. As a consequence, the iterator class also serves the `winstl::combobox_sequence` class; the differences are abstracted as a simple traits class. For ListBox controls, this is the `listbox_operation_traits` class, as shown in Listing 33.3. Note the use of both `char` and `wchar_t` forms of `get_text()`. This allows the sequence class, which is parameterized by string type, to work with both multibyte and wide windows regardless of whether the code is compiled for one or the other.

Listing 33.3 Definition of `listbox_operation_traits`

```
// In namespace winstl
struct listbox_operation_traits
{
public: // Operations
  static int err_constant()
  {
    return LB_ERR;
  }
  static int get_count(HWND hwnd)
  {
    return ::SendMessage(hwnd, LB_GETCOUNT, 0, 0);
  }
  static int get_text_len(HWND hwnd, int index)
  {
    return ::SendMessage(hwnd, LB_GETTEXTLEN, (WPARAM)index, 0);
  }
  static int get_text(HWND hwnd, int index, char* s)
  {
    return ::SendMessageA(hwnd, LB_GETTEXT, (WPARAM)index, (LPARAM)s);
  }
  static int get_text(HWND hwnd, int index, wchar_t* s)
  {
    return ::SendMessageW(hwnd, LB_GETTEXT, (WPARAM)index, (LPARAM)s);
  }
};
```

This traits class is used in the `listbox_sequence` class to parameterize `listbox_const_iterator` to derive the sequence's member type `const_iterator`, as shown in Listing 33.4. All the other member types are then defined in terms of this type.

Listing 33.4 Definition of `listbox_sequence`

```
// In namespace winstl
template <typename S> // String type, e.g., std::wstring, CString
class listbox_sequence
{
public: // Member Types
  typedef listbox_sequence<S>                     class_type;
  typedef listbox_const_iterator< S
                                , listbox_operation_traits
                                >                 const_iterator;
  typedef typename const_iterator::char_type      char_type;
  . . . // And so on for all other member types
private:
  typedef listbox_operation_traits                control_traits_type;
public: // Construction
  explicit listbox_sequence(HWND hwndListBox)
    : m_hwnd(hwndListBox)
  {}
public: // State
  size_type size() const
  {
    return size_type(control_traits_type::get_count(m_hwnd));
  }
  bool empty() const; // Returns 0 == size()
public: // Iteration
  const_iterator  begin() const
  {
    return const_iterator(m_hwnd, 0);
  }
  const_iterator  end() const
  {
    return const_iterator(m_hwnd, int(size()));
  }
  const_reverse_iterator rbegin() const
  {
    return const_reverse_iterator(end());
  }
  const_reverse_iterator rend() const
  {
    return const_reverse_iterator(begin());
  }
public: // Element Access
  value_type operator [](difference_type index) const
  {
    return const_iterator::get_value_at_(m_hwnd, index);
  }
private: // Member Variables
  HWND    m_hwnd;
};
```

The `listbox_const_iterator` class template is shown in Listing 33.5.

Listing 33.5 Definition of `listbox_const_iterator`

```
template< typename S  // String type
        , typename CT // Control traits type
        >
class listbox_const_iterator
  : public std::iterator<std::random_access_iterator_tag
                        , S, ptrdiff_t
                        , S const*, S const&
                        >
{
public: // Member Types
  typedef S                                    value_type;
  typedef value_type const&                    const_reference;
  typedef value_type const*                    const_pointer;
  . . .
  typedef CT                                   control_traits_type;
public: // Construction
  listbox_const_iterator(HWND hwndListBox, int index)
    : m_hwnd(hwndListBox)
    , m_index(index)
    , m_bRetrieved(false)
  {}
public: // Iterator Methods
  const_reference operator *() const;
  const_pointer   operator ->() const
  {
    return &operator *();
  }
  class_type& operator ++()
  {
    ++m_index;
    m_bRetrieved = false;
    return *this;
  }
  . . .
  class_type& operator --()
  {
    --m_index;
    m_bRetrieved = false;
    return *this;
  }
  . . .
  difference_type compare(class_type const& rhs) const
  {
    return m_index - rhs.m_index;
  }
  bool operator == (class_type const& rhs) const
  {
    return 0 == compare(rhs);
  }
private: // Member Variables
  HWND                m_hwnd;
  int                 m_index;
  mutable bool        m_bRetrieved;
  mutable value_type  m_value;
};
```

External iterator invalidation is handled by detection of the *LB_ERR* (or *CB_ERR*) code returned by one of the traits text methods in the iterator's dereference operator (Listing 33.6).

Listing 33.6 Definition of the Dereference Operator

```
const_reference listbox_const_iterator<. . .>::operator *() const
{
  if(!m_bRetrieved)
  {
    int len;
    if(control_traits_type::err_constant() ==
        (len = control_traits_type::get_text_len(m_hwnd, m_index)))
    {
      throw stlsoft::external_iterator_invalidation("external iterator
invalidation: control contents may have been altered externally");
    }
    buffer_type buffer(1 + len);
    if(control_traits_type::err_constant() ==
        (len = control_traits_type::get_text(m_hwnd, m_index
                                            , &buffer[0])))
    {
      throw stlsoft::external_iterator_invalidation("external iterator
invalidation: control contents may have been altered externally");
    }
    m_value.assign(&buffer[0], buffer.size() - 1);
    m_bRetrieved = true;
  }
  return m_value;
}
```

The ramifications of this scheme are that `size()` may not always equal `std::distance(begin(), end())`, and a full enumeration of a range from which elements have been removed will fail only when a dereference attempts to elicit the value of an element with an invalid index. But the code is well defined in conditions of externally changed control elements. Invalidation by intrathread application side effects is handled.

As for invalidation by extrathread action, the issue of a race condition between *`LB_GETTEXT`* and *`LB_GETTEXTLEN`* is handled by stipulating that the behavior of the `listbox_sequence` and `combobox_sequence` classes is not well defined if used outside a message handler. (This is standardeze for politely, if somewhat obliquely, saying "Do not use it in those circumstances.") This constraint will not trouble 99.999% of application programmers; they will certainly be manipulating these controls within message handlers. In any case, there's no alternative since there's no way to safely cater to the circumstance where *`LB_GETTEXT`* returns a larger value than *`LB_GETTEXTLEN`* in the general case.

33.3 Enumerating Registry Keys and Values

Now we come to the main game of this final chapter of Part II: handling invalidation by extrathread action. We'll use the Windows **Registry** API to illustrate the effects of this kind of external invalidation and to demonstrate a technique for handling it.

The Windows registry is a shared, system-wide hierarchical database wherein string, integer, and binary information may be stored. If you're not familiar with it, it's best to think of it as a set of file systems (known as *hives*) where registry keys act like directories, being repositories for registry values (like files) and as the parents for subkeys (like subdirectories). There are several crucial differences between these two concepts, including the fact that you cannot "move" to a parent key using *`".."`*, but it's a good enough mapping to serve as a basis for our understanding.

The **Registry** API is a suite of functions for the manipulation of registry keys and their values, including `RegCreateKey()`, `RegOpenKey()`, `RegCloseKey()`, `RegDeleteKey()`, `RegEnumKey()`, `RegEnumKeyEx()`, `RegSetValue()`, `RegQueryValue()`, `RegEnumValue()`, and so on.

For STL extenders, the functions `RegEnumKeyEx()` and `RegEnumValue()` are particularly interesting since they use indexing as their enumeration mechanism, suggesting subscriptable collections and random access iterators (Section 1.3.5).

```
LONG RegEnumKeyEx(
  HKEY      hKey,             // Handle to key to enumerate
  DWORD     dwIndex,          // Subkey index
  LPTSTR    lpName,           // Subkey name
  LPDWORD   lpcName,          // Size of subkey buffer
  LPDWORD   lpReserved,       // Reserved
  LPTSTR    lpClass,          // Class string buffer
  LPDWORD   lpcClass,         // Size of class string buffer
  PFILETIME lpftLastWriteTime // Last write time
);
```

`hKey` is the handle to the key whose subkeys will be enumerated. `dwIndex` is the index of the subkey whose information is to be retrieved. `lpName` is a pointer to a buffer to receive the key name. `lpcName` is a pointer to a `DWORD` (a 32-bit unsigned integer) that specifies the size of the buffer pointed to by `lpName` and receives the number of characters written into it. The last four parameters don't impact the discussion in this chapter, so I won't discuss them here. Enumeration is conducted by specifying an index of 0 for the first call and then incrementing upon each successful call, which returns *`ERROR_SUCCESS`* (*`0`*), until *`ERROR_NO_MORE_ITEMS`* is returned, as shown in Listing 33.7. If the buffer is not large enough, the function returns *`ERROR_MORE_DATA`*. Any other return indicates a genuine error, such as insufficient privileges.

Listing 33.7 Longhand Enumeration of Registry Keys Using Windows API

```
HKEY                         key   = . . .
DWORD                        index = 0;
stlsoft::auto_buffer<char>  buff(100);
for(;;)
{
  DWORD n   = buff.size();
  LONG  res = ::RegEnumKeyEx(key, index, &buff[0], &n
                            , NULL, NULL, NULL, NULL);
  if(ERROR_MORE_DATA == res)
  {
    buff.resize(2 * buff.size()); // Insufficient buffer
  }
  else if(ERROR_SUCCESS == res)
  {
    ::printf("subkey name=%.*s\n", int(buff.size()), buff.data());
    ++index;
  }
  else if(ERROR_NO_MORE_ITEMS == res)
  {
    break; // No more data
  }
  else
  {
    . . . // Report error and stop further processing
  }
}
```

Even though this code uses `auto_buffer` (Section 16.2) to simplify the hassles attendant with the local buffer memory management, this is still a lot of ugly code to do such a simple thing. The equivalent, using the `winstl::reg_key_sequence` class template that I'll be discussing in this chapter, is demonstrably more straightforward, as shown in Listing 33.8.

Listing 33.8 Shorthand Enumeration of Registry Keys Using `reg_key_sequence`

```
using winstl::reg_key_sequence;

HKEY              key = . . .
reg_key_sequence  keys(key);
for(reg_key_sequence::iterator b = keys.begin(); b != keys.end(); ++b)
{
  ::printf("subkey name=%s\n", (*b).name().c_str());
}
```

Either way, this all seems pretty straightforward, and `winstl::reg_key_sequence` appears to provide nothing more than the usual advantages of succinctness, a common (STL) idiom, exception safety, expressiveness, robustness, and so on. Great though those benefits are, we've seen them enough times in the chapters in this part. What's the biggie in this case?

33.3.1 So What's the Problem?

Because the registry is available to all processes on a given system, it is possible for one process to modify the subkeys or values of a given key while another is midway through an enumeration. As an example, pretend that we're using the code in Listing 33.8 to enumerate the subkeys of *`HKEY_CURRENT_USER\Software\XSTL\Vol1\test\EII\Registry`*, which contains, at the start of our enumeration, the following subkeys:

```
HKEY_CURRENT_USER
  \Software
    \XSTL
      \Vol1
        \test
          \EII
            \Registry
              \Key#1
              \Key#2
              \Key#3
              \Key#4
              \Key#5
```

Let's say our process has just printed out *`Key#3`* and is about to loop around to try the next call to `RegEnumKeyEx()`. If another process deletes subkey *`Key#4`*, the next output from our program will be *`Key#5`*, after which `RegEnumKeyEx()` will return *`ERROR_NO_MORE_ITEMS`* and the enumeration will terminate. The view it had of the registry has been altered by an external agent: We've experienced invalidation by extrathread action. You might think this is no problem.

After all, if we're enumerating the contents of a file system directory, and a file that we didn't know we had is taken away from us while leaving us with a consistent picture of the contents *now*, we don't complain.

That's a fair point, but it's missing half the picture—the ugly half. First, we must remember that the file system enumeration API functions `opendir()` and `FindNextFile()` return the next file as defined by the underlying API or the operating system. This is different from enumeration with the **Registry** API, where the client code maintains an index into the ostensible random access interface. So, instead of imagining an element being deleted, let's imagine that one is added. Specifically, consider that we're at the same point in the enumeration and another process inserts *Key#2.1*. Since the registry orders keys lexicographically, this now becomes the third subkey. The next time our process calls `RegEnumKeyEx()` (with index 3), it will receive the name of the fourth subkey, which is now *Key#3*. The enumeration will yield:

```
subkey name=Key#1
subkey name=Key#2
subkey name=Key#3
subkey name=Key#3
subkey name=Key#4
subkey name=Key#5
```

Obviously, this is not good. Thankfully, the **Registry** API provides a mechanism to help us detect this, in the form of the function `RegNotifyChangeKeyValue()`, declared as follows:

```
LONG RegNotifyChangeKeyValue(
  HKEY    hKey,            // Handle to key to watch
  BOOL    bWatchSubtree,   // Watch subkeys too?
  DWORD   dwNotifyFilter,  // Types of events to watch
  HANDLE  hEvent,          // Synchronization object to signal on change
  BOOL    fAsynchronous,   // Wait for change, or signal the object?
);
```

The semantics of this function depend on what events you wish to monitor, as expressed by the values specified by `dwNotifyFilter`. There are a number of flags to enable you to monitor changes to key values, attributes, security information, and so on. The one we're concerned with in this case is *REG_NOTIFY_CHANGE_NAME*, which monitors whether any subkeys are added or deleted. The function works in both synchronous and asynchronous form. Specifying a nonzero value for the `fAsynchronous` flag and passing the handle of a (currently unsignalled) Windows event synchronization object causes the function to return immediately and the event to be signalled upon the first change that matches the stipulated monitoring. This is the mode used here and in the **WinSTL Registry** library components. Specifying a zero value for `fAsynchronous` causes the function to wait synchronously for a matching change. (I've never heard of any code that uses this function synchronously and can offer no opinion on how useful it might be.) The purpose of `bWatchSubtree` is pretty clear: If it is nonzero, the key *and* its subkeys are monitored; otherwise, changes to the subkeys are ignored. Let's plug this back into the longhand program and see if we can prevent the logical invalidation of our output. Listing 33.9 shows the changes.

Listing 33.9 Changes to the Manual Enumeration to Handle External Changes

```
HKEY                          key   = . . .
DWORD                         index = 0;
stlsoft::auto_buffer<char>    buff(100);
winstl::event                 ev(true, false);

::RegNotifyChangeKeyValue(key
                        , true
                        , REG_NOTIFY_CHANGE_NAME
                        , ev.get()
                        , true);
for(;;)
{
  . . . // All identical to previous version

  if(WAIT_OBJECT_0 == ::WaitForSingleObject(ev.get(), 0))
  {
    printf("External invalidation!\n");
    break;
  }
}
```

Under the same conditions as in the previous execution, the output from this version is:

```
subkey name=Key#1
subkey name=Key#2
subkey name=Key#3
External invalidation!
```

So, we can regain some sense in the program behavior, but this is an onerous burden to place on the user of the **Registry** API. This is particularly so because monitoring is a one-off action: When the event is signalled, the underlying activities invoked by `RegNotifyChangeKeyValue()` cease, and you will need to call it again to reregister! Time for some STL extension, methinks.

(Perversely, subsequent calls on the same key handle ignore the `bWatchSubtree` and `dwNotifyFilter` parameters. If you want to change these characteristics of the monitoring, you will need to open another key handle and attach the monitoring to that. Don't you just love APIs whose abstractions leak worse than a prime minister's principle private secretary?)

33.3.2 The WinSTL Registry Library

The **WinSTL Registry** library provides a set of ***Façade*** classes over the Windows **Registry** API, including several STL collections. Following in the stead of other Windows extensions, compilation in both ANSI/multibyte and Unicode variants is supported, requiring abstraction of the character encoding and the actual `A`/`W` form functions from the **Registry** API. Consequently there are six class templates and three classes, as shown in Table 33.1.

Table 33.1 WinSTL Registry Library Components

Class or Class template	Purpose
`registry_exception`	General exception thrown by the library
`wrong_value_type_exception`	Exception thrown to indicate a conflict between expected and actual value types; derives from `registry_exception`
`access_denied_exception`	Exception thrown when the caller does not have access rights for a requested operation; derives from `registry_exception`
`reg_traits<>`	***Traits*** class that abstracts the `A`/`W` forms of the **Registry** API for the specializing character type
`basic_reg_key<>`	***Façade*** class that represents a specific key in the Registry and provides methods for manipulating subkeys and values
`basic_reg_value<>`	***Façade*** class that represents a specific key value in the registry and provides methods for accessing the contents according to the actual type
`reg_blob<>`	Special class used by `basic_reg_value` that is used to represent the contents of binary values
`basic_reg_key_sequence<>`	STL collection for enumeration of subkeys of a given registry key (either `HKEY` or `basic_reg_key` instance)
`basic_reg_value_sequence<>`	STL collection for enumeration of values of a given registry key (either `HKEY` or `basic_reg_key` instance)

The `basic_reg_key_sequence` and `basic_reg_value_sequence` class templates are very similar in basic structure to the `basic_findfile_sequence` class templates from the **InetSTL** (Section 21.1) and **WinSTL** (Section 20.4) subprojects. Each has a dedicated iterator class—`basic_reg_key_sequence_iterator` and `basic_reg_value_sequence_iterator`—and their value classes are in fact `basic_reg_key` and `basic_reg_value`, respectively. The only break from a full object-oriented model is that `basic_reg_key` does not itself act as a collection of either subkeys or values; you must specifically create an instance of the requisite collection (both of which accept a `basic_reg_key` argument in constructor overloads).

`reg_traits` performs a similar service as `filesystem_traits` (Section 16.3), allowing the other classes to be written in terms of a single syntactic interface, rather than having to worry about the `A`/`W` forms of the Windows API functions.

33.3.3 Handling External Iterator Invalidation

With our ***Façade*** classes, we have two choices for handling the invalidation. One choice is to have an event in the sequence class, and each iterator will check it via a back pointer. In resource terms, this is a more attractive option since the event object is a kernel object, and our instincts tell us that kernel objects are expensive and scarce resources. The alternative is for each iterator instance to have an event. This raises alarms because of the possible profusion of kernel objects, but it's not as bad as it sounds since events will be shared between *related* iterator instances, in much the same way as for the `environment_map` (Chapter 25). Still, we're going to create an event object every time we call `begin()`, which should give us pause at least.

Fortunately, the semantics in this case force us to accept that there's only one valid choice. Consider the code that follows. The problem is clear: The range represented by [`b2`, `e2`) was "created" after the subkey was deleted. At the point of its creation, the key *`Vamoose`* is already gone. But attempting to enumerate the range will still throw an exception.

```
reg_key                      key(HKEY_CURRENT_USER, "SOFTWARE\\ . . ");
reg_key_sequence             keys(key);
reg_key_sequence::iterator  b1;
reg_key_sequence::iterator  e1;
reg_key_sequence::iterator  b2;
reg_key_sequence::iterator  e2;

b1 = keys.begin();
e1 = keys.end();
key.delete_sub_key("Vamoose");
b2 = keys.begin();
e2 = keys.end();
++b2; // Throws an exception!
```

It gets worse. Now consider the case where we catch one of the exceptions. Assuming the event is auto-reset—a wait operation consumes its signalled state—the invalid increment of `b1` will proceed unmolested.

```
b1 = keys.begin();
e1 = keys.end();
key.delete_sub_key("Vamoose");
b2 = keys.begin();
e2 = keys.end();
try
{
  ++b2; // Throws an exception
}
catch(stlsoft::iterator_invalidation&)
{}
++b1; // Doesn't throw!
```

Conversely, if the event is manual reset—it continues to be signalled until explicitly reset via `ResetEvent()`—the following will throw an exception against the poor application programmer's reasonable intent.

```
b1 = keys.begin();
e1 = keys.end();
key.delete_sub_key("Vamoose");
b2 = keys.begin();
e2 = keys.end();
try
{
  ++b1; // Throws an exception, as expected
}
catch(stlsoft::iterator_invalidation&)
{}
++b2; // Throws from here as well. Could be surprising. . . .
```

Thus, rather than being a portal into the subkeys of a given key, a `reg_key_sequence` instance is rather a chimeric snapshot of its state: Values and attributes may change without notification, but any subkey changes invalidate all possible iterators derived from the sequence instance. Clearly, we must associate the monitoring with iterator instances. As mentioned earlier, the costs of this will be amortized between all associated iterator instances by holding the event object in a shared handle, as we've seen with other collections. Hence, only two event objects are created in the following code:

```
reg_key_sequence::iterator  b1 = keys.begin(); // New event object
reg_key_sequence::iterator  b2 = b1; // Shares state
reg_key_sequence::iterator  b3;

b3 = keys.begin(); // New event object
b1 = b3; // Shares state
```

Now we know what we need to do. It's time to look at the implementations.

33.3.4 `winstl::basic_reg_key_sequence`

Listing 33.10 shows the class interface of `basic_reg_key_sequence`. Familiarity with `basic_findfile_sequence` (Chapter 20) will help a lot here. There are five constructors, a destructor, forward and reverse iterator methods, and two size methods.

Listing 33.10 Definition of `basic_reg_key_sequence`

```
// In namespace winstl
template< typename C  // Character type
        , typename T = reg_traits<C>
        , typename A = processheap_allocator<C>
        >
class basic_reg_key_sequence
{
```

```
public: // Member Types
  typedef C                                      char_type;
  typedef T                                      traits_type;
  typedef A                                      allocator_type;
  typedef basic_reg_key_sequence<C, T, A>        class_type;
  typedef basic_reg_key<C, T, A>                 key_type;
  typedef key_type                               value_type;
  typedef typename traits_type::size_type        size_type;
  typedef basic_reg_key_sequence_iterator<C, T, value_type, A>
                                                 iterator;
  typedef key_type&                              reference;
  typedef key_type const&                        const_reference;
  typedef HKEY                                   hkey_type;
  typedef ptrdiff_t                              difference_type;
  typedef std::reverse_iterator<iterator>        reverse_iterator;
public: // Construction
  basic_reg_key_sequence( hkey_type          hkey
                        , char_type const*   sub_key_name
                        , REGSAM             accessMask = KEY_READ);
  basic_reg_key_sequence( hkey_type          hkey
                        , char_type const*   sub_key_name
                        , REGSAM             accessMask
                        , bool           bMonitorExternalInvalidation);
  explicit basic_reg_key_sequence(key_type const& key);
  basic_reg_key_sequence( key_type const& key
                       , REGSAM               accessMask);
  basic_reg_key_sequence( key_type const& key
                       , REGSAM               accessMask
                       , bool            bMonitorExternalInvalidation);
  ~basic_reg_key_sequence() throw();
public: // Iteration
  iterator          begin();
  iterator          end();
  reverse_iterator  rbegin();
  reverse_iterator  rend();
public: // Size
  size_type   current_size() const;
  bool        empty() const;
private: // Member Variables
  hkey_type     m_hkey;
  const REGSAM  m_accessMask;
  const bool    m_bMonitorExternalInvalidation;
private: // Not to be implemented
  basic_reg_key_sequence(class_type const&);
  class_type& operator =(class_type const&);
};
```

```
typedef basic_reg_key_sequence<char>       reg_key_sequence_a;
typedef basic_reg_key_sequence<wchar_t>    reg_key_sequence_w;
typedef basic_reg_key_sequence<TCHAR>      reg_key_sequence;
```

There are five constructors in order to facilitate sensible semantics between the `accessMask` and `bMonitorExternalInvalidation` parameters. In the cases where there is no `bMonitorExternalInvalidation` parameter, the presence/absence of the `KEY_NOTIFY` flag in the `accessMask` determines how the `m_bMonitorExternalInvalidation` member is initialized. Conversely, when the `bMonitorExternalInvalidation` parameter is present, it overrides the presence/absence of the `KEY_NOTIFY` flag in the `accessMask`. Clearly, these overloads support sequences whose iterators monitor changes, as well as those that do not. You might elect to use the latter if you do not care about any such changes or if you have exclusive access to the key (e.g., via a key's security descriptor).

The provision of `current_size()`, rather than `size()`, emphasizes the fact that it is a transient value and may not reflect the number of elements enumerated in [`begin()`, `end()`) or [`rbegin()`, `rend()`). The remainder of the methods are pretty straightforward in semantics. Listing 33.11 shows the implementation of the `begin()` and `end()` methods.

Listing 33.11 Definition of the Iteration Methods

```
template <. . .>
iterator basic_reg_key_sequence<. . .>::begin() const
{
  size_type   cchKeyName  = 0;
  size_type   numEntries  = 0;
  result_type res = traits_type::reg_query_info(m_hkey, NULL, NULL
                             , &numEntries, &cchKeyName, NULL, NULL
                             , NULL, NULL, NULL, NULL);
  if(ERROR_SUCCESS != res)
  {
    . . . // Throw appropriate exception
  }
  else
  {
    if(0 != numEntries)
    {
      registry_util::shared_handle* handle= create_shared_handle_(res);
      ref_ptr<registry_util::shared_handle> ref(handle, false);
      auto_buffer<char_type>                buffer(++cchKeyName);
      for(; !buffer.empty(); buffer.resize(2 * buffer.size()))
      {
        cchKeyName = buffer.size();
        res = traits_type::reg_enum_key(m_hkey, 0, &buffer[0]
                                       , &cchKeyName);
        if(ERROR_MORE_DATA == res)
        {
          continue; // "Let's go round again"
        }
        else if(ERROR_SUCCESS != res)
        {
          . . . // Throw appropriate exception
        }
        else
        {
          handle->test_reset_and_throw();
          return iterator(handle, buffer.data(), cchKeyName, 0
                        , m_accessMask);
        }
      }
    }
  }
  return end();
}
template <. . .>
iterator basic_reg_key_sequence<C, T, A>::end() const
{
  result_type                           res;
  registry_util::shared_handle* handle = create_shared_handle_(res);
  ref_ptr<registry_util::shared_handle> ref(handle, false);
  return iterator(handle, NULL,0, iterator::sentinel_(), m_accessMask);
}
```

There's a lot of code here, but much of it is for error handling. Since we're dealing with resources straight from an API, this is to be expected. The functionality is actually pretty straightforward. The first call, to `traits_type::reg_query_info()`, results in an invocation of the Windows **Registry** API function `RegQueryInfoKeyA/W()`, in order to obtain the number of subkeys and the current maximum subkey name length. The former is used to determine whether there are any entries to retrieve; if not, `end()` is returned. The latter is used as a basis for the size of the buffer used to retrieve the zeroth subkey's name, using an `auto_buffer`.

If one or more entries are indicated, the shared handle is created before any retrieval, via `create_shared_handle_()` (Listing 33.12). This function duplicates the key handle and then calls `shared_handle::create_shared_handle()`, which is a factory function that creates an instance of `shared_handle` or `monitored_shared_handle`, according to the `m_bMonitorExternalInvalidation` argument. The utility class template `scoped_handle` guarantees that the duplicated key handle is released in case any exception is thrown by `create_shared_handle()`; `sh.detach()` releases the key, now owned by the shared handle instance, after it has been fully constructed, before returning to the caller.

Listing 33.12 Definition of the `create_shared_handle_()` Worker Method

```
template <. . .>
registry_util::shared_handle*
 basic_reg_key_sequence<C, T, A>::
                                   create_shared_handle_(result_type& res)
{
    hkey_type hkey2 = traits_type::key_dup(m_hkey, m_accessMask, &res);
    if(NULL == hkey2)
    {
      . . . // Throw exception
    }
    else
    {
      scoped_handle<HKEY>            sh(hkey2, ::RegCloseKey);
      registry_util::shared_handle* handle =
                   registry_util::create_shared_handle(hkey2
                                          , m_bMonitorExternalInvalidation
                                          , REG_NOTIFY_CHANGE_NAME);
      sh.detach();
      return handle;
    }
}
```

Back in the caller, `basic_reg_key_sequence::begin()` (Listing 33.11), another utility class template, `ref_ptr`, swallows the shared handle instance, so that it will not be lost if an exception is thrown. The remainder of the function obtains the full name of the zeroth element; the resizing in response to *`ERROR_MORE_DATA`* is necessary in case a new key, with a name longer than `cchKeyName` and lexicographically less than all others, is added between the call to `reg_query_info()` and `reg_enum_key()`. Once the full name is obtained, the iterator constructor is invoked, passing the handle, the name (pointer and length), the index (`0`), and the access mask.

The `end()` method (Listing 33.11) creates a shared handle instance, passing it to the iterator constructor with the special index returned by the iterator's `sentinel_()` method. Because the iterators are bidirectional, it must be possible to meaningfully decrement the endpoint iterator. Hence, it must be given a shared handle, just as that returned by `begin()` must.

Now that we've seen the salient parts of the sequence class, we need to look in more detail at the iterator class, `basic_reg_key_sequence_iterator`. Listing 33.13 shows the class interface.

Listing 33.13 Definition of `basic_reg_key_sequence_iterator`

```
// namespace winstl
template< typename C
        , typename T
        , typename V
        , typename A
        >
class basic_reg_key_sequence_iterator
  : public std::iterator< std::bidirectional_iterator_tag
                        , V, ptrdiff_t
                        , void, V   // BVT
                        >
{
public: // Member Types
  typedef C                                                   char_type;
  typedef T                                                   traits_type;
  typedef V                                                   value_type;
  typedef A                                                   allocator_type;
  typedef basic_reg_key_sequence_iterator<C, T, V, A> class_type;
  typedef typename traits_type::size_type             size_type;
  . . . // And difference_type, string_type, index_type, hkey_type
private:
  typedef typename traits_type::result_type             result_type;
private: // Construction
  friend class basic_reg_key_sequence<C, T, A>;
  basic_reg_key_sequence_iterator(
                            registry_util::shared_handle* handle
                          , char_type const*              name
                          , size_type                     cchName
                          , index_type                    index
                          , REGSAM                        accessMask)
    : m_handle(handle)
    , m_index(index)
    , m_name(name, cchName)
    , m_accessMask(accessMask)
  {
    WINSTL_ASSERT(NULL != m_handle);
    m_handle->test_reset_and_throw();
    m_handle->AddRef();
  }
public:
  basic_reg_key_sequence_iterator();
  basic_reg_key_sequence_iterator(class_type const& rhs);
  ~basic_reg_key_sequence_iterator() throw();
  class_type& operator =(class_type const& rhs);
public: // Accessors
  const string_type&  get_key_name() const; // Returns m_name
public: // Bidirectional Iterator Methods
  class_type& operator ++();
  class_type& operator --0();
  const class_type operator ++(int);
  const class_type operator --(int);
  const value_type operator *() const;
  bool equal(class_type const& rhs) const;
  bool operator ==(class_type const& rhs) const;
  bool operator !=(class_type const& rhs) const;
private: // Implementation
  static index_type sentinel_();  // Returns 0x7FFFFFFF
private: // Member Variables
  registry_util::shared_handle* m_handle;
  index_type                    m_index;
  string_type                   m_name;
  REGSAM                        m_accessMask;
};
```

The class is a bidirectional iterator. The reason it's not random access is prosaic: I couldn't face all the complexities in making it so. It's complicated enough as it is. The iterator supports the *by-value temporary* element reference category. In support of this, the iterator contains the name of the enumerated subkey with which it can invoke `RegOpenKeyExA/W()` to create an instance of `basic_reg_key` when dereferenced, along with the access mask. It contains an index to represent iteration state and to facilitate comparison between iterator instances. The shared handle (`m_handle`), shared by reference counting between iterator instances, contains both the registry key handle and, optionally, the event. We'll see how this is handled shortly, when we look at the shared handle classes, but first I want to show you the implementations of the preincrement and predecrement operators. The preincrement operator is shown in Listing 33.14.

Listing 33.14 Definition of the Preincrement Operator

```
template <. . .>
class_type& basic_reg_key_sequence_iterator<C, T, V, A>::operator ++()
{
  WINSTL_MESSAGE_ASSERT("Attempting to increment an invalid iterator!"
                       , sentinel_() != m_index);
  size_type   cchKeyName  = 0;
  result_type res = traits_type::reg_query_info(m_handle->m_hkey, NULL
                                  , NULL, NULL, &cchKeyName, NULL, NULL
                                  , NULL, NULL, NULL, NULL);
  if(ERROR_SUCCESS != res)
  {
    . . . // Throw appropriate exception
  }
  else
  {
    auto_buffer<char_type>  buffer(++cchKeyName);
    for(; !buffer.empty(); buffer.resize(2 * buffer.size()))
```

```
      {
        cchKeyName = buffer.size();
        res = traits_type::reg_enum_key(m_handle->m_hkey, 1 + m_index
                                        , &buffer[0], &cchKeyName);
        if(ERROR_MORE_DATA == res)
        {
          continue; // "Let's go round again"
        }
        else if(ERROR_NO_MORE_ITEMS == res)
        {
          m_index = sentinel_(); // Become the endpoint iterator
          break;
        }
        else if(ERROR_SUCCESS != res)
        {
          . . . // Throw appropriate exception
        }
        m_name.assign(buffer.data(), cchKeyName);
        ++m_index;
        break;
      }
    }
    m_handle->test_reset_and_throw();
    return *this;
}
```

Much of this should be familiar to us, having seen the implementation of `begin()`. But there are two important differences. First, when the last item has been reached, the index is set to the value obtained from `sentinel_()` (which is `0x7FFFFFFF`). Second, the `test_reset_and_throw()` method of the shared handle instance is invoked. Ostensibly this tests for iterator invalidation, but the precise behavior depends on which shared handle class was returned by the factory.

Let's look at the definition of the shared handle classes. The nonmonitoring class is called `shared_handle` , defined in the subnamespace `registry_util` (Listing 33.15).

Listing 33.15 Definition of `shared_handle`

```
// In namespace winstl::registry_util
struct shared_handle
{
public: // Member Types
  typedef shared_handle   class_type;
  typedef HKEY            handle_type;
public: // Member Variables
  handle_type m_hkey;
private:
  sint32_t    m_refCount;
public: // Construction
```

```
  explicit shared_handle(handle_type hkey)
    : m_hkey(hkey)
    , m_refCount(1)
  {}
protected:
  shared_handle(handle_type hkey, sint32_t refCount)
    : m_hkey(hkey)
    , m_refCount(refCount)
  {}
protected:
  virtual ~shared_handle() throw()
  {
    WINSTL_MESSAGE_ASSERT("Shared search handle being destroyed with
outstanding references!", 0 == m_refCount);
    if(NULL != m_hkey)
    {
      ::RegCloseKey(m_hkey);
    }
  }
public: // Operations
  sint32_t AddRef()
  {
    return ++m_refCount;
  }
  sint32_t Release()
  {
    sint32_t rc = --m_refCount;
    if(0 == rc)
    {
      delete this;
    }
    return rc;
  }
  virtual void test_reset_and_throw()
  {}
private: // Not to be implemented
  . . . // Proscribe copy constructor and copy assign operator
};
```

This class is quite similar to the other `shared_handle` classes we've seen in earlier parts of the book (Sections 19.3.7 and 20.5.3), but in this case the destructor is virtual, and a virtual method `test_reset_and_throw()` is defined. For this class, `test_reset_and_throw()` does nothing, corresponding with the fact that the `shared_handle` class does not monitor changes.

The `monitored_shared_handle` class (Listing 33.16), derived from `shared_handle`, contains additional members for the monitoring: the `event` class instance and the event type. For monitoring changes to subkeys, the event type is *`REG_NOTIFY_CHANGE_NAME`*.

Listing 33.16 Definition of `monitored_shared_handle`

```
// In namespace winstl::registry_util
struct monitored_shared_handle
  : public shared_handle
{
public: // Member Types
  typedef shared_handle           parent_class_type;
  typedef monitored_shared_handle class_type;
public: // Construction
  monitored_shared_handle(handle_type hkey, int eventType)
    : parent_class_type(hkey, 0)
    , m_monitor(true, false)
    , m_eventType(eventType)
  {
    set();
    AddRef();
  }
private: // Operations
  virtual void test_reset_and_throw()
  {
    // 1. Test
    if(WAIT_OBJECT_0 == ::WaitForSingleObject(m_monitor.get(), 0))
    {
      // 2. Reset
      set();
      // 3. Throw
      throw stlsoft::external_iterator_invalidation("registry contents
changed");
    }
  }
  void set()
  {
    try
    {
      dl_call<LONG>("ADVAPI32.DLL", "S:RegNotifyChangeKeyValue", m_hkey
                  , false, (int)m_eventType, m_monitor.get(), true);
    }
    catch(missing_entry_point_exception&)
    {
      if( 0 != (::GetVersion() & 0x80000000) &&
        LOBYTE(LOWORD(GetVersion())) == 4 &&
        HIBYTE(LOWORD(GetVersion())) < 10)
      {
        ; // Do nothing, and absorb it
      }
      else
      {
        throw;
      }
    }
  }
private: // Member Variables
  const int   m_eventType;
  event       m_monitor;  // Event that will monitor changes to the API
private: // Not to be implemented
  . . . // Proscribe copy constructor and copy assign operator
};
```

The `test_reset_and_throw()` takes serious action in this case. First, the event instance is checked via the Windows system call `WaitForSingleObject()`. If it is signalled, it is reset, to facilitate the case where a user may catch the external invalidation and wish to proceed. Finally, the exception is thrown. The `set()` method, also called in the constructor to commence monitoring, is implemented using `dl_call()` because the `RegNotifyChangeKeyValue()` function is not available on Windows 95. The `try-catch` block quenches this exception for Windows 95, for which the **WinSTL Registry** library does not offer monitoring of external iterator invalidation.

Listing 33.17 shows the implementation of the factory function `registry_util::create_shared_handle()`, which constructs the appropriate shared handle instance.

Listing 33.17 Definition of the `create_shared_handle()` Utility Function

```
// In namespace winstl::registry_util
static shared_handle* create_shared_handle(HKEY hkey
                    , bool bMonitorExternalInvalidation, int eventType)
{
  if(bMonitorExternalInvalidation)
  {
    return new monitored_shared_handle(hkey, eventType);
  }
  else
  {
    return new shared_handle(hkey);
  }
}
```

The final part of the puzzle is the iterator's predecrement operator, shown in Listing 33.18.

Listing 33.18 Definition of the Predecrement Operator

```
template <. . .>
class_type& basic_reg_key_sequence_iterator<. . .>::operator --()
{
  WINSTL_MESSAGE_ASSERT("Attempting to decrement an invalid iterator"
                      , NULL != m_handle);
  size_type   cchKeyName = 0;
  size_type   numEntries = 0;
  result_type res         = traits_type::reg_query_info(m_handle->m_hkey
                              , NULL, NULL, &numEntries, &cchKeyName
                              , NULL, NULL, NULL, NULL, NULL, NULL);
  if(ERROR_SUCCESS != res)
  {
    . . . // Throw appropriate exception
  }
  else
  {
    auto_buffer<char_type>  buffer(++cchKeyName);
    DWORD                   index;
    if(m_index == sentinel_())
    {
      index = numEntries - 1;
    }
    else
    {
      index = m_index - 1;
    }
    for(; !buffer.empty(); buffer.resize(2 * buffer.size()))
    {
      cchKeyName  =   buffer.size();
      res = traits_type::reg_enum_key(m_handle->m_hkey, index
                                      , &buffer[0], &cchKeyName);
      if(ERROR_MORE_DATA == res)
      {
        continue; // "Let's go round again"
      }
      else if(ERROR_SUCCESS != res)
      {
        . . . // Throw appropriate exception
      }
      m_name.assign(buffer.data(), cchKeyName);
      m_index = index;
      break;
    }
  }
  m_handle->test_reset_and_throw();
  return *this;
}
```

This is very similar to the preincrement operator, with the exception that the index is derived either by decrementing the current index or, if currently at the endpoint (as denoted by `m_index == sentinel_()`), by setting it to one less than the total number of elements.

As it now stands, this implementation will catch any post-`begin()` invalidation the next time the iterator is incremented or decremented. But we could still find ourselves dereferencing an iterator that has been invalidated, one whose invalidation will not become apparent until the next increment or decrement. So the implementation is completed by an invalidation test in the dereference operator, as shown in Listing 33.19.

Listing 33.19 Definition of the Dereference Operator

```
template <. . .>
value_type
 basic_reg_key_sequence_iterator<C, T, V, A>::operator *() const
{
  WINSTL_MESSAGE_ASSERT("Attempting to dereference an invalid iterator"
                      , NULL != m_handle);
  m_handle->test_reset_and_throw();
  return value_type(m_handle->m_hkey, m_name, m_accessMask);
}
```

33.4 Summary

All in all, you can see that the issue of external iterator invalidation is quite involved, and the classes that handle it represent a nontrivial bit of kit; I won't insult your intelligence by attempting to pretend otherwise. There are three forms of invalidation: invalidation by intrathread API side effects, invalidation by intrathread application side effects, and invalidation by extrathread action. We looked in detail at how these can be handled in STL extension classes and demonstrated that it is possible to have well-defined implementations of sequences and their iterators in the context of external invalidation given the availability of mechanisms with which it can be detected. Importantly, we have applied the *Principle of Most Surprise* where it was sorely needed.

33.5 On the CD

The CD contains a section describing the situation of external iterator invalidation in popular XML libraries.

PART THREE

Iterators

This part of the book covers two main topics: output iterators (Section 1.3.2) and iterator adaptors.

The first chapter in this part, Chapter 34, looks at an enhancement to `std::ostream_iterator` that provides for prefixes along with suffixes and is completely backward-compatible. This is then immediately followed by an intermezzo, Chapter 35, in which a common strategy for implementing output iterators is shown to result in fatuous code, and a new pattern that can be readily applied to avoid such code is described.

Next, Chapter 36 presents our first case of iterator adaptation, that of transforming the values and/or type of the elements in a range. It demonstrates that a transforming iterator must bear the *by-value temporary* element reference category (Section 3.3.5) to support the *random access iterator* category (Section 1.3.5), or the *transient* element reference category (Section 3.3.4) to support the *bidirectional iterator* category (Section 1.3.4) or lower. The chapter also shows that using a transforming iterator can yield surprisingly significant performance advantages. Once again, an intermezzo, Chapter 37 in this case, discusses a related issue, that of the imperfect naming schemes for creator functions.

The following chapter, Chapter 38, describes an iterator adaptor that allows the elements in a range of structures to be manipulated in terms of just one member of the structure. Although the definition of the iterator adaptor itself is entirely straightforward, there's a long and sinuous tale of the definition of creator functions sufficiently capable of handling the permutations of iterator category, mutability, and structure element mutability. Again, performance advantages are described and quantified.

The theme returns to output iterators for Chapters 39 and 40, which describe components that facilitate the building of strings, either raw character buffers or string class instances, in combination with source sequences and standard algorithms. In both cases, techniques for maximizing flexibility to enable compatibility with a wide variety of source and destination string types are examined and tradeoffs discussed.

The next chapter, Chapter 41, doesn't involve an iterator at all! Rather, it discusses in detail the implementation of the `adapted_iterator_traits` component, a traits class that provides facilities over and above those of `std::iterator_traits`, facilities that are a necessity for defining STL extension iterators (and collections, for that matter). This component discriminates a number of characteristics of the iterator types used to specialize it, including mutability and element reference category, facilitating simple and straightforward implementations of iterator adaptors.

Use of `adapted_iterator_traits` is illustrated in Chapter 42, which describes a component used to filter out elements in a range. It discusses the issues involved in supporting different iterator categories of the emergent range and, importantly, demonstrates the restrictions imposed on client code that uses filtering iterators. The CD contains another chapter, Indexed Iteration, that further illustrates the use of `adapted_iterator_traits`.

The final chapter in the book, Chapter 43, demonstrates how iterator adaptors can be used in concert and illustrates how this can be achieved in surprisingly simple ways.

CHAPTER 34

An Enhanced `ostream_iterator`

I have benefited greatly from criticism, and at no time have I suffered a lack thereof.

—Winston Churchill

I'm so gorgeous, there's a six-month waiting list for birds to suddenly appear, every time I am near!

—Cat, *Red Dwarf*

34.1 Introduction

Do you ever find yourself wanting to output the contents of a sequence where the elements are to be indented, perhaps by a tab space (`'\t'`), as in the following example?

```
Header Files:
    H:\freelibs\b64\current\include\b64\b64.h
    H:\freelibs\b64\current\include\b64\cpp\b64.hpp
Implementation Files:
    H:\freelibs\b64\current\src\b64.c
    H:\freelibs\b64\current\test\C\C.c
    H:\freelibs\b64\current\test\Cpp\Cpp.cpp
```

Using `std::ostream_iterator`, this is disproportionately difficult and inelegant to achieve. Consider that we're searching for source files under the current directory, using the **recls/STL** library (itself an adaptation of a collection in what should now be, after reading Part II, characteristic STL extension style). **recls/STL** provides the `recls::stl::search_sequence` collection (a typedef of `recls::stl::basic_search_sequence<char>`), to which we pass the search directory, pattern, and flags. We can use this in combination with `std::copy()` and `std::ostream_iterator`, as shown in Listing 34.1, to achieve the desired output.

Listing 34.1 Formatting Output Using `std::ostream_iterator`

```
typedef recls::stl::search_sequence  srchseq_t;
using recls::RECLS_F_RECURSIVE;

srchseq_t headers(".", "*.h|*.hpp", RECLS_F_RECURSIVE);
srchseq_t impls(".", "*.c|*.cpp", RECLS_F_RECURSIVE);

std::cout << "Header Files:" << std::endl << "\t";
```

```
8  std::copy(headers.begin(), headers.end()
9   , std::ostream_iterator<srchseq_t::value_type>(std::cout, "\n\t"));
10 std::cout << "\r";
11
12 std::cout << "Implementation Files:" << std::endl << "\t";
13 std::copy(impls.begin(), impls.end()
14  , std::ostream_iterator<srchseq_t::value_type>(std::cout, "\n\t"));
15 std::cout << "\r";
```

Obviously, there's a degree of mess here in that the formatting we seek to apply on lines 8–9 and 13–14 leaks out into lines 7, 10, 12. and 15. I'm certain you can imagine how, in more complex cases, this can lead to convoluted and fragile code, something that would just not happen were `std::ostream_iterator` a tiny bit smarter. This chapter describes how `std::ostream_iterator` can be enhanced in a simple but crucial way, in the form of `stlsoft::ostream_iterator`.

Before we look in depth at the problem and the simple solution, let's see that solution in action (Listing 34.2).

Listing 34.2 Formatting Output Using `stlsoft::ostream_iterator`

```
typedef recls::stl::search_sequence  srchseq_t;
using recls::RECLS_F_RECURSIVE;

srchseq_t headers(".", "*.h|*.hpp", RECLS_F_RECURSIVE);
srchseq_t impls(".", "*.c|*.cpp", RECLS_F_RECURSIVE);

std::cout << "Header Files:" << std::endl;
std::copy(headers.begin(), headers.end()
        , stlsoft::ostream_iterator<srchseq_t::value_type>(std::cout
                                                     , "\t", "\n"));

std::cout << "Implementation Files:" << std::endl;
std::copy(impls.begin(), impls.end()
        , stlsoft::ostream_iterator<srchseq_t::value_type>(std::cout
                                                     , "\t", "\n"));
```

Now the formatting is entirely located where it should be, in the invocation of the iterator's constructor. Naturally, this component can be used easily in formatted output that employs different levels of indentation.

(If we wanted to be especially pious with regards to *DRY SPOT*, we might declare a single instance of `stlsoft::ostream_iterator` with the required prefix and suffix and pass it to the two invocations of `std::copy()`. But that's not as clear-cut as you might think, as we'll shortly see.)

34.2 std::ostream_iterator

The standard (C++-03: 24.5.2) defines the interface of the `std::ostream_iterator` class template, as shown in Listing 34.3.

Listing 34.3 Definition of `stlsoft::ostream_iterator`

```
// In namespace std
template< typename V                          // The type to be inserted
        , typename C = char                   // Char encoding of stream
        , typename T = std::char_traits<C>  // Traits type
        >
class ostream_iterator
  : public iterator<output_iterator_tag, void, void, void, void>
{
public: // Member Types
  typedef C                                               char_type;
  typedef T                                               traits_type;
  typedef std::basic_ostream<char_type, traits_type>  ostream_type;
  typedef ostream_iterator<V, C, T>                       class_type;
public: // Construction
  explicit ostream_iterator(ostream_type& os);
  ostream_iterator(ostream_type& os, char_type const* delim);
  ostream_iterator(class_type const& rhs);
  ~ostream_iterator() throw();
public: // Assignment
  class_type& operator =(V const& value);
public: // Output Iterator Methods
  class_type& operator *();
  class_type& operator ++();
  class_type operator ++(int);
private:
  . . .
};
```

This is a classic output iterator, whereby each instance remembers the stream and the (optional) delimiter from which it is constructed and uses them to effect formatted output when a value is assigned to it. Implementations typically maintain a pointer to the stream, which we'll call `m_stm`, and a copy of the delimiter pointer (i.e., of type `char_type const*`), which we'll call `m_delim`. Listing 34.4 shows the implementation of the assignment operator.

Listing 34.4 Definition of the Assignment Operator

```
template<typename V, typename C, typename T>
ostream_iterator<V, C, T>&
 ostream_iterator<V, C, T>::operator =(V const& value)
{
  *m_stm << value;
  if(NULL != m_delim)
  {
    *m_stm << m_delim;
  }
  return *this;
}
```

It's a really simple, clever idea with just one flaw: the lack of ambition demonstrated in the previous section.

34.2.1 `void` Difference Type

Note in Listing 34.3 that `ostream_iterator` uses the standard manner of instantiating the `std::iterator` type generator (Section 12.2) class template, by which all types except the iterator category are `void`. These are defined as `void` to help prevent the use of output iterators in contexts where their behavior would be undefined. For example, the standard (C++-03: 24.3.4) defines the `std::distance()` algorithm's return type in terms of the given iterator's `difference_type`, as in the following:

```
template <typename I>
typename std::iterator_traits<I>::difference_type
 distance(I from, I to);
```

Since the evaluation of such distance for an output iterator is not meaningful, output iterators should define their `difference_type` (usually via `std::iterator`, as shown in Listing 34.3) to be `void`, which will precipitate a compilation error if a user tries to apply `std::distance()` to such types.

Tip: Define member types as `void` to (help) proscribe unsupported operations.

34.3 `stlsoft::ostream_iterator`

The **STLSoft** libraries contain a very modest enhancement to `std::ostream_iterator`, imaginatively called `stlsoft::ostream_iterator`. Its definition is shown in Listing 34.5.

Listing 34.5 Definition of `stlsoft::ostream_iterator`

```
// In namespace stlsoft
template< typename V
        , typename C = char
        , typename T = std::char_traits<C>
        , typename S = std::basic_string<C, T>
        >
```

```
class ostream_iterator
  : public std::iterator<std::output_iterator_tag
                        , void, void, void, void>
{
public: // Member Types
  typedef V                                           assigned_type;
  typedef C                                           char_type;
  typedef T                                           traits_type;
  typedef S                                           string_type;
  typedef std::basic_ostream<char_type, traits_type>  ostream_type;
  typedef ostream_iterator<V, C, T, S>                class_type;
public: // Construction
  explicit ostream_iterator(ostream_type& os)
    : m_stm(&os)
    , m_prefix()
    , m_suffix()
  {}
  template <typename S1>
  explicit ostream_iterator(ostream_type& os, S1 const& suffix)
    : m_stm(&os)
    , m_prefix()
    , m_suffix(stlsoft::c_str_ptr(suffix))
  {}
  template< typename S1
          , typename S2
          >
  ostream_iterator(ostream_type& os, S1 const& prefix, S2 const& suffix)
    : m_stm(&os)
    , m_prefix(stlsoft::c_str_ptr(prefix))
    , m_suffix(stlsoft::c_str_ptr(suffix))
  {}
  ostream_iterator(class_type const& rhs)
    : m_stm(rhs.m_stm)
    , m_prefix(rhs.m_prefix)
    , m_suffix(rhs.m_suffix)
  {}
  ~ostream_iterator() throw()
  {}
public: // Assignment
  class_type& operator =(assigned_type const& value)
  {
    *m_stm << m_prefix << value << m_suffix;
    return *this;
  }
public: // Output Iterator Methods
  class_type& operator *()
  {
    return *this;
  }
  class_type& operator ++()
  {
    return *this;
  }
  class_type operator ++(int)
  {
    return *this;
  }
private: // Member Variables
  ostream_type* m_stm;
  string_type   m_prefix;
  string_type   m_suffix;
};
```

The main difference is the sole functional enhancement: separation of the delimiter into a prefix and a suffix. For full compatibility with `std::ostream_iterator` semantics, the second, two-parameter constructor specifies the suffix, not the prefix, since `std::ostream_iterator` inserts the delimiter into the stream after the value. The pros and cons of this are discussed in Section 34.3.4.

That's it for the interface. We will now examine the several implementation differences.

34.3.1 Shims, Naturally

The most obvious implementation difference is the use of string access shims (Section 9.3.1). These afford all the usual flexibility to work with a wide variety of string or string-representable types. Indeed, all of the following parameterizations of the iterator are well formed:

```
std::string                     prefix("prefix");
stlsoft::simple_string          suffix("suffix");

stlsoft::ostream_iterator<int>  osi1(std::cout);
stlsoft::ostream_iterator<int>  osi2(std::cout, "suffix");
stlsoft::ostream_iterator<int>  osi3(std::cout, "prefix", "suffix");
stlsoft::ostream_iterator<int>  osi4(std::cout, suffix);
stlsoft::ostream_iterator<int>  osi5(std::cout, prefix, "");
stlsoft::ostream_iterator<int>  osi6(std::cout, prefix, "suffix");
stlsoft::ostream_iterator<int>  osi7(std::cout, "prefix", suffix);
stlsoft::ostream_iterator<int>  osi8(std::cout, prefix, suffix);
```

34.3.2 Safe Semantics

The standard does not prescribe how `std::ostream_iterator` is to be implemented, which is usually perfectly reasonable. However, in this case there is no stipulation as to whether the iterator instance should take a copy of the delimiter contents or merely, as most implementations do, copy the pointer. As long as `ostream_iterator` is used in its familiar context, as a temporary passed to an algorithm, this is irrelevant. However, it's not hard to make a mess:

```
std::string                 delim("\n");
std::ostream_iterator<int>  osi(std::cout, delim.c_str());
std::vector<int>            ints(10);

delim = "something else";

std::copy(ints.begin(), ints.end(), osi); // Undefined behavior!!
```

The issue can be compounded when using string access shims in the constructor to make it more generic. Consider what would happen if `m_prefix` and `m_suffix` were of type `char_type const*`, rather than of type `string_type`. It would be possible to use the iterator class template with any type whose shim function returns a temporary, as in the following:

```
std::vector<int>  ints(10);
VARIANT           pre = . . .
VARIANT           suf = . . .

std::copy(ints.begin(), ints.end()
        , stlsoft::ostream_iterator<int>(std::cout, pre, suf)); // Boom!
```

This is not something obvious even to the trained eye. The problem is that the language requires that temporaries exist for the lifetime of their *enclosing* full expression (C++-03: 12.2;3). Let's look again at the requisite lines from the implementation, highlighting the shim invocations:

```
template<typename S1, typename S2>
ostream_iterator(ostream_type& os, S1 const& prefix, S2 const& suffix)
  : m_stm(&os)
  , m_prefix(stlsoft::c_str_ptr(prefix))
  , m_suffix(stlsoft::c_str_ptr(suffix))
{}
```

By the time the constructor completes, the temporary instances of the conversion class returned by the `c_str_ptr(VARIANT const&)` overload invocations have been created, had their implicit conversion operator called, and been destroyed. The `m_prefix` and `m_suffix` pointers would be left holding onto garbage. When these pointers are used within the `std::copy()` statement, it's all over, Red Rover!

This is a problem with wide potential, but thankfully we can avoid it by adhering to one simple rule.

Rule: If your constructor uses conversion (or access) shims, you must not hold onto the results of the shim functions in pointer (or reference) member variables.

This rule offers only three options. First, we could use (but not hold) the result pointer within the constructor body, as we did in the case of `unixstl::glob_sequence` (Section 17.3.5). Second, we could copy the result into a string member variable, as we did in the case of `unixstl::readdir_sequence` (Section 19.3.2).

The final option is to eschew the use of string access shims entirely and stick with C-style string pointers as constructor parameters. But this pushes all responsibility for conversion out to the application code, thereby violating the *Principle of Composition* (as far too many C++ libraries are wont to do).

Given that the **IOStreams** are anything but lightning-quick themselves, the prudent choice here is to err on the side of flexibility and safety. Thus `stlsoft::ostream_iterator` stores copies of the prefix and suffix arguments in member variables of `string_type`. The nice side effect of this is that the assignment operator implementation becomes extremely simple, just a single statement:

```
*m_stm << m_prefix << value << m_suffix;
```

Tip: Don't emulate undefined vulnerabilities in the standard without careful consideration.

34.3.3 `std::ostream_iterator` Compatibility

The first of the two constructors provides full compatibility with `std::ostream_iterator`. The second, three-parameter constructor provides the additional functionality. To define a prefix-only iterator is straightforward: Just specify the empty string (`""`) as the third parameter in the three-parameter constructor:

```
std::copy(impls.begin(), impls.end()
       , stlsoft::ostream_iterator<srchseq_t::value_type>(std::cout
                                                          , "\t", ""));
```

34.3.4 A Clash of Design Principles?

The constructors of `stlsoft::ostream_iterator` break an important guideline of consistency between overloads and default arguments. The guideline requires that overloads should behave as if they were implemented as one constructor with a number of default arguments. Additional arguments, which refine the behavior/state requested of the function, are stacked on the end.

Guideline: An overload that provides additional parameters should not change the sequence of parameters with respect to the overridden method.

However, the third overloaded constructor of `stlsoft::ostream_iterator` specifies its refinement in the *middle* of the arguments. This is a consequence of applying the *Principle of Least Surprise*: A user will expect to specify a prefix before a suffix. Doing it the other way would result in client code such as the following:

```
std::cout << "Header Files:" << std::endl;
std::copy(headers.begin(), headers.end()
        , stlsoft::ostream_iterator<srchseq_t::value_type>(std::cout
                                                          , "\n", "\t"));
```

This is actually a conflict between levels of discoverability. When viewing the class (template) in isolation, a second overload of `(..., suffix, prefix)` is probably more discoverable. But when viewed in action, the overload of `(..., prefix, suffix)` is definitely more discoverable. Given that, I feel that the result is worth transgressing the guideline and have opted for the latter. But it's an equivocal point, to be sure. You may see it differently.

34.4 Defining Stream Insertion Operators

You may wonder why this code works as well as it does, specifically why `recls::stl::basic_search_sequence<>::value_type` is compatible with `ostream_iterator`. The reason is that `ostream_iterator` can work with any type for which a stream insertion operator is defined.

One way we could have implemented this would be as shown in Listing 34.6. Note that `recls::stl::basic_search_sequence<>::value_type` is actually the class template `recls::stl::basic_search_sequence_value_type<>`, whose name is another win for succinctness. (Thankfully, you never need to specify it in client code.)

Listing 34.6 Stream Insertion Operators

```
namespace recls::stl
{
  std::basic_ostream<char>&
   operator <<(std::basic_ostream<char>&                        stm
             , basic_search_sequence_value_type<char, . . .>& v)
  {
    return stm << v.get_path();
  }
  std::basic_ostream<wchar_t>&
   operator <<(std::basic_ostream<wchar_t>&                        stm
             , basic_search_sequence_value_type<wchar_t, . . .>& v)
  {
    return stm << v.get_path();
  }
} // namespace recls::stl
```

However, there are three problems with this code. First, we've got two functions doing much the same thing. Second, we've had to explicitly choose the traits type that will be supported with `basic_search_sequence_value_type`. Third, we've assumed that the stream will be derived from `std::basic_ostream`. A better alternative, which addresses all three problems, is to define the operator as a function template, as shown next. (I separated the return from the insertion, just in case someone wrote an inserter and forgot to provide the expected return type: a mutable [non-`const`] reference to the stream.)

```
template<typename S, typename C, typename T>
S& operator <<(S& s, basic_search_sequence_value_type<C, T> const& v)
{
  s << v.get_path();
  return s;
}
```

This can handle any traits type with which `basic_search_sequence` may be specialized and works with any stream type that can insert specializations of `std::basic_string` (the default string type of the **recls**/**STL** mapping). And with one additional little flourish—a string access shim, of course—we can make it compatible with *any* stream type that understands C-style strings.

```
template<typename S, typename C, typename T>
S& operator <<(S& s, basic_search_sequence_value_type<C, T> const& v)
{
  s << stlsoft::c_str_ptr(v.get_path()));
  return s;
}
```

Now you can stream to `std::cout` or even, should you so wish, to an instance of **MFC**'s `CArchive`!

Tip: Implement generic insertion operators.

34.5 Summary

We've examined the `std::ostream_iterator` component and shown how, with relatively little effort, we can enhance its design to support the principles of *Composition*, *Diversity*, and *Modularity*. And, although we've willingly (but advisedly) transgressed an important C++ design principle, the component also supports the *Principle of Least Surprise*.

CHAPTER 35

Intermezzo: Proscribing Fatuous Output Iterator Syntax Using the Dereference Proxy Pattern

If knowledge can create problems, it is not through ignorance that we can solve them.
—Isaac Asimov

As was discussed in the last chapter, `std::ostream_iterator` supports the required semantics of an output iterator by using a simple implementation trick, whereby the dereference operator (`operator *()`) returns an instance to the object and an assignment operator (`operator =()`) taking the assigned type is defined. (See Listing 34.5 to see how it is implemented for `stlsoft::ostream_iterator`.) Consider the following code:

```
1  std::ostream_iterator<int>  iter(std::cout);
2
3  *iter = 10;  // Sensible
```

Line 3 is equivalent to:

```
iter.operator *().operator =(10);
```

It is also, surprisingly, equivalent to:

```
iter.operator *();    // Returns iter
iter.operator =(10);
```

Unfortunately, this implementation technique also means that the following fatuous code is quite legal:

```
4  iter = 11;   // Silly
```

Since an output iterator is modeled after a pointer, this is tantamount to writing the following:

```
5  int  buffer[10];
6  int* iter = &buffer[0];
7
8  *iter = 10;
9  iter = 11;   // Compilation error!
```

`std::ostream_iterator` supports a syntax that the output iterator refinement does not prescribe. The abstraction has sprung a leak! But worry not, we're going to plug it using a pattern I call ***Dereference Proxy***.

Definition: A dereference proxy is a nested class that is returned from the dereference operator and has an appropriately defined assignment operator.

Using this pattern would change the expansion of line 3 to the following:

```
std::ostream_iterator<int>::deref_proxy prx = iter.operator *();
prx.operator =(10);
```

And it turns line 4 into a compilation error, which is nice.

35.1 `stlsoft::ostream_iterator::deref_proxy`

Retrofitting `stlsoft::ostream_iterator` with a dereference proxy involves four simple steps. First, the nested class is declared and given friend status, as shown in Listing 35.1.

Listing 35.1 Enhanced Definition Using the Dereference Proxy Pattern: Nested Class

```
template< typename V
        , typename C = char
        , typename T = std::char_traits<C>
        , typename S = std::basic_string<C, T>
        >
class ostream_iterator
  : public std::iterator<std::output_iterator_tag
                        , void, void, void, void>
{
public: // Member Types
  typedef V                                          assigned_type;
  . . .
  typedef std::basic_ostream<char_type, traits_type>  ostream_type;
  typedef ostream_iterator<V, C, T, S>               class_type;
private:
  class deref_proxy;
  friend class deref_proxy;
public: // Construction
  explicit ostream_iterator(ostream_type& os);
  . . .
```

Second, the public assignment operator is turned into a private method, which it is my practice to call `invoke_()` (Listing 35.2).

Listing 35.2 Enhanced Definition Using the Dereference Proxy Pattern: `invoke_()`

```
~~public: // Assignment~~
  ~~class_type& operator =(assigned_type const& value)~~
private: // Implementation
  void invoke_(assigned_type const& value)
  {
    m_stm << m_prefix << value << m_suffix;
  }
  . . .
```

Third, the `deref_proxy` class is defined (Listing 35.3).

Listing 35.3 Enhanced Definition Using the Dereference Proxy Pattern: Definition of `deref_proxy`

```
private:
  class deref_proxy
  {
  public: // Construction
    deref_proxy(ostream_iterator* it)
      : m_it(it)
    {}
  public: // Assignment
    void operator =(assigned_type const& value)
    {
      m_it->invoke_(value);
    }
  private: // Member Variables
    ostream_iterator* const   m_it;
  };
  . . .
```

Finally, the dereference operator return value is changed to return an instance of `deref_proxy`, rather than a reference to the iterator class (Listing 35.4).

Listing 35.4 Enhanced Definition Using the Dereference Proxy Pattern: Dereference Operator

```
public: // Output Iterator Methods
  ~~class_type& operator *()~~
  deref_proxy operator *()
  {
    return deref_proxy(this);
  }
  class_type& operator ++();
  class_type operator ++(int);
  . . .
};
```

And that's it! Note that I've not bothered to make `deref_proxy` a reusable template, in order to avoid the nonstandard nature of defining templates as friends (or having to make `invoke_()` public).

So, to avoid fatuous and abstraction-leaky syntax, make use of the following tip.

Tip: Always implement output operators by using the ***Dereference Proxy*** pattern.

C H A P T E R 3 6

Transform Iterator

The most common of all follies is to believe passionately in the palpably not true.
It is the chief occupation of mankind.
—Henry Louis Mencken

I reject your reality, and substitute my own!
—Adam Savage

36.1 Introduction

This chapter, the first on the subject of iterator adaptors, describes a simple iterator adaptor that allows the apparent values and/or types of the elements in a range to be modified, via the iterator adaptor class template `stlsoft::transform_iterator`. It illustrates several of the issues associated with writing iterator adaptors, including determination of the adapted iterator category, support for different iterator category refinements, and the restrictions on selection of the supported reference category in respect to underlying and apparent iterator categories.

We're going to focus on the transformation of values in a container of `int` for the purposes of this chapter, but I'll give you a taste now of how such transformations can be applied in nontrivial cases, to enable iterators to be used with algorithms. In Chapter 28 we looked at `enumerator_sequence` and used it to wrap the **recls**/**COM** COM objects in a discoverable and exception-safe STL collection. In Listing 28.2 we showed the shorthand version of a block of code that will conduct a file system search and output the results of invoking the `IFileEntry:: get_Path()` method on each entry. Despite use of an STL collection, that version still required the coding of a manual loop, something counter to the STL spirit. Using `transform_iterator` (via its creator function `transformer()`), we can rewrite this code using standard components and the **COMSTL** unary function class `propget_invoker` (via its creator function `propget_invoke()`), as shown in the following example. (Don't get hung up on the details here—`propget_invoker` is one of the function classes discussed in Volume 2. For now, all you need to know is that you specify an interface method and return type, and when the function object is invoked, it squirrels away the returned property value in an instance of the return type.)

```
typedef comstl::bstr  bstr_t;

std::copy(stlsoft::transformer( entries.begin()
            , comstl::propget_invoke<bstr_t>(&IFileEntry::get_Path))
```

```
        , stlsoft::transformer( entries.end()
              , comstl::propget_invoke<bstr_t>(&IFileEntry::get_Path))
        , std::ostream_iterator<bstr_t, wchar_t>(std::wcout, L"\n"));
```

Some issues of discoverability remain, which will be discussed later in the chapter, and no one could reasonably claim that this is a trivial bit of code. However, with a little thought it can be readily digested. The creator function `transformer()` takes an iterator and a function object instance. The creator function `propget_invoke()` takes the address of a COM interface method and is explicitly specialized with the return type `bstr_t` (`comstl::bstr`). `propget_invoke()` returns a function object that will invoke the method `IFileEntry::get_Path()` and return an instance of `bstr_t`. `transformer()` returns an iterator that will apply that function object to the underlying range (denoted by [`entries.begin()`, `entries. end()`). `ostream_iterator<bstr_t, wchar_t>` is specialized so as to work with `std::wcout` and expects to be assigned instances of `bstr_t`. `std::copy()` ties the transformed range to the output, and the paths of all entries are written to successive lines.

I've shown you this case more as an example of the power of what can be achieved by combining STL extension components, rather than as a motivation for why we need a transforming iterator. For that, we will use a simpler example and appeal to the base instinct of all C++ programmers: efficiency.

36.2 Motivation

Let's say that we have a range of integer values and we want to calculate the mean of the absolute value of each element. Assume that we have available to us an algorithm `avg_mean()` that can calculate the mean of a range of numbers of arbitrary type (Listing 36.1). This algorithm will work with input iterators (or any higher refinement).

Listing 36.1 Definition of the `avg_mean()` Function

```
template <typename T // "Value" type
        , typename I // Iterator type
        >
T avg_mean(I from, I to)
{
  T sum = 0;
  T n   = 0;
  for(; from != to; ++from, ++n)
  {
    sum += *from;
  }
  return (0 == n) ? sum : sum / n;
}
```

We cannot simply apply the `avg_mean()` algorithm to our range of numbers since we want a mean of the absolutes. We need to transform, by negation, any negative values in the range. How can this be done?

36.2.1 Using `std::transform()`

Classic STL suggests that we use the `std::transform()` algorithm. One way to do this is in combination with `std::back_inserter()`, as shown in Listing 36.2. (In order to derive meaningful comparisons between the different approaches examined, each one will use the C library function `abs()`. It would be more efficient to manually test and negate where possible, but we need to be able to observe the performance differences due only to the different algorithmic approaches.)

Listing 36.2 Definition of `calc_abs_mean()` Using `back_inserter()`

```
template <typename T // "Value" type
        , typename I // Iterator type
        >
T calc_abs_mean(I from, I to)
{
  std::vector<T>  absolutes;
  std::transform(from, to
               , std::back_inserter(absolutes), std::ptr_fun(::abs));
  return avg_mean<T>(absolutes.begin(), absolutes.end());
}
```

Alternatively, we could transform the contents of a container in place, as in Listing 36.3.

Listing 36.3 Definition of `calc_abs_mean()` Using a Range Constructor

```
template <typename T, typename I>
T calc_abs_mean(I from, I to)
{
  std::vector<T>  absolutes(from, to); // Copy the source range
  std::transform( absolutes.begin(), absolutes.end(), absolutes.begin()
                , std::ptr_fun(::abs));
  return avg_mean<T>(absolutes.begin(), absolutes.end());
}
```

Note, in both cases, the use of the explicit argument list to inform the compiler of the algorithm's number type. This is needed because the function template does not use the template parameter `T` in the function argument list. (There are techniques for inferring types, which would work in this case, but that's more an issue for Volume 2, so we'll go with nice and simple here.)

Both versions look nice, are eminently discoverable, and earn us double brownie points because we've used two standard algorithms and a function object adaptor and have hand-coded nothing. Unfortunately, looks deceive in this case. There's actually an awful lot of work going on here for such a simple task. Too much, in fact, and it shows up in the poor performance that we'll discuss later.

Two aspects of each implementation result in higher than necessary costs. First, a container is created as a scratch area in which to write the transformed values, costing time and memory. Second, the elements in the source range [`from`, `to`) have to be copied to the container. In the version using `std::back_inserter()`, this copying has to be done one element at a time, possibly precipitating multiple allocations in the container and further increasing costs.

36.2.2 Using the Transform Iterator

In both cases that use `std::transform()`, the standard C function `abs()` is used to calculate the absolute value from the element value. Wouldn't it be nicer to use the results of the `abs()` calls directly with `avg_mean()`, rather than having to use a costly intermediate? With `transform_iterator`, we can (Listing 36.4).

Listing 36.4 Definition of `calc_abs_mean()` Using `transformer()`

```
template <typename T, typename I>
T calc_abs_mean(I from, I to)
{
  return avg_mean<T>( stlsoft::transformer(from, std::ptr_fun(::abs))
                    , stlsoft::transformer(to, std::ptr_fun(::abs)));
}
```

It looks nice, but does its apparent simplicity translate into good performance? Six versions were compared: the two originals, variants of each of those using `std::list<T>`, the version using `transform_iterator`, and the hand-coded algorithm shown in Listing 36.5.

Listing 36.5 Hand-Coded Form of `calc_abs_mean()`

```
template <typename T, typename I>
T calc_abs_mean(I from, I to)
{
  T sum = 0;
  T n   = 0;
  for(; from != to; ++from, ++n)
  {
    sum += ::abs(*from);
  }
  return (0 == n) ? sum : sum / n;
}
```

The results of a scenario that calculates the absolute mean of a range of 100,000 integers, repeated 100 times, are shown in Table 36.1. They make a pretty compelling argument that copying and transforming in place is a bad idea. Note that, as is so often the case with the STL, the combination of `transform_iterator` with the algorithm is on a par with the hand-coded version.

Table 36.1 Relative Times (in Milliseconds) of `calc_abs_mean()` Implementations, with 100 Iterations of 100,000 Integers

Function	CodeWarrior 8	Digital Mars 8.45	GCC 3.4	Intel 8	Visual C++ 7.1
`list` with `back_inserter`	2,682	4,519	9,546	7,153	7,564
`list` with range constructor	2,745	4,569	9,713	7,961	7,634
`vector` with `back_inserter`	597	352	622	568	523
`vector` with range constructor	346	427	477	405	462
`transform_iterator`	128	146	160	158	161
Hand-coded	112	144	168	158	159

This impressive performance is not just a function of the extent of the size of the source range of integers. Using a small size yields a very similar outcome, as can be seen in Table 36.2, though in this case the hand-coded form generally has the advantage over `transform_iterator`.

Table 36.2 Relative Times (in Milliseconds) of `calc_abs_mean()` Implementations, with 100,000 Iterations of 100 Integers

Function	CodeWarrior 8	Digital Mars 8.45	GCC 3.4	Intel 8	Visual C++ 7.1
`list` with `back_inserter`	1,956	4,001	5,660	4,425	4,926
`list` with range constructor	1,840	4,349	5,954	4,466	4,993
`vector` with `back_inserter`	650	694	651	676	788
`vector` with range constructor	189	287	231	117	187
`transform_iterator`	114	109	107	20	75
Hand-coded	48	70	25	21	24

We've established that `transform_iterator` is worth a look, so let's dig now into the implementation.

36.3 Defining Iterator Adaptors

When defining iterator adaptor classes, there are several things you must consider.

- What is the value type of the adaptor?
- What are the other member types?
- Can the adaptor support the iterator category of the base iterator type?
- Can the adaptor support the element reference category (Section 3.3) of the base iterator type?
- How will instances of the adaptor be created?

36.3.1 Creator Functions

The last of these questions is generally the easiest to answer because most iterator adaptors use creator functions. The template parameter list of `transform_iterator` consists of two parameters: the type of the base iterator and the type of the unary function that will perform the transformation.

```
template< typename I // Base iterator type
        , typename F // Unary function type
        >
class transform_iterator;
```

Given that, we can define a creator function `transformer()` as follows. (The intermezzo following this chapter will discuss the issues involved in choosing a name for this creator.)

```
template <typename I, typename F>
transform_iterator<I, F> transformer(I it, F fn)
{
  return transform_iterator<I, F>(it, fn);
}
```

This affords a considerable simplification in client code. Where earlier we saw the use of the `transformer()` creator function to neatly implement `calc_abs_mean()`, things would be decidedly messier were we to specialize the iterator adaptor template ourselves:

```
template <typename I, typename F>
T calc_abs_mean(I from, I to)
{
  return avg_mean<T>( stlsoft::transform_iterator<I
                           , std::pointer_to_unary_function<int, int>
                           >(from, std::ptr_fun(::abs))
                    , stlsoft::transform_iterator<I
                          , std::pointer_to_unary_function<int, int>
                          > (to, std::ptr_fun(::abs)));
}
```

Awful stuff! Programmers would think twice if they had to write such code for a simple operation, and there's precious little chance that the more advanced uses of iterator adaptation would ever see the inside of a lexer.

36.3.2 Value Type

Since the transform iterator uses a unary function object to effect its transformation, we might reasonably infer that the iterator's `value_type` will be the function's `result_type`.

However, the remaining aspects of the adaptation are decidedly less obvious and can lead you down a blind alley or two, so I intend to employ the lowest trick of the struggling author: Use deliberately flawed versions to highlight the issues in absurdum before triumphantly parading a sound version for your delectation.

36.4 `stlsoft::transform_iterator`

36.4.1 Version 1

Listing 36.6 shows a definition of a first version of `transform_iterator`, which contains a couple of imperfections. Take a moment and note what you think might be wrong with it before reading on. (I confess I've never once in my life been able to follow such directions from an author—whether through impatience or dullness quite escapes my ken—so you mustn't feel the slightest pang if you can't do it either.)

Listing 36.6 First Version of `transform_iterator`

```
template <typename I, typename F>
class transform_iterator
{
public: // Member Types
  typedef I                                     iterator_type;
  typedef F                                     transform_function_type;
  typedef typename transform_function_type::result_type
                                                value_type;
  typedef std::iterator_traits<I>               traits_type;
  typedef typename traits_type::iterator_category
                                                iterator_category;
  typedef typename traits_type::difference_type
                                                difference_type;
  typedef typename traits_type::pointer         pointer;
  typedef typename traits_type::reference       reference;
  typedef ????                                  const_pointer;
  typedef ????                                  const_reference;
  typedef transform_iterator<I, F>              class_type;
public: // Construction
  transform_iterator(iterator_type it, transform_function_type fn)
    : m_it(it)
    , m_transformer(fn)
  {}
  transform_iterator()
    : m_it()
    , m_transformer()
  {}
  iterator_type base() const
  {
    return m_it;
  }
```

```
public: // Accessors
  reference operator *()
  {
    return m_transformer(*m_it);
  }
  const_reference operator *() const
  {
    return m_transformer(*m_it);
  }
  pointer operator ->()
  {
    ????
  }
  const_pointer operator ->() const
  {
    ????
  }
public: // Input/Forward Iterator Methods
  class_type& operator ++()
  {
    ++m_it;
    return *this;
  }
  class_type& operator ++(int); // Standard boilerplate implementation
public: // Bidirectional Iterator Methods
  class_type& operator --()
  {
    --m_it;
    return *this;
  }
  class_type& operator --(int); // Standard boilerplate implementation
public: // Random Access Iterator Methods
  class_type& operator +=(difference_type d)
  {
    m_it += d;
    return *this;
  }
  class_type& operator -=(difference_type d)
  {
    m_it -= d;
    return *this;
  }
  reference operator [](difference_type index)
  {
    return m_transformer(m_it[index]);
  }
  const_reference operator [](difference_type index) const
  {
    return m_transformer(m_it[index]);
  }
public: // Comparison
  bool equal(class_type const& rhs) const // Used by ==, !=
  {
    return m_it == rhs.m_it;
  }
  int compare(class_type const& rhs) const // Used by <, <=, >, >=
  {
    return m_it - rhs.m_it;
  }
  difference_type distance(class_type const& rhs) const // Used by –
  {
    return m_it - rhs.m_it;
  }
private: // Member Variables
  iterator_type           m_it;
  transform_function_type m_transformer;
};
```

Before we deal with the problems in this version, let's briefly cover the aspects that are correct since they will be common to all the versions discussed henceforth, including the final (correct) one.

36.4.2 Construction

Two constructors are explicitly defined. The first takes an instance of the base iterator type and an instance of the transforming function, with which the two member variables are initialized. The second is a default constructor, facilitating the common (pretty much mandatory) requirement to be able to declare an instance of the iterator type without explicit initialization.

The class also has a third, implicitly defined, constructor: the copy constructor. Since iterators and function objects must be *CopyConstructible*, we can leave it to the compiler to provide this constructor. The same applies to the copy assignment operator, which is similarly well defined by the compiler because iterators and function objects must be *Assignable*.

The other method grouped in the **`Construction`** section is `base()`, which returns a copy of the current state of the base iterator member `m_it`. This follows the convention established by `std::reverse_iterator`.

36.4.3 Increment and Decrement Operators and Pointer Arithmetic Methods

The increment and decrement operators, which support the *input/forward* and *bidirectional* iterator categories, respectively, are implemented correctly. They simply increment or decrement, respectively, the base iterator member `m_it`. There's nothing more required of them because `transform_iterator` transforms the values and/or types of the elements of the underlying range; it does not alter the number or arrangement of the elements. (We'll look at such an iterator adaptor in Chapter 42.)

The two other methods for altering the iterator position, `operator +=()` and `operator -=()`, are also well defined, though only in the case that the base iterator type has the *random access* (or *contiguous*) category. If it does not, the compiler will refuse to instantiate these methods in just the same way as it does for any template instantiation where the function body writes cheques (that's *checks* to my U.S. friends) that its putative specializing type(s) cannot cash.

36.4.4 Comparison and Arithmetic Operators

The `equal()`, `compare()`, and `distance()` methods are provided as a convenience for the implementation of the free function comparison and arithmetic operators, though the latter two are well defined only in the case where the base type iterator is random access or contiguous. The implementation of the free function (in)equality comparison operator functions `operator ==()` and `operator !=()` in terms of `equal()` has been a consistent theme throughout Part II, so I'm taking that as a given. We can also define the comparison and arithmetic operators in terms of the public methods, as shown in Listing 36.7.

Listing 36.7 Implementation of the Comparison and Arithmetic Operators

```
template <typename I, typename F>
bool operator <(transform_iterator<I, F> const &lhs
              , transform_iterator<I, F> const &rhs)
{
  return lhs.compare(rhs) < 0;
}
template <typename I, typename F>
bool operator <=( transform_iterator<I, F> const &lhs
                , transform_iterator<I, F> const &rhs)
{
  return lhs.compare(rhs) <= 0;
}
. . . // And > and >=
template <typename I, typename F>
transform_iterator<I, F>
  operator +( transform_iterator<I, F> const &lhs
            , typename transform_iterator<I, F>::difference_type rhs)
{
  return transform_iterator<I, F>(lhs) += rhs;
}
. . . // And operator -
```

36.4.5 And the Problem Is . . .

Let's see the flaw in the first version by taking it for a test drive. Consider the following code:

```
std::list<int>  ints(1);
std::copy(stlsoft::transformer(ints.begin(), std::ptr_fun(::abs))
        , stlsoft::transformer(ints.end(), std::ptr_fun(::abs))
        , std::ostream_iterator<int>(std::cout, " "));
```

This fails to compile, with the compiler informing us that it cannot convert a temporary instance to a mutating reference (i.e., reference to non-`const`):

```
. . .
reference operator *()
{
  return m_transformer(*m_it); // Compile error!
}
. . .
```

The problem is that the `reference` member type of the iterator is the `reference` member type of the base iterator type (`std::list<int>::reference`), which is `int&`. But the `result_type` of `std::pointer_to_unary_function<int, int>` (what the creator function `std::ptr_fun(::abs)` returns) is `int`, and as a function return it is an *rvalue*. For very good reasons, C++ does not allow conversion from *rvalue* to a mutating reference. Hence, the code does not compile.

There's a second problem. Note the question marks in the definition of the `const_pointer` and `const_reference` member types and in the implementation of the member selection operator, `operator ->()`. As it stands, there's no simple way to deduce these member types (though it can be done, using techniques we'll see later in Chapter 41 when implementing different types of iterator adaptation). And there's no clear way in which we can support member selection operators since to what would we return a pointer?

Finally, since the iterator can be used to transform the apparent value type, for example, from `int` to `std::string`, it's quite wrong to attempt to define the member types of the adaptor in terms of the base iterator type, notwithstanding the compelling foregoing practical objections.

36.4.6 Version 2

Clearly, we cannot define the `reference` member type (and the `pointer` member type, for that matter) in terms of the underlying iterator. But the final criticism holds a key. Perhaps we can define `pointer` and `reference` as shown in Listing 36.8 (which shows only the salient differences compared with version 1)?

Listing 36.8 Second Version of `transform_iterator`

```
template <typename I, typename F>
class transform_iterator
{
public: // Member Types
  . . .
  typedef typename traits_type::difference_type difference_type;
  typedef value_type*                           pointer;
  typedef value_type&                           reference;
  typedef value_type const*                     const_pointer;
  typedef value_type const&                     const_reference;
  . . .
public: // Accessors
  reference operator *()
  {
    m_current = m_transformer(*m_it);
    return m_current;
  }
  const_reference operator *() const
  {
    m_current = m_transformer(*m_it);
    return m_current;
  }
  pointer operator ->()
  {
    m_current = m_transformer(*m_it);
    return &m_current;
  }
  const_pointer operator ->() const
  {
    m_current = m_transformer(*m_it);
    return &m_current;
  }
  . . .
private: // Member Variables
  iterator_type           m_it;
  transform_function_type m_transformer;
  mutable value_type      m_current;
};
```

There are two differences from version 1. First, the member types `pointer`, `const_pointer`, `reference`, and `const_reference` are all defined in terms of the `value_type` of the adaptor, rather than (indirectly) that of the base iterator type. Second, the adaptor has a member variable `m_current` that holds the transformed value, to which references and pointers may be returned. This means that the iterator now has the *transient* element reference category (Section 3.3.4).

This version differs from the original in several significant aspects. For one thing, it will actually compile and run correctly with the code shown in Section 36.4.5, which is nice. Also, it means that the member selection operator(s) may be defined. However, two important problems remain.

First, the subscript operators are not well defined. Any attempt to compile an expression with the mutating (non-`const`) subscript operator will not compile, for the same reasons that we encountered with the dereference operator in version 1. And, worse, any attempt to compile an expression with the nonmutating (`const`) subscript operator will compile, but it will crash because the function returns a reference to a temporary, that temporary ceasing to exist at the end of the function, leaving client code hanging onto who knows what?

The second problem is even more insidious. Because the mutating (non-`const`) overloads of the dereference and member selection operators return mutable (non-`const`) pointers/references, client code may mutate the values thus returned. But those changes will only be written as far as the `m_current` variable, to be overwritten the next time any of these operators are called.

Of course, this is a result of our inadvertently supporting the apparent mutation by client code of a transformed value, something that makes no sense logically. It should therefore be little surprise that we've ended up with semantics that are fatuous at best.

This version could be fixed by eschewing support for random access iterators and by defining the `pointer` and `reference` member types to be nonmutating (`const`) (i.e., to make them equivalent to the `const_pointer` and `const_reference` member types, respectively). If that was the chosen approach, it might also be prudent to use a Boolean flag member variable to ensure that `m_current` is updated only in the case that it's stale, that is, after construction, or after increment/decrement. Listing 36.9 shows such an alternate version.

Listing 36.9 Alternate Stateful Version of `transform_iterator`

```
template <typename I, typename F>
class transform_iterator
{
  . . .
public: // Construction
  transform_iterator(iterator_type it, transform_function_type fn)
    : m_it(it)
    , m_transformer(fn)
    , m_stale(true)
  {}
  transform_iterator()
    : m_it()
    , m_transformer()
    , m_stale(true)
  {}
  . . .
public: // Accessors
  reference operator *()
  {
    if(m_stale)
    {
      m_current = m_transformer(*m_it);
      m_stale   = false;
    }
    return m_current;
  }
  const_reference operator *() const; // As above
  pointer operator ->();              // As above, but return &m_current
  const_pointer operator ->() const;  // As above, but return &m_current
  . . .
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    ++m_it;
    m_stale = true;
    return *this;
  }
  . . .
public: // Bidirectional Iterator Methods
  class_type& operator --()
  {
    --m_it;
    m_stale = true;
    return *this;
  }
  . . .
private: // Member Variables
  iterator_type           m_it;
  transform_function_type m_transformer;
  mutable value_type      m_current;
  bool                    m_stale;
};
```

In cases where you don't need to adapt random access iterators, this is an adequate strategy.

Tip: Consider caching the value obtained from a base iterator type in an iterator adaptor, to support a member selection operator, when support for random access iterators is not required.

36.4.7 `stlsoft::transform_iterator`

The approach I chose for the version of `transform_iterator` that appears in the **STLSoft** libraries recognizes that a transforming iterator is, at heart, a *by-value temporary* (Section 3.3.5) element reference beast. The final version is shown in Listing 36.10 (with only salient differences from the other versions).

Listing 36.10 Final Version of `transform_iterator`

```
// In namespace stlsoft
template <typename I, typename F>
class transform_iterator
{
public: // Member Types
  . . .
  typedef typename traits_type::difference_type
                                              difference_type;
  typedef void                                    pointer;
  typedef void                                    reference;
  typedef value_type                              effective_reference;
  typedef const value_type                        effective_const_reference;
  typedef transform_iterator<I, F>                class_type;
public: // Construction
  . . .
public: // Accessors
  effective_reference       operator *()
  {
    return m_transformer(*m_it);
  }
  effective_const_reference operator *() const
  {
    return m_transformer(*m_it);
  }
  // NOTE: No member selection operator (see below)
  . . .
public: // Random Access Iterator Methods
  class_type& operator +=(difference_type d);
  class_type& operator -=(difference_type d);
  effective_reference       operator [](difference_type index)
  {
    return m_transformer(m_it[index]);
  }
  effective_const_reference operator [](difference_type index) const
  {
    return m_transformer(m_it[index]);
  }
  . . .
private: // Member Variables
  iterator_type           m_it;
  transform_function_type m_transformer;
private: // Not to be implemented
  struct iterator_is_BVT_so_no_member_selection_operators
  {};
  iterator_is_BVT_so_no_member_selection_operators* operator ->() const;
};
```

Because it has an element reference category of by-value temporary, `transform_iterator` defines `pointer` and `reference` member types to `void`. It uses a convention of my own in defining `effective_reference` and `effective_const_reference` member types, defined as `value_type` and `const value_type`, respectively, for the return types of the dereference and subscript operators. These operators are now well defined, irrespective of the characteristics of the base iterator type.

Obviously there are no analogous `effective_pointer` and `effective_const_pointer` member types because a by-value temporary iterator cannot support the member selection operator. If you try to compile code that attempts to invoke this operator on such an iterator, the compiler will usually inform you that `operator ->()` is not defined for the given type. If you're an iterator guru, that may be enough, but I add a little helpful flourish in declaring this operator private and giving it an informative return type. Any users who may not know get a better clue as to their misstep.

Rule: Transforming iterator adaptors that support random access iteration must have by-value temporary element reference category semantics. Those that do not support random access iteration may provide transient element reference category semantics and member selection operators.

36.5 Composite Transformations

The interesting thing about accepting the inevitable nature of transforming iterators as being by-value temporary is that they can now be used to adapt *any* iterator type (other than output iterators, of course), even those that are themselves by-value temporary. This means that we can chain transformations, as in the following:

```
struct entry_to_mod_time
  : public std::unary_function< recls::stl::search_sequence::value_type
                              , recls::recls_time_t>
{
  typedef recls::stl::search_sequence::value_type   value_type;
  recls::recls_time_t operator ()(value_type const& entry) const
  {
    return entry.get_modification_time();
  }
};
struct time_to_string
  : public std::unary_function<recls::recls_time_t, std::string>
{
  std::string operator ()(recls::recls_time_t const& tm) const
  {
    return stlsoft::c_str_ptr(tm);
  }
};

recls::stl::search_sequence files(".", "*.h|*.hpp"
                                , recls::FILES | recls::RECURSIVE);
std::copy(stlsoft::transformer( stlsoft::transformer( files.begin()
                                                    , entry_to_mod_time())
                              , time_to_string())
```

```
        , stlsoft::transformer( stlsoft::transformer( files.end()
                                                    , entry_to_mod_time())
                              , time_to_string())
        , std::ostream_iterator<std::string>(std::cout, "\n"));
```

Here, two reasonably generic function types, `entry_to_mod_time` and `time_to_string`, act in concert by doing double transformations. One of my reviewers observed, "This is *cool*!" You might agree. I couldn't possibly comment.

36.6 *DRY SPOT* Violations?

The main downside in using iterator adaptors such as `transform_iterator`, particularly in compositions like the one shown in the previous section, is the apparent *DRY SPOT* violation (Chapter 5). We are required to specify the function object in both iterator expressions, even though the endpoint iterator never makes use of it. The reason for this is that the standard algorithms are defined to take a single iterator template parameter. In other words, `std::copy()` is declared as:

```
template< typename I  // The input iterator type
        , typename O  // The output iterator type
        >
O copy(I first, I last, O result);
```

rather than as:

```
template< typename I1 // An input iterator type
        , typename I2 // Another input iterator type
        , typename O  // The output iterator type
        >
O copy(I1 first, I2 last, O result);
```

We might presume that this is to cut down on ambiguity, or it might just be that no one considered support for heterogeneous input iterators at the time. Whatever the reason might be, it's irrelevant since it's unlikely to change.

36.6.1 Use Typedefs and Nontemporary Function Objects

One option we may elect to use is to define a typedef that encapsulates all the type information that would otherwise be inferred in two places by the compiler (based on the creator function). For our implementation of `calc_abs_mean()`, this would look like the following:

```
template <typename T, typename I>
T calc_abs_mean(I from, I to)
{
  typedef std::pointer_to_unary_function<int, int>  fun_t;
  return avg_mean<T>( stlsoft::transformer(from, fun_t(::abs))
                    , stlsoft::transformer(to, fun_t(::abs)));
}
```

This has bought us nothing but less transparent code because the user could still mistakenly specify, say, `toupper()`, to the second `fun_t` instance. (Not that it matters with `avg_mean()`, of course, since the endpoint iterator doesn't use its function object.) We can take it a step further and declare a *SPOT* (Chapter 5) for both type and adapted function by declaring a single explicit nontemporary *instance* of the function object:

```
template <typename T, typename I>
T calc_abs_mean(I from, I to)
{
  std::pointer_to_unary_function<int, int>  fn(::abs);
  return avg_mean<T>( stlsoft::transformer(from, fn)
                    , stlsoft::transformer(to, fn));
}
```

This is certainly an improvement, but it's still poorer than the original version. However, that's largely due to the fact that the original function object expression, `std::ptr_fun(::abs)`, is not terribly complex in and of itself. Contrast that with the expression involving `propget_invoke()` shown below. Applying the explicit instance technique here does, in my opinion, lead to an increase in discoverability.

```
typedef comstl::bstr                    bstr_t;
typedef comstl::propget_invoker< bstr_t
                                , IFileEntry
                                , BSTR
                                >     transform_t;

transform_t   tx(&IFileEntry::get_Path);
std::copy(stlsoft::transformer(entries.begin(), tx)
        , stlsoft::transformer(entries.end(), tx)
        , std::ostream_iterator<bstr_t, wchar_t>(std::wcout, L"\n"));
```

Tip: Consider using nontemporary function objects to simplify complex iterator adaptation expressions.

This is still not terribly impressive, though. (A better solution in this case is to use *ranges*, but that will have to wait until Volume 2. Well, I've got to keep back *some* incentive for you to buy it.)

36.6.2 Use Heterogeneous Iterators and Algorithms

It would be so much nicer if we could skip the duplication and just write expressions such as:

```
return avg_mean<T>( stlsoft::transformer(from, std::ptr_fun(::abs))
                  , to);
```

and:

```
std::copy(stlsoft::transformer( entries.begin()
            , comstl::propget_invoke<bstr_t>(&IFileEntry::get_Path))
        , entries.end()
        , std::ostream_iterator<bstr_t, wchar_t>(std::wcout, L"\n"));
```

and have the compiler handle it all for us. After all, the adapted forms of `from` and `entries.begin()` are really manipulating things that are meaningfully comparable with `to` and `entries.end()`. So another approach would be to define an equivalent set of algorithms to those provided in the standard but have them take heterogeneous iterator types. Listing 36.11 shows a heterogeneous iterator version of `copy()` defined in the notional `stdex` namespace.

Listing 36.11 Notional `copy()` Algorithm Taking Heterogeneous Iterator Types

```
namespace stdex
{
  template< typename I1
          , typename I2
          , typename O
          >
  O copy(I1 i1, I2 i2, O o)
  {
    for(; from != to; ++from, ++result)
    {
      *result = *from;
    }
    return result;
  }
} // namespace stdex
```

Consider that we might add additional facilities to `transform_iterator` to support such algorithms, as shown in Listing 36.12.

Listing 36.12 Notional Adjustment to `transform_iterator` and Operators

```
template <typename I, typename F>
class transform_iterator
{
public: // Member Types
  . . .
  typedef I         iterator_type;
  . . .
public: // Comparison
  bool equal(class_type const& rhs) const;
  bool equal(iterator_type it) const
  {
    return m_it == it;
  }
  int compare(class_type const& rhs) const;
  int compare(iterator_type it) const;
  difference_type distance(class_type const& rhs) const;
  difference_type distance(iterator_type it) const;
  . . .
};

template <typename I, typename F>
bool operator ==(transform_iterator<I, F> const& lhs
               , transform_iterator<I, F> const& rhs);

template <typename I, typename F>
bool operator ==( transform_iterator<I, F> const& lhs, I rhs)
{
  return lhs.equal(rhs);
}

template <typename I, typename F>
bool operator ==(I lhs, transform_iterator<I, F> const& rhs)
{
  return rhs.equal(lhs);
}

. . . // And for !=, <, <=, >, >=, +, -
```

I've experimented with this technique a great deal. It's a really nice solution with one all-too-obvious flaw: It puts the burden on the user to remember to use the extended algorithms, rather than those from the standard. As such, it's not likely to be a winning strategy in the main. But you should keep it in mind for those occasions where you can control the client code and the payoff is worth the use of a nonstandard algorithm.

36.6.3 Wear It, but Beware

Perhaps the key here is to realize that, although there's a duplication of typing effort, such cases rarely represent a true *DRY SPOT* violation, precisely because of the homogeneity of the iterator type in the standard algorithms. This means that the compiler will protect us from writing code that will violate *DRY SPOT*, *in most cases*. For example, if you write the following erroneous code, the compiler will rescue you from your stupor:

```
  return avg_mean<T>( stlsoft::transformer(from, std::ptr_fun(::abs))
                    , stlsoft::transformer(to, std::ptr_fun(::labs)));
```

This is an error because `labs()` takes and returns a `long`, whereas `abs()` takes and returns an `int`. However, this type resolution is not a total proof against error. As I mentioned earlier, if we'd specified `toupper()`, or any function with the same argument and return type as `abs()`, the code would compile. When used with algorithms where the iterators are in pairs and the second is an endpoint iterator, this error is benign. However, where the algorithm may use the elements represented by two or more of its iterator arguments, you must take particular care to ensure that the right function is used. In such cases, you should get another pair of human eyes on your code or use nontemporary function objects, or both.

36.7 A Spoonful of Sequence Helps the Medicine . . . ?

I spent significant effort throughout Part II attempting to convince you of the benefits of adapting real-world collections to STL extension collections, rather than as iterators. Perhaps some of that same medicine is appropriate here?

Alas, that's not the case. Despite the drawback that two creator functions in the same expression cannot communicate either type or state to each other in a usable fashion, the fact remains that creator functions do a large amount of the heavy lifting for us. Doing away with them is almost always a retrograde step. Consider the following rewrite of `calc_abs_mean()` in terms of a notional `iterator_adaptor_sequence` class template: It's 100% *DRY SPOT* compliant, but it's ugly and tryingly verbose.

```
template <typename T, typename I>
T calc_abs_mean(I from, I to)
{
  iterator_adaptor_sequence<I
                          , std::pointer_to_unary_function<int, int>
                          >   s(from, to, std::ptr_fun(::abs));
  return avg_mean<T>(s.begin(), s.end());
}
```

So, for iterator adaptation in client code, I think it's safe to recommend the following tip.

Tip: Prefer creator functions over sequence adaptors.

36.8 Summary

In this chapter, we've looked at how an iterator may be used to transform the types and/or values of the elements of an underlying sequence, and I demonstrated that such an iterator can be beneficial in terms of performance and, in combination with creator functions, in terms of syntax. We've seen that the implementation of such an iterator must involve compromise: Either a transforming iterator must proscribe random access iterator semantics, or it must prescribe the by-value temporary element reference category. The iterator component described, `stlsoft::transform_iterator`, takes the latter approach.

We've noted that the standard library algorithms use homogeneous iterator pairs to define ranges and seen that one consequence of this is that iterator adaptors can suffer from mild, and usually benign, *DRY SPOT* violations. But users must beware of this possibility.

36.9 On the CD

The CD contains a section that describes a compromise in functionality to allow `transform_iterator` to be used with older compilers that do not support the standard library's definition of `std::iterator_traits`.

CHAPTER 37

Intermezzo: Discretion Being the Better Part of Nomenclature . . .

You can tell a Yorkshireman, but you can't tell him much.

—(Uncle) Michael Gibbs

I originally wanted to give the `transform_iterator` creator function the name `transform()`, which the standard library already uses for its transforming algorithm overloads. Having these two different facilities share the same name should work because they're in separate namespaces. Further, even when both are brought into the same namespace via a *using-declaration*, there's no clash because the standard algorithm overloads have four and five parameters, and the `transform_iterator` creator function has only two.

```
#include <stlsoft/iterators/transform_iterator.hpp>
#include <algorithm>

using std::transform;
using stlsoft::transform;

int   from[5];
transform(&from[0], &from[0] + 5, &from[0], ::abs); // std::transform
transform(&from[0], ::abs);                         //
stlsoft::transform
```

Unfortunately, there is still a potential problem, albeit a political rather than technical one. Consider the case where <*stlsoft/iterators/transform_iterator.hpp*> has been included and <*algorithm*> (whether directly or indirectly) has not. Further, the user in this case has taken the foolhardy step of using *using-directives*, rather than *using-declarations* (something I'm sure you would never do, gentle reader).

```
#include <stlsoft/iterators/transform_iterator.hpp>

using namespace std;
using namespace stlsoft;
```

```
int   from[5];
transform(&from[0], &from[0] + 5, &from[0], ::abs); // std::transform
transform(&from[0], ::abs);                         //
stlsoft::transform
```

Things look sort of okay, but the compiler's going to be more crotchety than an art teacher taking games. To my Australasian readers, *games* means P.E. (physical education); to my U.S. readers, this means gym class. I've never understood why physical education lessons in England in the 1970s and 1980s were referred to as games since there was precious little fun to be had playing rugby in cold rain and a howling gale. Our sports master liked to join in and always made a point of tackling his charges at full pace, then lifting himself off the sodden pitch using the back of a boy's head. (The art teacher to whom I alluded used to hate taking games. He much preferred a custom form of cricket, wherein his shoe was the bat, and the unhappy behind of an unfortunate schoolboy played the part of the ball.)

Anyway, enough of this rancorous digression—we have a cross compiler to placate. It will rightly report that `transform()` takes two parameters and not four. The really nasty part of this, of course, is that the code shown above is not realistic in its simplicity. In the real world, if you've skipped the explicit inclusion of *<algorithm>*, it's likely to have been included indirectly elsewhere, and this problem will manifest itself when you port your pristine code to another platform (usually only when you've accompanied this effort by a grand hubristic fanfare about how confident you are it will work the first time when ported to another compiler or platform).

Although you might (like me) feel that eschewing the use of *using-directives* in all possible circumstances is the wisest course, you might not. Perhaps the less contentious option is to choose your names carefully.

Tip: Avoid using the names of functions defined in the standard.

Given that `transform()` is not appropriate, what should we call it? Alas, this is where the English language breaks down somewhat. Turning to the standard for examples, we discover the creator function names in Table 37.1.

Table 37.1 Iterator Creator Functions

Iterator Type	Creator Function
back_insert_iterator	back_inserter()
front_insert_iterator	front_inserter()
ostream_iterator	–
istream_iterator	–

When defined, the creator function names are nouns. This might incline us to follow suit and use the name `transformer()`. Interestingly, where creator functions are used for function objects, some are nouns and some verbs (Table 37.2).

Table 37.2 Function Adaptor Creator Functions

Function Type	**Creator Function**
`binder1st`	`bind1st()`
`binder2nd`	`bind2nd()`
`pointer_to_unary_function`	`ptr_fun()`
`pointer_to_binary_function`	`ptr_fun()`
`mem_fun_t`	`mem_fun()`
`mem_fun_ref_t`	`mem_fun_ref()`

There's not a whole lot of consistency here, so I chose to go with a noun form: `transformer()`. This works fine in this case, but we don't have to travel far to hit inconsistency again. Thinking of a name for a creator function for the `filter_iterator` class template, discussed in Chapter 42, is not so cut and dried. Should it be `filter()` or `filterer()`?

Given the vagaries of the English language, as in this case, I tend to provide an additional, thoroughly unambiguous name for a creator function, `make_`*`class_name`*`()`, for example, `make_transform_iterator()`, `make_filter_iterator()`, and so on. This is consistent with the standard creator for `std::pair`: `std::make_pair()`. This approach is exceedingly unlikely to cause a name clash, and you can always remember it.

Tip: Always provide a predictable equivalent to the creator function's short name.

CHAPTER 38

Member Selector Iterator

The only country in the world where being "too clever by half" is an insult.

—A. A. Gill, on England

If you want the rainbow, you've got to put up with the rain.

—Dolly Parton

38.1 Introduction

Throughout most of the rest of the book, compiler specifics are eschewed (and banished to the CD), and a utopian view of compiler and standard library facilities is presented. This chapter is different. We will see time after time how strange and subtle bugs, "features," and implementation-defined behavior stymie our attempts to create what is, in essence, a very simple component. Along the way we'll highlight issues of `const`-ness, compare performance of algorithm use with iterator adaptors against use with custom functions, and see just how useful generator templates can be. We will finally end up with an efficient and highly (though not completely) portable component that facilitates the use of STL algorithms with sequences of structures.

38.2 Motivation

The **Pantheios** library provides logging facilities that are easy to use, 100% type-safe, generic, extensible, and extremely efficient and that interface to customizable back-end logging services. Sounds like a big ask? Let's dig in.

Down in the lowest level of the **Pantheios** (C) API, the function `pantheios_log_n()` takes three arguments: the severity level, a pointer to an array of string structures, and the length of that array:

```
struct pan_slice_t
{
  size_t      len;
  char const* str;
};

void pantheios_log_n( int                 severity
                    , size_t              numSlices
                    , pan_slice_t const*  slices);
```

The sole purpose of this function, the lowest level of the core library, is to concatenate strings into a contiguous, nul-terminated, C-style string and pass that, along with the severity level, off to an externally defined function—that's where the customizable transport(s) comes in—as follows:

```
// This function carries the prepared statement to the transport(s)
int pantheios_be_logEntry(void*       feToken
                        , void*       beToken
                        , int         severity
                        , char const* entry
                        , size_t      cchEntry);

void pantheios_log_n( int                  severity
                    , size_t               numSlices
                    , pan_slice_t const*   slices)
{
  // Concatenate the n strings
  size_t  len     = . . .
  char*   result  = . . .

  pantheios_be_logEntry(. . . , severity, result, len);
}
```

In any logging library, you want to maximize efficiency, so avoiding unnecessary copies and allocation is a serious consideration. No matter how efficient the concatenation of string classes can be made to be (see Chapter 25 of *Imperfect C++*), a custom solution is always going to be a better option in a case such as this. Hence, the concatenation functionality in `pantheios_log_n()` is actually along the lines of the following steps.

1. Determine the total length required to store the concatenated sequence.
2. Allocate a buffer of that length.
3. Write the strings into the buffer.
4. Nul-terminate the buffer.

38.2.1 `std::accumulate()`

All of those steps can be written in C, but that's kind of tiresome. C++ is attractive for several reasons in such an instance. The buffer should be in a class so that it's automatically deallocated by the destructor; using `auto_buffer` (Section 16.2) affords significant efficiency gains for short log entries since it avoids a trip to the heap. Further, summing elements and concatenating sequences should be bread and butter to STL. As we'll see, however, it's not quite as simple as that.

Let's start by taking a look at the length calculation. Summation is effected in STL by use of the `accumulate()` function templates, defined in the standard as follows:

```
template< typename I  // Iterator type, e.g., int const*
        , typename V  // Value type, e.g., int
        >
V accumulate(I first, I last, V init);

template< typename I  // Iterator type, e.g., int const*
        , typename V  // Value type, e.g., int
        , typename BF // Binary function, e.g., int prod(int x, int y);
        >
V accumulate(I first, I last, V init, BF binary_func);
```

These functions are great for summing sequences, as follows:

```
int ai[] = { 1, 3, 5, 7, 9 };

assert(25 == std::accumulate(&ai[0], &ai[0] + STLSOFT_NUM_ELEMENTS(ai),
0));
```

The `init` parameter serves two purposes. It initializes the sequence, usually to *0*, and it is also used by the compiler to deduce the type used in the evaluation of the sum, and hence the return type.

So how might we use `accumulate()` to sum the `pan_slice_t` array? A naïve attempt might look like this:

```
void pantheios_log_n( int                   severity
                    , size_t                numSlices
                    , pan_slice_t const*    slices)
{
  . . .
  size_t len = std::accumulate(slices, slices + n, size_t(0)); // Error!
```

Unfortunately, if we try to use `accumulate()` with a sequence of `pan_slice_t`, we'll be promptly informed by the compiler that there's no + operator for the operands `unsigned int` and `pan_slice_t`. That's no surprise. C++ sensibly does not define arithmetic operators for nonfundamental, nonnumeric types. What we really want is to sum the `len` members in the sequence of `pan_slice_t` structures, not the structures themselves. To work out how to do this, we need to look at a typical implementation of `std::accumulate()`:

```
template <typename I, typename T>
T accumulate(I first, I last, T init)
{
  for(; first != last; ++first)
  {
    init += *first;
  }
  return init;
}
```

The point at which we need to handle the difference between what we've got and what's required is the application of the * to first. There are two options in this case: Use the four-parameter variant of accumulate() with a custom-written function, or use a different iterator type. The former option requires the user to define a function, as in:

```
size_t sum_slice(size_t total, pan_slice_t const& slice)
{
  return total + slice.len;
}
```

This would be used as follows:

```
void pantheios_log_n( int                 severity
                    , size_t              numSlices
                    , pan_slice_t const*  slices)
{
  . . .
  size_t len = std::accumulate(slices, slices + n, size_t(0)
                              , sum_slice);
```

There are two problems with this approach. First, it means every time we want to select a member from the elements of a range, we must write a function. This is hardly in the spirit of generic programming and is just a big drag. Pah!

The other reason is that it's slower. The timings (in milliseconds) shown in Table 38.1 were obtained for a program that calculates the sum of the lengths of 100,000 slices, repeated 200 times, using sum_slice() and using the member_selector_iterator, which we'll come to know and love shortly.

Table 38.1 Performance of Accumulation Using member_selector_iterator and Custom Function

Method	Code Warrior	Digital Mars	GCC	Intel 8	Visual C++ 7.1
sum_slice	170	246	229	146	152
member_selector_iterator	128	154	138	114	112

38.3 stlsoft::member_selector_iterator

What we need is an iterator adaptor that presents a different value type, based on selecting a member of the underlying iterator's value type: the member_selector_iterator. Before we look at its definition, let's look at it in action:

```
size_t totalLen = std::accumulate(
                          member_selector(slices, &pan_slice_t::len)
                        , member_selector(slices + n, &pan_slice_t::len)
                        , size_t(0));
```

member_selector() is a creator function (Section 5.1.3) that creates an instance of member_selector_iterator, based on the arguments of an iterator and a pointer to member. As with all creator functions, it obviates the need to explicitly parameterize the class template. The function looks like the following:

```
template< typename I  // Iterator type
        , class    C  // Class type
        , typename M  // Selected member type
        >
member_selector_iterator<I, C, M>
 member_selector(I iterator, M C::*member)
{
  return member_selector_iterator<I, C, M>(iterator, member);
}
```

Both creator function and class template are parameterized by the underlying iterator type, the type of the class, and the type of the class member. Fundamentally, the selector iterator, shown in Listing 38.1, is a very simple thing. It holds an instance of the underlying iterator, which it subjects to its own advancement operations (++, --, and so on), and a pointer to member used to dereference the underlying iterator.

Listing 38.1 Definition of member_selector_iterator

```
// In namespace stlsoft
template< typename I  // Iterator type
        , class    C  // Class type
        , typename M  // Selected member type
        >
class member_selector_iterator
  : public iterator<typename std::iterator_traits<I>::iterator_category
                  , M, ptrdiff_t
                  , M*, M&
                  >
{
public: // Member Types
  . . .
  typedef M&        reference;
  typedef M const&  const_reference;
  . . .
public: // Construction
  member_selector_iterator(I it, M C::*member)
    : m_it(it)
    , m_member(member)
  {}
  . . .
public: // Input Iteration Methods
  class_type& operator ++()
  {
    ++m_it;
    return *this;
  }
  reference operator *()
  {
    return (*m_it).*m_member;
  }
  const_reference operator *() const
  {
    return (*m_it).*m_member;
  }
  . . .
  bool equal(class_type const& rhs) const
  {
    assert(m_member == rhs.m_member);
    return m_it == rhs.m_it;
  }  . . .
private: // Member Variables
  I     m_it;
  M C::*m_member;
};
```

The constructor takes a copy of the iterator and the pointer to member. The preincrement operator, `operator ++()`, and all the other increment/decrement operators (not shown) defer to the underlying iterator, as does the `equal()` comparison method. The only nontrivial part of the implementation is the use of the member pointer `m_member` in the `operator *()` methods. The expression `(*m_it).*m_member` has the following meaning.

1. Dereference `m_it` to acquire a reference to the underlying iterator's value type.
2. Apply the `.*` pointer-to-member operator to the value type reference, to access (a reference to) the given member.

And that's the full picture for the iterator class. If that were everything to it, we could knock off early and go out for chocolate shakes and some `pan_slice` pizza with Chrysta. The problem comes when we try to apply this template.

38.4 Creator Function Woes

Let's enumerate the possible ways in which the iterator can be used:

1. Nonmutating access to a non-`const` array
2. Nonmutating access to a `const` array
3. Mutating access to a non-`const` array

4. Nonmutating access to a non-`const` collection with class-type iterators
5. Nonmutating access to a `const` collection with class-type iterators
6. Mutating access to a collection with class-type iterators

There is also the issue of selecting a `const` member of a class. We'll defer that for now and tackle these six cases.

A word of warning: The following subsections may have you wondering why anyone would engineer a solution in a world of such baffling inconsistencies, never mind inflict the minutiae on his poor unsuspecting readers. The reason is twofold. First, the payoff is worth it. `member_selector_iterator` is highly discoverable, easy to use, and highly efficient. Second, I believe the lessons I learned in this particular quest will be generally applicable to cases of nontrivial iterator adaptation, which you may, inspired by these humble 500+ pages, be inclined to have a go at. Whatever the rationale, hold on to your hat—there's a bumpy road ahead.

38.4.1 Nonmutating Access to a Non-`const` Array

This looks like the following:

```
struct S
{
  int i;
};
S as[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::copy(member_selector(&as[0], &S::i)
        , member_selector(&as[0] + n, &S::i)
        , std::ostream_iterator<int>(std::cout, " "));
```

All the compilers in my test set—Borland 5.6, CodeWarrior 8, Digital Mars 8.45, GCC 3.4, Intel 8, Visual C++ 6, and Visual C++ 7.1—work fine with this.

38.4.2 Nonmutating Access to a `const` Array

This looks like the following:

```
const S as[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::copy(member_selector(&as[0], &S::i)
        , member_selector(&as[0] + n, &S::i)
        , std::ostream_iterator<int>(std::cout, " "));
```

Now only Digital Mars, Intel, and (both versions of) Visual C++ compile. The others—which happen to be correct—complain that a reference to `const int` cannot be converted to a reference to `int` in the non-`const` dereference operator method:

```
  reference operator *()
  {
    return (*m_it).*m_member; // Error here!
  }
```

It may surprise you that this method is called, rather than the `const` version, because STL algorithms take iterators by (non-`const`) value. Therefore, there is a mismatch between the iterator type, `S const*`, and the type of the member, `int`. Dereferencing the iterator with `.*` yields `const int`, not `int`. The obvious tactic is to add a second function template, which has a more specialized iterator type, a `const` pointer:

```
template <typename C, typename M>
member_selector_iterator<C const*, C, const M>
 member_selector(C const* iterator, M C::*member)
{
  return member_selector_iterator<C const*, C, const M>(iterator
                                                       , member);
}
```

Now all compilers work apart from Digital Mars and Visual C++ 6. If we now attempt to compile both client forms together, with both the original and the pointer-to-`const` creator functions, again all compilers succeed in compiling except for Digital Mars and Visual C++ 6. So far, we can achieve portability if we just compile out the more specialized form with Digital Mars, and `#ifdef` out the non-`const` dereference operator for Visual C++ 6. Alas, there are more problems waiting around the corner.

38.4.3 Mutating Access to a Non-`const` Array

This looks like the following:

```
struct doubler
{
public:
  int operator ()(int i) const
  {
    return 2 * i;
  }
};
S as[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::transform( member_selector(&as[0], &S::i)
              , member_selector(&as[0] + n, &S::i)
              , member_selector(&as[0], &S::i)
              , doubler());
```

With our existing two creator functions, all compilers work with this, except for Visual C++ 6, which we can subsequently discount from any form of mutating member selection.

38.4.4 Nonmutating Access to a Non-`const` Collection with Class-Type Iterators

This looks like the following:

```
std::list<S>  lst = . . .
std::copy(member_selector(lst.begin(), &S::i)
        , member_selector(lst.end(), &S::i)
        , std::ostream_iterator<int>(std::cout, " "));
```

As long as we keep in mind that we don't provide a pointer-to-`const` version for Digital Mars and that we hide the non-`const` member version of the dereference operator for Visual C++, we're still okay with this.

38.4.5 Nonmutating Access to a `const` Collection with Class-Type Iterators

This looks like the following:

```
const std::list<S>  lst = . . .
std::copy(member_selector(lst.begin(), &S::i)
        , member_selector(lst.end(), &S::i)
        , std::ostream_iterator<int>(std::cout, " "));
```

Surprisingly, this seemingly innocuous change causes every single compiler except Digital Mars to balk. The problem is that we're back to the issue whereby the non-`const` dereference operator attempts to convert a reference-to-`const` into a reference-to-non-`const`. Unfortunately, the fix here is anything but simple because the iterators returned by the `const` versions of `begin()` and `end()` are *not* `const iterator`; rather, they are `const_iterator`. So we can't do something like this:

```
template <typename I, typename C, typename M>
member_selector_iterator<const I, C, const M>
 member_selector(const I iterator, M C::*member)
{
  return member_selector_iterator<const I, C, const M>(iterator
                                                      , member);
}
```

We need to deduce whether the iterator type manipulates `const` value types or non-`const`. Because we need to deduce a type, rather than just behavior, in order to formulate a return type, we must use *type detection* (Section 13.4.2); selection of specialization via function template overloading is not an option.

We need to achieve a way to have something examine the iterator type and from that deduce whether to return either `member_selector_iterator<I, C, M>` or `member_selector_iterator<I, C, const M>`. We've already seen how to do *type selection* (Section 13.4.1), so all we need is the compile-time value indicating whether we want `const` or non-`const`.

The first attempt might be to detect whether the iterator's value type is const, using std::iterator_traits and base_type_traits (Section 12.1.1), as in the following:

```
base_type_traits<typename std::iterator_traits<I>::value_type>::is_const
```

This would be used as shown in the following stupendous construction, using select_first_type_if (Section 13.4.1) and base_type_traits:

```
template <typename  I, class C, typename  M>
typename select_first_type_if< member_selector_iterator<I, C, const M>
      , member_selector_iterator<I, C, M>
      , base_type_traits<typename
                          std::iterator_traits<I>::value_type>::is_const
      >::type member_selector(I iterator, M C::*member)
{
  typedef typename
   select_first_type_if< member_selector_iterator<I, C, const M>
            , member_selector_iterator<I, C, M>
            , base_type_traits<typename
                          std::iterator_traits<I>::value_type>::is_const
            >::type iterator_t;
  return iterator_t(iterator, member);
}
```

Now if you can read that without wondering whether the world's gone mad, you're a better man (or woman) than I. This is awful stuff. Let's see if we can effect some simplification. How about this:

```
template <typename  I, class C, typename  M>
typename msi_traits<I, C, M>::type
 member_selector(I iterator, M C::*member)
{
  typedef typename msi_traits<I, C, M>::type  iterator_t;
  return iterator_t(iterator, member);
}
```

We've offloaded all that horrible type detection and selection to the generator template msi_traits, which keeps the complexity to a manageable level. Here's the definition, which breaks things down into member types to enhance readability:

```
template <typename  I, class C, typename  M>
struct msi_traits
{
private: // Member Types
  typedef member_selector_iterator<I, C, const M>   const_msi_type;
  typedef member_selector_iterator<I, C, M>         non_const_msi_type;
  typedef typename std::iterator_traits<I>::value_type
                                                       tested_member_type;
public:
  typedef typename select_first_type_if< const_msi_type
                    , non_const_msi_type
                    , base_type_traits<tested_member_type>::is_const
                                     >::type       type;
};
```

Unfortunately, although it did work for a couple of compilers, this is not an answer. The reason is that, as the standard prescribes (C++-03: 24.3.1;2), the specialization of `std::iterator_traits` for pointers-to-`const` takes the following form:

```
template <typename T>
struct std::iterator_traits<T const*>
{
  typedef random_access_iterator_tag  iterator_category;
  typedef T                           value_type;
  typedef T const*                    pointer;
  typedef T const*                    reference;
  typedef ptrdiff_t                   difference_type;
};
```

Clearly, testing the `const`-ness of the value type of a (`const`) pointer iterator would yield an invalid value. (I never investigated the reason that some compilers give desired, rather than correct, behavior with this test since it's not well founded. I suspected that some compilers' libraries may in fact define the `value_type` to be `const T`, but this is not the case, so it's probably just bugs in the compiler implementations. Whatever the case, we needn't concern ourselves further with it.)

Although testing `value_type` is not the solution, the definition of the partial specialization shown earlier gives a clue to a more promising solution. Since the `pointer` and `reference` members of the pointer-to-`const` specialization of `std::iterator_traits` *do* contain `const` information in their types, we can use one of them:

```
template <typename  I, class C, typename  M>
struct msi_traits
{
  . . .
  typedef typename std::iterator_traits<I>::pointer tested_member_type;
```

With this adjustment, we have far better coverage. CodeWarrior, GCC, Intel, and Visual C++ 7.1 all compile this without a second glance. Bearing in mind that Visual C++ 6 is already out of the picture, the only compilers that are troubled by this are Borland and Digital Mars. With Digital Mars, we can still just stick with the original function that did the right thing for the wrong reasons. But at this point we must consign Borland 5.6 to the Visual C++ 6 bin since there's just insufficient support for sophisticated template manipulation to devise a practicable way around this issue.

38.4.6 Mutating Access to a Collection with Class-Type Iterators

This looks like the following:

```
std::list<int>  lst = . . .
std::transform( member_selector(lst.begin(), &S::i)
              , member_selector(lst.end(), &S::i)
              , member_selector(lst.begin(), &S::i)
              , doubler());
```

Thankfully, all the work we've already done has paid off here. Again CodeWarrior, GCC, Intel, and Visual C++ 7.1 all work with this form, and Digital Mars continues to work with the single original simple creator function, so we can achieve very good support for member selection from these compilers by a single `#ifdef`. As for Borland 5.6 and Visual C++ 6, we can provide nonmutating access for them, and also allow mutating access for arrays for Borland, by providing appropriate `#ifdefs`. While not perfect, that's pretty good coverage for what we'd have to say is a relatively advanced technique.

38.4.7 Selecting `const` members

So far we've examined read-only and read/write access to non-`const` members of (sequences of) `const` and non-`const` classes. This includes `pan_slice_t`, whose `len` member is non-`const`. What we've not covered is how to deal with accessing `const` members of sequences of class types, such as the following:

```
struct cpan_slice_t
{
  const size_t      len;
  char const* const str;
};
```

We'll ignore the issue of how you would initialize and manipulate arrays of this type and focus on the ramifications of what will happen when we try to enumerate a sequence of non-`const cpan_slice_t` instances. The problem we might expect is that the deduction mechanism we've developed thus far will fail to require a `const` member type when the iterator is a non-`const` one.

```
struct CS
{
  const int i;
};
CS acs[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
std::copy(member_selector(&acs[0], &CS::i)
        , member_selector(&acs[0] + n, &CS::i)
        , std::ostream_iterator<int>(std::cout, " "));
```

Thankfully, this is all taken care of by the existing implementation. The reason is that when we take the address of `CS::i`, we create a pointer of type `const int C::*`; hence `M` is deduced to be `const int`, not `int`. Since the `M` and `I` template parameters to the creator function are essentially unrelated, the non-`const`-ness of the iterator (type `I`) does not override or otherwise interfere with the `const`-ness of `M`. So it all just works. *Phew!*

38.5 Summary

Well, this chapter has certainly been one of contradictions. We've shown how, in use, the `member_selector_iterator` strongly upholds the principles of *Composition* and *Diversity* and offers a highly discoverable interface, engendering transparent client code. However, we've seen that in order to support this apparent simplicity, we must jump through a great many hoops and have an implementation that suffers considerably in transparency in most of the creator functions. It's not perfect—anyone, me included, trying to go back and make significant changes to the implementation would have a bad day—but sometimes compromises are the best one can hope for. Given what `member_selector_iterator` delivers to its users, I am content in what has been achieved.

38.6 On the CD

The CD contains a section giving even more horrifying details on handling inconsistencies between compilers and libraries.

CHAPTER 39

C-Style String Concatenation

I may not agree with what you say but I shall defend your right to say it to the death.
—Voltaire

I too was concocting this very same plan; already our minds are becoming one!
—Puss-in-boots, *Shrek 2*

39.1 Motivation

In the previous chapter we looked at how the **Pantheios** library core used `member_selector_iterator` for calculating the total logging entry string length from the array of string slices (`pan_slice_t const*`), via the standard library algorithm `accumulate()`. In this chapter we're going to look at the other major part of the logging operation: concatenation of the string slices into the allocated buffer, using the standard library algorithm `copy()`. Let's dive straight in and look at the function implementation (with `stlsoft` namespace qualifications, contract enforcements, and error checking elided):

```
int pantheios_log_n(pan_sev_t           severity
                  , pan_slice_t const*  slices
                  , size_t              numSlices)
{
  // 1. Determine length required
  size_t  n = std::accumulate(member_selector(slices, &pan_slice_t::len)
                , member_selector(slices + numSlices, &pan_slice_t::len)
                , size_t(0));
  // 2. Allocate memory
  auto_buffer<char, 2048>   buffer(1 + n);
  // 3. Write the strings into the buffer
  std::copy(slices, slices + numSlices
          , cstring_concatenator(&buffer[0]));
  // 4. Nul-terminate the buffer
  buffer[n] = '\0';
  // 5. Pass to back end
  return pantheios_be_logEntry(. . ., severity, &buffer[0], n);
}
```

Step 1 was covered in the previous chapter on `member_selector_iterator` and its creator functions. Step 2 ensures that sufficient memory is available, via `auto_buffer` (Section 16.2). Steps 4 and 5 are pretty self-explanatory. That leaves step 3.

The statement is a call to the `std::copy()` algorithm, passing in iterators denoting the string slice array, along with whatever the creator function `cstring_concatenator()` returns. We know that this type must be a viable output iterator (Section 1.3.2), and I can tell you that it's an instance of the subject of this chapter, `stlsoft::cstring_concatenator_iterator`.

39.2 An Inflexible Version

One notable difference from step 1 is that we're not selecting members, à la `member_selector<slices, &pan_slice_t::ptr)`; we're just using the `pan_slice_t` types as a whole. Looking at this, you might assume that the `cstring_concatenator_iterator` is specifically written to manipulate the type `pan_slice_t`. Such a definition might look like that shown in Listing 39.1. Although not the actual definition of `cstring_concatenator_iterator`, several features are in common with the actual implementation, so we'll examine it carefully. The **`Member Types`** section is pretty standard for an output iterator, with most types being left to the parent class specialization of `std::iterator`. Since it's an output iterator, I employ the ***Dereference Proxy*** pattern (Chapter 35).

Listing 39.1 Putative Definition of `cstring_concatenator_iterator`

```
class cstring_concatenator_iterator
  : public std::iterator<output_iterator_tag, void, void, void, void>
{
public: // Member Types
  typedef char                          char_type;
  typedef cstring_concatenator_iterator class_type;
private:
  class deref_proxy;
  friend class deref_proxy;
public: // Construction
  explicit cstring_concatenator_iterator(char* s)
    : m_dest(s)
  {
    STLSOFT_ASSERT(NULL != s);
  }
public: // Output Iterator Operations
  class_type& operator ++();
  class_type& operator ++(int);
  deref_proxy operator *();
private: // Implementation
  class deref_proxy
  {
  public: // Construction
    deref_proxy(cstring_concatenator_iterator* it)
```

```
      : m_it(it)
    {}
  public: // Assignment
    void operator =(pan_slice_t const& slice)
    {
      m_it->invoke_(slice);
    }
  private: // Member Variables
    cstring_concatenator_iterator* const m_it;
  private: // Not to be implemented
    void operator =(deref_proxy const&);
  };
private: // Implementation
  void concat_(char const* s, size_t len)
  {
    stlsoft::copy_n(m_dest, s, len);
    m_dest += len;
  }
  void invoke_(pan_slice_t const& slice)
  {
    this->concat_(slice.ptr, slice.len);
  }
private: // Member Variables
  char_type*  m_dest;
};
```

When an instance of the iterator is dereferenced, it produces an instance of `deref_proxy`, and when that is assigned an instance of `pan_slice_t`, it calls `cstring_concatenator_iterator::invoke_()`. This, in turn, calls `concat_()`, passing the slice's pointer and length. (The reason for placing the actions into the separate `concat_()` method will become clear as we progress.) `concat_()` then copies each element in the range from the source string to the destination buffer pointer passed to the iterator's constructor, using the `stlsoft::copy_n()` algorithm, and advances the destination buffer pointer the appropriate number of characters. (`copy_n()` was an omission from the C++-98 standard and will be included in C++-0x. It is already provided in some standard library distributions, but I use the **STLSoft** version to maximize portability. This and many other algorithms will be discussed in Volume 2.)

In this way, each subsequent assignment appends to the next space available, producing a contiguous output, as the following example illustrates:

```
char                                buff[12];
cstring_concatenator_iterator<char> cci(&buff[0]);

*cci = pan_slice_t("Black", 5);
*cci = pan_slice_t(" ", 1);
*cci = pan_slice_t("Grape", 5);

assert(0 == ::strncmp(&buff[0], "Black Grape", 11));
```

However, there's an obvious problem with the implementation: It's tied to the pan_slice_t type. As such, it's anything but a good bit of reusable code. Maybe the correct choice is to handle just C-style strings (i.e., char const*)? It is a C-string concatenator iterator, after all.

In that case, the definition of cstring_concatenator_iterator::invoke_() would be along the following lines:

```
  void invoke_(char const* s)
  {
    this->concat_(s, ::strlen(s));
  }
  . . .
```

The statement of step 3 would then need to use the member_selector_iterator to select the ptr member of the slices, before they were passed to the output iterator:

```
std::copy(member_selector(slices, &pan_slice_t::ptr)
        , member_selector(slices + numSlices, &pan_slice_t::ptr)
        , cstring_concatenator(&buffer[0]));
```

Alas, this will fail. As discussed in Section 9.3.1, the **Pantheios** application-layer function templates use the **c_str_data_a** and **c_str_len_a** shims, rather than **c_str_ptr** (or **c_str_ptr_a**), to instantiate a pan_slice_t instance for a given logged variable. This facilitates greater efficiency in cases where the logged types store a contiguous array of characters representing their contents but do *not* nul-terminate it. The c_str_ptr_a() functions for such types do not need to allocate memory, copy contents and nul-terminate them, and then deallocate the memory after the shim is called, which has obvious performance implications.

Thus, there's no guarantee that a given pan_slice_t::ptr will point to a nul-terminated C-style string, so strlen() will yield invalid values. At best, there will be benign overwrites; at worst, the code will crash. The definition of cstring_concatenator_iterator would be limited to working with sequences of types that are not implicitly convertible to C strings (char*, char[], char const*, const char[], **MFC**'s CString, and so on). Since implicit conversions, especially to char const*, are generally a poor idea, this would hardly be a recommendable feature. Thankfully, there's a better, more generic way: using string access shims (Section 9.3.1), of course!

39.3 stlsoft::cstring_concatenator_iterator

The final definition of the iterator (Listing 39.2) includes only the salient differences from the earlier definition. The class provides an optional facility for having the iterator(s) keep a running total of the number of characters written into the destination buffer, which can be useful when you wish to insert additional character sequences into the destination at particular points in the iteration. (It also affords the use of contract programming [Chapter 7] verification on the client side to assert that the number written is as expected.)

Listing 39.2 **Final Definition of** `cstring_concatenator_iterator`

```
template <typename C>
class cstring_concatenator_iterator
  : public std::iterator<output_iterator_tag, void, void, void, void>
{
public: // Member Types
  typedef C                                char_type;
  typedef cstring_concatenator_iterator<C>  class_type;
  . . .
public: // Construction
  explicit cstring_concatenator_iterator(char_type* dest
                                        , size_t*    pNumWritten = NULL)
    : m_dest(dest)
    , m_numWritten((NULL != pNumWritten) ? pNumWritten : dummy_())
  {
    *m_numWritten = 0;
  }
public: // Output Iteration Methods
  . . .
private: // Implementation
  class deref_proxy
  {
    . . .
  public: // Assignment
    template <typename S>
    void operator =(S const& s)
    {
      m_it->invoke_(s);
    }
    . . .
  };
private: // Implementation
  void concat_(char_type const* s, size_t len)
  {
    std::copy_n(m_dest, s, len);
    m_dest        +=  len;
    *m_numWritten +=  len;
  }
  template <typename S>
  void invoke_(S const& s)
  {
    // 'Fire' the shims, and pass to the proxied iterator instance
    this->concat_(::stlsoft::c_str_data(s), ::stlsoft::c_str_len(s));
  }
  static size_t*  dummy_()
  {
    static size_t   s_dummy;
    return& s_dummy;
  }
private: // Member Variables
  char_type*  m_dest;
  size_t*     m_numWritten;
};
```

Both `cstring_concatenator_iterator::deref_proxy::operator =()` and `cstring_concatenator_iterator::invoke_()` are function templates that take a single argument of a type, assumed to be a string or representable as a string. It is now clear why the `invoke_()` method elicits the string pointer and length from the argument passed to `deref_proxy::operator =()` before calling `concat_()`. This is a case of a general pattern whereby the return values from shim functions are preserved in the statement of a function call, observing the rule about the use of the return values from access shims (Section 9.3). It also saves the cost of invoking the shim functions multiple times in cases where the resultant values are required more than once. Such costs can be considerable when they return temporary instances of a string type representing the converted value.

Tip: Use worker functions to ensure that shims are invoked only once per logical operation.

Again, with relatively straightforward use of string access shims, we are afforded a huge leap in flexibility without costing anything in performance, robustness, or discoverability, and with only a minor cost in component transparency. `cstring_concatenator_iterator` can now be used with *any* type for which the **c_str_data** and **c_str_len** shims are defined and will faithfully concatenate exactly the right number of characters (as specified by `c_str_len()`) regardless of whether the C-style string returned by `c_str_data()` is nul-terminated or not.

39.4 Creator Functions

All that remains is to define the creator function template taking a mutating (non-`const`) pointer to a character buffer into which the results will be written. When I first added the facility to retrieve the running total, I added a second overload, as follows:

```
template <typename C>
cstring_concatenator_iterator<C> cstring_concatenator(C* s)
{
  return cstring_concatenator_iterator<C>(s, NULL);
}
template <typename C>
cstring_concatenator_iterator<C>
 cstring_concatenator(C* s, size_t& numWritten)
{
  return cstring_concatenator_iterator<C>(s, &numWritten);
}
```

Note that numWritten in the second overload is a reference to size_t, rather than a pointer, whose address is taken and passed to the iterator constructor. In general, this is a useful technique when you wish to be sure that a null pointer must not be specified to the underlying function or constructor. However, in this case, a null pointer is entirely acceptable to the iterator constructor, so the additional complexity was unnecessary and misleading. This was refactored to a single creator function:

```
template <typename C>
cstring_concatenator_iterator<C>
 cstring_concatenator(C* s, size_t* pNumWritten = NULL)
{
  return cstring_concatenator_iterator<C>(s, pNumWritten);
}
```

This is also better for client code. It's more obvious that numWritten will be subject to modification in this:

```
std::copy(. . . , stlsoft::cstring_concatenator(&result[0]
                                                , &numWritten));
```

than in this:

```
std::copy(. . . , stlsoft::cstring_concatenator(&result[0]
                                                , numWritten));
```

Tip: When a null pointer is acceptable, prefer using pointer parameters to clearly indicate mutability in client code.

39.5 Summary

Given the definition of cstring_concatenator_iterator (and its creator function, cstring_concatenator()), we can see how pantheios_log_n() supports the required functionality of the **Pantheios** function templates with maximum flexibility. Step 1 (Section 38.3) calculates the length of the required buffer by a single pass of the slices array. Step 2 provides up to 2,048 characters of stack storage and therefore allocates from the heap only in the case where the total required length is 2,048 or more. Step 3 concatenates all the string slices (which may not be nul-terminated) into the allocated buffer in a single pass, without any conversions (since c_str_data() for pan_slice_t simply returns the ptr member). Step 4 nul-terminates the string, and Step 5 passes this off to the back-end log function, which is implemented in an external library. The string is nul-terminated and the length specified to afford maximum simplicity in the back-end libraries.

I submit that the function body of `pantheios_log_n()` is an excellent example of STL extensions put to good use. There is no extraneous code and the algorithm invocations indicate the intent and mechanisms in a clear and transparent manner.

39.6 On the CD

The CD contains a merry old tale of a compiler that generates bad object code and the cheeky little remedy.

CHAPTER 40

String Object Concatenation

And above all things, never think that you're not good enough yourself. A man should never think that. My belief is that in life people will take you at your own reckoning.
— Isaac Asimov

Luck favors the prepared!
—Edna Mole

40.1 Introduction

In several of my projects I have had occasion to tokenize out string fragments from composite strings and then package them up again in a different form. We saw in Chapter 27 how tokenization can be effected easily and efficiently. Using the component described here, the tokens can be just as easily packaged up again.

Whereas we looked at how to write an output iterator for C-style string concatenation in the previous chapter, now we're going to look at the more object-oriented side of that equation: concatenation into a string instance. The two implementations share several features, which I won't discuss again, but there are some notable differences, which form the subject of this chapter.

40.2 `stlsoft::string_concatenator_iterator`

The simple version of this iterator is shown in Listing 40.1.

Listing 40.1 First Version of `string_concatenator_iterator`

```
template< typename S  // The type of string the iterator writes into
        , typename D  // The type of the delimiter
        >
class string_concatenator_iterator
  : public std::iterator<std::output_iterator_tag
                        , void, void, void, void>
{
public: // Member Types
  typedef S                                    string_type;
  typedef D                                    delimiter_type;
  typedef string_concatenator_iterator<S, D>   class_type;
private:
  class deref_proxy;
  friend class deref_proxy;
public: // Construction
  string_concatenator_iterator(string_type&           s
                             , delimiter_type const& delim)
    : m_s(s)
    , m_delim(delim)
  {}
public: // Output Iterator Operations
  deref_proxy operator *()
  {
    return deref_proxy(this);
  }
  class_type& operator ++();    // Returns *this
  class_type& operator ++(int); // Returns *this
private: // Implementation
  class deref_proxy
  {
  public: // Construction
    deref_proxy(string_concatenator_iterator* it)
      : m_it(it)
    {}
  public: // Assignment
    template <typename S3>
    void operator =(S3 const& value)
    {
      m_it->invoke_(value);
    }
  private: // Member Variables
    string_concatenator_iterator* const   m_it;
  };
  template <typename S3>
  void invoke_(S3 const& value)
  {
    if(0 != c_str_len(m_str))
    {
      m_str += c_str_ptr(m_delim);
    }
    m_str += c_str_ptr(value);
  }
private: // Member Variables
  string_type&          m_str;
  delimiter_type const& m_delim;
};
```

Naturally, `string_concatenator_iterator` comes with the obligatory creator function:

```
template <typename S, typename D>
string_concatenator_iterator<S, D>
```

```
 string_concatenator(S& s, D const& delim)
{
  return string_concatenator_iterator<S, D>(s, delim);
}
```

At first glance this seems like an eminently straightforward output iterator. It specifies the requisite member types, its pre- and postincrement operators return `*this`, and it uses the ***Dereference Proxy*** pattern (Chapter 35). As we saw with `cstring_concatenator_iterator` (Section 39.3), the `deref_proxy::operator =()` method is a function template that passes its argument to `string_concatenator_iterator::invoke_()`, which is also a function template.

It is in this method, `invoke_()`, where string concatenation is effected, via three *string access shim* (Section 9.3.1) functions. First, **c_str_len** is used to determine whether the destination string, `m_str`, currently has any content. If it does, a copy of the delimiter, held by the member variable `m_delim`, is appended to it by one overload of `c_str_ptr()` before the `value` parameter is appended. Finally, `value` is appended by another `c_str_ptr()` overload. (If the types of the destination string and the delimiter are the same, it will actually be the same overload.) In this way, a string such as *`abc,def,gh,i,j,klmn`* can be built up without an extraneous leading or trailing delimiter, which (in my opinion) is the form in which people generally prefer to see their composite strings.

The creator function `string_concatenator()` completes the picture, facilitating succinct and transparent client code statements for string concatenation, as in the following:

```
char const*   strings[] =
{
    "abc"
  , "defg"
  , "h"
  , "ijklm"
};
std::string   result;

std::copy(&strings[0], &strings[0] + STLSOFT_NUM_ELEMENTS(strings)
        , stlsoft::string_concatenator(result, ""));

assert("abcdefghijklm" == result);
```

It can be used in combination with `string_tokeniser` (Section 27.6) and a unary function object to effect transformations of the tokens in a string, as in this snippet from the command-line processing module of one of my development tools:

```
stlsoft::string_tokeniser<string_t, char> notTokens((*it).second, ',');
string_t                                  translatedItems;
std::transform( notTokens.begin(), notTokens.end()
              , stlsoft::string_concatenator(translatedItems, ",")
              , not_hyphen()); // Adds leading '-' or removes if there
```

```
(*it).second.swap(translatedItems);
```

This would change the string "*bc56,-cw8,dm,gcc34,-vc6,vc7*" into "*-bc56, cw8,-dm,-gcc34,vc6,-vc7*".

40.3 Heterogeneity of String Types

There are actually up to three distinct string types involved in the iterator. First, there's the type of the destination string, `S`. Second is the type of the delimiter, `D`. Third is the type of the element passed to the assignment operator (of `deref_proxy`). Because `invoke_()` uses the **c_str_ptr** string access shim, these three types can be all the same, all different, or any combination thereof. We can even use the iterator with apparently unimaginable combinations, as in the following mix of **MFC**, **STL,** and **STLSoft** code:

```
typedef CArray<COleVariant, COleVariant const&>   VariantArray_t;

VariantArray_t  ar;
CComboBox&      wnd = . . . // An edit window containing the delimiter
CString         result;

ar.Add(COleVariant("Space");
ar.Add(COleVariant(long(1999));
ar.Add(COleVariant("- More like it's");
ar.Add(COleVariant(COleDateTime(2005, 12, 23, 13, 14, 52));

// Use CArray instance adaptor (Section 24.7), so can use ar as STL seq
mfcstl::CArray_iadaptor<CArray<COleVariant, COleVariant const&>
                      >   arr(ar);

std::copy(  arr.begin(), arr.end()
        ,   stlsoft::string_concatenator(result, wnd));

std::cout << static_cast<char const*>(result) << std::endl;
```

As long as the headers that declare the string access shims for **COM** (*<comstl/shims/access/string.hpp>*) and **MFC** (*<mfcstl/shims/access/ string.hpp>*) types are included in the compilation, this incredible-looking code will compile and run successfully. Assuming the combo box contains a space in its text field, the code will output something like "*Space 1999 - More like it's 23/12/2005 1:14:52 pm*". Which, you have to admit, is a lot of flexibility: `VARIANT`s, window handles, and `CString`s all working together seamlessly with a generic component is about as much as you could ask.

40.4 But . . .

Naturally, things are never as simple as they seem. If we look closely at the implementation shown in Listing 40.1, we can see a couple of problems.

40.4.1 Assignability

For one thing, the iterator as defined does not satisfy the requirements of an output iterator: It's not *Assignable*. This is because I've followed my instincts and made the member variables references. But as we all know from C++ 101, references cannot be reassigned. This means that the compiler cannot implicitly generate a copy assignment operator, and that, my dear friends, means that those standard libraries that use concept checking will dutifully report its failure to come up to scratch and will fail to compile it.

Before we deal with this problem, we need to look at a second issue since the solution to both comes together in a single set of modifications.

40.4.2 Dangling References

The other issue is that, as a nonmutating (`const`) reference, the `m_delim` member can become stale without our noticing it. This issue was discussed in Section 34.3.2 in reference to `ostream_iterator`, in which case we opted for taking a copy because the iterator is to be used with the **IOStreams**, and the **IOStreams** are, er, well, not very fast. But in this case, it's quite likely that we will want to concatenate strings in circumstances where performance is at a premium, and where taking a copy of the delimiter might seem gratuitous just to avoid a bug that will manifest only when the iterator is used in an anti-idiomatic manner.

40.4.3 Solution

The answer to both problems lies in two simple measures, as shown in the final implementation (Listing 40.2).

Listing 40.2 Final Implementation of `string_concatenator_iterator`

```
template <typename S, typename D>
class string_concatenator_iterator
  : public std::iterator<std::output_iterator_tag
                        , void, void, void, void>
{
public: // Member Types
  . . .
private: // Construction
  string_concatenator_iterator(string_type*          s
                              , delimiter_type const* delim)
    : m_s(s)
    , m_delim(delim)
  {}
public:
  static class_type create(string_type& s, delimiter_type const& delim)
  {
    return class_type(&s, &delim);
  }
public: // Output Iterator Operations
  . . .
private: // Implementation
  class deref_proxy
  {
    . . . // No change from previous version
  };
  template <typename S3>
  void invoke_(S3 const& value)
  {
    if(0 != c_str_len(*m_str))
    {
      *m_str += c_str_ptr(*m_delim);
    }
    *m_str += c_str_ptr(value);
  }
private: // Member Variables
  string_type*          m_str;
  delimiter_type const* m_delim;
};

template <typename S, typename D>
string_concatenator_iterator<S, D>
 string_concatenator(S& s, D const& delim)
{
  return string_concatenator_iterator<S, D>::create(s, delim);
}
```

First, the member variables are changed to pointers. This affects only the constructor and the `invoke_()` method. Second, the constructor is made private and a public static `create()` method added, which is invoked by the creator function. To be sure, this does not make it impossible to declare nontemporary instances of the iterator, but it does mean that to do so the user would have to call `create()`, and we can reasonably assume that this would give him or her pause, sufficient to read the documentation for the class and consider the rightness of the intended use.

And that's it. Now the iterator behaves as it should, and with all the flexibility we've demonstrated that it should provide.

40.5 Summary

We've seen how with just a little effort and the use of string access shims, we can make a very flexible component to facilitate string concatenation via standard algorithms. The component, `stlsoft::string_concatenator_iterator`, fulfills the principles of *Composition*, *Least Surprise*, *Modularity*, and *Transparency*. It provides a highly discoverable interface—just a single simple function call—and facilitates good transparency in client code.

CHAPTER 41

Adapted Iterators Traits

That's a sinewy and complicated question with ramifications that would take more time than we have to explore adequately. Suppose you break off the fragment that most puzzles you and phrase that concisely?

—Jonathon Hemlock

This is fun! Are you mad?

—Arnold J. Rimmer, *Red Dwarf*

41.1 Introduction

In order to successfully navigate the remaining chapters of Part III, we are going to need some powerful tools to help us on our way. In the next chapter, we'll look at how to apply a selection predicate to filter out elements of the underlying range. And an extra chapter, Indexed Iteration, included on the CD looks at the business of indexing iterators: applying an adaptation that associates an ordinal representing the iteration state with an iterator, while preserving the base iterator interface, type characteristics, and behavior. In both cases we will need to be able to detect, manipulate, and re-present aspects of the base iterator type.

Consequently, this chapter presents the `adapted_iterator_traits` class template from the **STLSoft** libraries. This chapter is long on complexity, high on past frustrations (mine), and, alas, rather short on fun. If you don't wish to immerse yourself in the morass that is type adaptation and its smelly counterpart, nonstandard standard library features, feel free to skip this chapter and come back and use it for reference to inform you on the material in the later chapters.

41.2 `stlsoft::adapted_iterator_traits`

Still with me? Good. I'm not going to present any motivating material here, as that's adequately handled in subsequent chapters, so I'm just going to present the traits to you as they are currently defined. I will, however, present only a simplified form first, one that does not cater to nonstandard standard libraries, in order to give you the best chance at divining the main purpose and function of the class. The extra material on the CD examines all those *really* ugly bits. (Try this tongue twister: "Taking a stand on standardizing nonstandard standard libraries stands to leave you in misunderstanding.")

The first thing to note is that there's not a single executed function involved in this class. Although it is reasonably large, every aspect of it is compile-time template-type deduction. It is, in every sense, metaprogramming.

We saw in Chapter 13 the use of the *inferred interface adaptation* (IIA) mechanism and its three constituent techniques of *type detection*, *type fixing*, and *type selection*. IIA is the primary technique by which the adapted iterator traits class achieves its purpose. Listing 41.1 shows a trivialized form of the class template.

Listing 41.1 Outline Definition of `adapted_iterator_traits`

```
// In namespace stlsoft
template <typename I>
struct adapted_iterator_traits
{
  typedef ????      iterator_category;
  typedef ????      value_type;
  typedef ????      difference_type;
  typedef ????      pointer;
  typedef ????      reference;
  typedef ????      const_pointer;
  typedef ????      const_reference;
  typedef ????      effective_reference;
  typedef ????      effective_const_reference;
  typedef ????      effective_pointer;
  typedef ????      effective_const_pointer;
};
```

Iterator adaptors specialize `adapted_iterator_traits` in terms of their base iterator types and use its member types to define their own member types. The first five are the usual member types we associate with iterator traits. We've already seen `effective_reference` and `effective_const_reference` in the `transform_iterator` adaptor class template in Section 36.4. The purpose of the other four will be revealed shortly.

In just the same way as `std::iterator_traits`, `adapted_iterator_traits` is partially specialized for pointers (and various *cv*-qualifier flavors thereof), as shown in Listing 41.2.

Listing 41.2 Partial Specializations of `adapted_iterator_traits` **for Pointer Types**

```
template <typename T>
struct adapted_iterator_traits<T*>
{
    typedef std::random_access_iterator_tag iterator_category;
    typedef T                               value_type;
    typedef ptrdiff_t                       difference_type;
    typedef value_type*                     pointer;
    typedef value_type const*               const_pointer;
    typedef value_type&                     reference;
    typedef value_type const&               const_reference;
    typedef reference                       effective_reference;
    typedef const_reference                 effective_const_reference;
    typedef pointer                         effective_pointer;
    typedef const_pointer                   effective_const_pointer;
};
template <typename T>
struct adapted_iterator_traits<T const*>
{
    . . . // All other members same as <T*> specialization
    typedef value_type const*              pointer;
    typedef value_type const*              const_pointer;
    typedef value_type const&              reference;
    typedef value_type const&              const_reference;
};
template <typename T>
struct adapted_iterator_traits<T volatile*>
{
    . . . // All other members same as <T*> specialization
    typedef value_type volatile*           pointer;
    typedef value_type volatile const*     const_pointer;
    typedef value_type volatile&           reference;
    typedef value_type volatile const&     const_reference;
};
template <typename T>
struct adapted_iterator_traits<T const volatile*>
{
    . . . // All other members same as <T*> specialization
    typedef value_type volatile const*     pointer;
    typedef value_type volatile const*     const_pointer;
    typedef value_type volatile const&     reference;
    typedef value_type volatile const&     const_reference;
};
```

There are no surprises here, though there a couple of points are worth noting. First, all variants define `effective_reference` as the same type as `reference`, and `effective_const_reference` as `const_reference`. This is because pointers, being *contiguous iterators* (Section 2.3.6), never exhibit an element reference category of *by-value temporary* (Section 3.3.5) or *void* (Section 3.3.6). (If there's something to point at, it can't very well be representable only as a temporary instance.)

Rule: Contiguous iterators never exhibit an element reference category of by-value temporary or void.

Second, the `const` variants always define their `pointer` and `reference` member types identically. In other words, `pointer` is a pointer-to-`const`, and `reference` is a reference-to-`const`. It is the deduction of sensible definitions for these same types for *non*pointer iterators, in the primary template, that is the big challenge. For the purposes of exposition, I will present the primary template a piece at a time.

41.2.1 iterator_category

This piece is very simple. Every nonpointer iterator type must define a member iterator_ category, so this first element is defined as follows:

```
template <typename I>
struct adapted_iterator_traits
{
  typedef typename I::iterator_category      iterator_category;
  . . .
```

41.2.2 value_type

The same thing applies here. Any type that doesn't have a value_type member type cannot be reasonably interpreted as an iterator, so:

```
  typedef typename I::value_type             value_type;
  . . .
```

41.2.3 difference_type

Here's where things start to get a little complicated. Some older libraries do not define a difference_type in their iterators, either defining no relevant member at all or defining it as distance_type. This is because they were crafted at a time when what became C++-98 was taking shape, so there's no criticism to be made here. But we do need to handle the issue.

First, we need to know whether the iterator type has the difference_type member type. This is achieved via type detection (Section 13.4.2), as follows:

```
private:
  enum { HAS_MEMBER_DIFFERENCE_TYPE = has_difference_type<I>::value };
  . . .
```

In anticipation of the member type being missing, we define a putative difference_type member, via type fixing (Section 13.4.3), using the cunningly named type fixer type for difference_type, fixer_difference_type:

```
private:
  enum { HAS_MEMBER_DIFFERENCE_TYPE = has_difference_type<I>::value };
  typedef typename fixer_difference_type< I
                          , HAS_MEMBER_DIFFERENCE_TYPE
                          >::difference_type  putative_difference_type_;
  . . .
```

Note that the type is private and it has the trailing underscore, denoting that it's an internal worker type only, in order to minimize the chances of clashes with other metaprogramming techniques. That's just my convention and you may well choose a different way to represent it. Just take care not to leak out into public those types that do not belong there.

The final step in this case is to select the type that will be used to define `difference_type`, via type selection (Section 13.4.1):

```
private:
  enum { HAS_MEMBER_DIFFERENCE_TYPE = has_difference_type<I>::value };
  typedef typename fixer_difference_type< I
                        , HAS_MEMBER_DIFFERENCE_TYPE
                        >::difference_type  putative_difference_type_;
  . . .
public:
  typedef typename select_first_type_if< putative_difference_type_
                        , ptrdiff_t
                        >::type             difference_type;
  . . .
```

And that's it! Regardless of the actual type of the iterator, if it has a `difference_type` member, that will be used; otherwise, a sensible default of `ptrdiff_t` will be used.

What we've seen here is metaprogramming IF-THEN-ELSE. Discrimination of all remaining member types uses this same technique and is more complex than this case only by virtue of the often multiple elements comprising the IF.

41.2.4 `pointer`

Excepting nonstandard older standard library implementations, if an iterator does not have a `pointer` member type, it's only safe to infer that it has by-value temporary element reference category and therefore to define its `pointer` member type to be `void`. Since the degenerate type of a type fixer is `void`, we need only perform type detection and type fixing, as in the following:

```
private:
  enum { HAS_MEMBER_POINTER = has_pointer<I>::value };
  typedef typename fixer_pointer< I
                        , HAS_MEMBER_POINTER
                        >::pointer          putative_pointer_;
  . . .
public:
  typedef putative_pointer_                 pointer;
  . . .
```

41.2.5 `reference`

For reasons that are too complex even for this chapter, compilers (apart from some that go above and beyond the call of duty) are unable to detect member types `reference` and `const_reference`. Thankfully, there are, to the best of my knowledge, no standard library implementations (however nonstandard they might be) that define iterators with member type `pointer` but without member type `reference`, or vice versa. Therefore, we can simply borrow the value of `HAS_MEMBER_POINTER` to infer the presence/absence of `reference`:

```
private:
  enum { HAS_MEMBER_POINTER   = has_pointer<I>::value };
  enum { HAS_MEMBER_REFERENCE  = HAS_MEMBER_POINTER         };
  typedef typename fixer_reference< I
                        , HAS_MEMBER_REFERENCE
                        >:: reference       putative_reference_;
  . . .
public:
  typedef putative_reference_                reference;
  . . .
```

41.2.6 `const_pointer` and `const_reference`

This is where things get a little tricky. We want to be able to appropriately and correctly define `const_pointer` and `const_reference` for any iterator type. If that iterator type has the by-value temporary (or void) element reference category, the member type would be `void`. But for iterators of other element reference categories, we would need to define them in terms of the value type. Thus, we need to know the element reference category of the iterator. As we've seen throughout Parts II and III, iterators of these two lowest element reference categories are distinguished by having the `pointer` member type of type `void`. Those of the void element reference category also have a `value_type` of type `void`. Hence, we can define member values denoting the category, using the type equivalence component `is_same_type` (Section 12.1.7), as follows:

```
private:
  enum { REF_CAT_IS_VOID  = is_same_type<value_type, void>::value };
  enum { REF_CAT_IS_BVT   = !REF_CAT_IS_VOID &&
                             is_same_type<pointer, void>::value    };
  . . .
```

With this knowledge, we can now appropriately define the `const_pointer` and `const_reference` members, via type selection:

```
public:
  typedef typename select_first_type_if<void
                        , value_type const*
                        , REF_CAT_IS_VOID || REF_CAT_IS_BVT
                        >::type              const_pointer;
  typedef typename select_first_type_if<void
                        , typename add_const_ref<value_type>::type
                        , REF_CAT_IS_VOID || REF_CAT_IS_BVT
                        >::type              const_reference;
  . . .
```

The `add_const_ref` type generator is used to apply `const&` to any type except `void`, which, of course, it leaves unmodified. This is required since the language does not allow a reference to `void`.

41.2.7 effective_reference and effective_const_reference

Recall from Section 36.4.7 that, when we looked at the return types of the dereference and subscript operators for transform_iterator, the effective_reference and effective_const_reference member types should be void for iterators with an element reference category of void, value_type and const value_type for those with an element reference category of by-value temporary, and value_type& and value_type const& for those with higher element reference categories. Hence, we use type selection one last time, as follows:

```
public:
  typedef typename select_first_type_if<value_type
                         , reference
                         , REF_CAT_IS_BVT
                         >::type            effective_reference;
  typedef typename select_first_type_if<
                           typename add_const<value_type>::type
                         , const_reference
                         , REF_CAT_IS_BVT
                         >::type            effective_const_reference;
  . . .
```

Note that there's no additional discrimination or definition of intermediate types. We have to test only for by-value temporary since the value type for iterators of void element reference category is void. The only addition is the use of the add_const type generator, which adds a const qualifier to any type other than void on which it is specialized. This is required since it's illegal to const-qualify void.

41.2.8 effective_pointer and effective_const_pointer

This leaves the effective_pointer and effective_const_pointer member types, which are used for the return types of the member selection operators. They follow a structure similar to that for the effective references, except that the type of both in the case of the by-value temporary element reference category is void.

```
public:
  typedef typename select_first_type_if<void
                         , pointer
                         , REF_CAT_IS_BVT
                         >::type            effective_pointer;
  typedef typename select_first_type_if<void
                         , const_pointer
                         , REF_CAT_IS_BVT
                         >::type            effective_const_pointer;
  . . .
```

41.2.9 Using the Traits

Using `adapted_iterator_traits`, iterator adaptor class template implementations can be relatively straightforward and follow the format shown in Listing 41.3.

Listing 41.3 Example Iterator Adaptor Using `adapted_iterator_traits`

```
template< typename I // The iterator to be adapted
        , typename T = adapted_iterator_traits<I>
        >
class some_kind_of_iterator_adaptor
{
public: // Member Types
  typedef I                                          base_iterator_type;
  typedef T                                          traits_type;
  typedef some_kind_of_iterator_adaptor<I>           class_type;
  typedef typename traits_type::iterator_category    iterator_category;
  // And for value_type, pointer, reference, difference_type,
const_pointer, const_reference, effective_reference,
effective_const_reference, effective_pointer and
effective_const_pointer.
  . . .
public: // Iteration Methods
  effective_reference       operator *();
  effective_const_reference operator *() const;
  effective_pointer         operator ->();
  effective_const_pointer   operator ->() const;
  . . .
public: // Element Access Methods
  effective_reference         operator [](difference_type index);
  effective_const_reference   operator [](difference_type index) const;
  . . .
```

If the user tries to use the adaptor in a way that the base iterator type does not support, the compiler will report that the function cannot return `void` since `adapted_iterator_traits` will have defined the given member type to `void` according to the base iterator type characteristics.

41.3 Summary

It's extraordinarily bad form to quote oneself, but in this case I feel it's fair to borrow from my *Philosophy of the Imperfect Practitioner* (defined in the preface to *Imperfect C++*), which has four tenets:

1. C++ is great but not perfect.
2. Wear a hairshirt.
3. Make the compiler your batman.
4. Never give up; there's always a solution.

That C++ is not perfect is obvious from the fact that we've needed to go to such mind-bending lengths to achieve our goal in crafting the traits class. That goal's worth reiterating, now, by the way: to give us iterator adaptors that are simple to code and that act as users expect, irrespective of their underlying type. The significance of this facility cannot be overstated!

Though perhaps not obvious to you right at this point, that C++ is great is evinced from the fact that such a class template *can* be created and used to the effects we'll see in the remaining chapters of Part III. Though all languages have good and bad aspects, it's not exaggerating to say that few offer the power of C++.

Wearing a hairshirt is not exemplified in the fact that this was an arduous journey. Rather, it's in the many places in which we've applied constraints and facilitated a reduced function set and, I suppose, in the fact that I've sought (and largely succeeded) to provide maximal power by discriminating between the capabilities of compilers and of the libraries with which they (usually) work.

We've certainly made the compiler our batman—a somewhat archaic English expression for an aide, helper, right-hand man (or woman!)—by having it perform these metaprogramming acrobatics at our behest. I can honestly say that my curious nature reaches its horizon at compiler technology, and from what I understand of that life from my friends who work on the other side, I am comfortable to "not know" and to merely continue to strongarm compilers to do my bidding (and yours).

In my opinion, the final tenet is the most apposite. Too often when you wish to use a library, you are informed that your compiler is "not supported." (Frankly, I find the alacrity with which that convenient but destructive phrase is wielded in some quarters bafflingly arrogant. I humbly suggest that you avoid such bleeding-edge-only libraries as you would a politician's handshake.) Given that C++ is both complex and continually evolving, it's simply unreasonable to expect the whole C++ programming world to march lockstep from one standard to the next. Further, since the number of C++ compiler vendors is gradually diminishing, we need to redouble our guard against proprietary control. Thus, techniques for maximizing backward compatibility will continue to be very important. As I hope I've shown you here, there is almost always a solution, so: Never give up!

41.4 On the CD

The CD contains an extra section, Old Libraries with New Compilers, that describes several further hurdles jumped in making the `adapted_iterator_traits` traits class work when using a new compiler (e.g., Intel 8) with an old standard library (such as that shipped with Visual C++ 6).

CHAPTER 42

Filtered Iteration

If you think you've arrived, you're ready to be shown the door.

—Steve Forbes

I can give you my word, but I know what it's worth and you don't.

—Nero Wolfe

42.1 Introduction

We saw in Chapter 36 that transforming an iterator is a matter of applying a unary function to the dereference. Can we do this with a predicate, to filter out items? We might imagine something like the following:

```
using recls::stl::search_sequence;
search_sequence  files(".", "*", recls::FILES | recls::RECURSIVE);

std::copy(filter(files.begin(), is_readonly())
    , filter(files.end(), is_readonly())
    , std::ostream_iterator<search_sequence::value_type>(std::cout
                                                        , "\n"));
```

42.2 An Invalid Version

How would this work? Naturally, `filter()` will be a creator function that returns an instance of a (suitably specialized) filtering iterator type. We might imagine an iterator class template such as that shown in Listing 42.1.

Listing 42.1 Invalid Version of `filter_iterator`

```
template< typename I // The adapted iterator
        , typename P // Unary predicate that will select items
        , typename T = adapted_iterator_traits<I>
        >
class filter_iterator
{
public: // Member Types
  typedef I                                             base_iterator_type;
  typedef P                                             filter_predicate_type;
  typedef T                                             traits_type;
  typedef filter_iterator<I, P, T>               class_type;
  typedef typename traits_type::iterator_category iterator_category;
  typedef typename traits_type::value_type       value_type;
  . . . // And so on, for usual members (from adapted_iterator_traits)
public: // Construction
  filter_iterator(I it, P pr)
    : m_it(it)
    , m_pr(pr)
  {
    for(; !m_pr(*m_it); ++m_it) // Get first "selected" position
    {}
  }
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    for(++m_it; !m_pr(*m_it); ++m_it) // Advance, then get next pos
    {}
    return *this;
  }
  class_type operator ++(int);         // Usual implementation
  reference operator *();              // Usual implementation
  const_reference operator *() const; // Usual implementation
private: // Member Variables
  I m_it;
  P m_pr;
};
```

Alas, the statement outputting read-only files shown in Section 42.1 will fail, probably in a crash. In fact, just about any use of this iterator will fail. There are two problems.

First, in the constructor for the first iterator, the active iterator, it uses the predicate and increment operator to ensure that the `filter_iterator` instance has the correct position before it is used. This correct position is the first one that matches the predicate, and that may be outside the given range [`files.begin()`, `files.end()`).

Second, the constructor for the second iterator, the one that adapts the endpoint iterator, dereferences its base iterator instance. It's a strict part of the STL *iterator* concept (Section 1.3) that we can "never [assume] that past-the-end values are dereferenceable" (C++-03: 24.1;5). (This also means that the implementation of `operator *()` is not well defined, but that's moot because we would have to go through an undefined constructor to get to a point where it could be invoked.)

42.3 Member Iterators Define the Range

It is clear that a filtering iterator instance must have access to a pair of iterators in order to avoid going outside the valid range. That being the case, our client code will be more verbose, for example:

```
search_sequence  files(".", "*", recls::FILES | recls::RECURSIVE);

std::copy(filter(files.begin(), files.end(), is_readonly())
    , filter(files.end(), files.end(), is_readonly())
    , std::ostream_iterator<search_sequence::value_type>(std::cout
                                                        , "\n"));
```

42.4 So . . . ?

One option might be to default the second endpoint iterator to `I()`, as follows:

```
template <. . .>
class filter_iterator
{
  . . .
public: // Construction
  filter_iterator(I it, P pr, I end = I())
    : m_it(it)
    , m_end(end)
    , m_pr(pr)
  {
    . . .
private: // Member Variables
  I m_it;
  I m_end;
  P m_pr;
};
```

However, this relies on the iterator type defining a default-constructed iterator as being equivalent to the endpoint iterator. Though this would work, on a case-by-case basis, for some iterators, including `readdir_sequence::const_iterator` (Section 19.3) and `findfile_sequence::const_iterator` (Section 20.5), it would not work for others, such as `glob_sequence::const_iterator` (Section 17.3). Or, if you prefer, it *might* work for `std::list`, `std::deque`, `std::map`, but it can't work for `std::vector` and, importantly, pointers.

Furthermore, providing this facility puts the onus on the user to know whether the assumption holds for a given iterator, which is both unreasonable and exceedingly likely to lead to failures. More leaking abstractions! Add the fact that such failures may never exhibit in testing, instead lurking until your product is out in the field, and this option is totally unacceptable.

"*Wait!*" you might say doggedly, "We can specialize the creator function to reject pointers." And so we can. However, there are plenty of iterators that are not pointers that do not satisfy the default-constructor/endpoint equivalence. For one, a random access iterator that is not a pointer will not do so.

Or you might wonder, "Can't we specialize to reject random access iterators?" Indeed, that would help, were it not for the fact that many of the iterators fulfilling other categories will also fail. In short, there's no getting away from the following rule and tip.

Rule: Never assume that a default-constructed instance of an iterator is equivalent to the endpoint iterator for the sequence or notional range for which the iterator acts.

Tip: Never use a filtering iterator adaptor that assumes, or allows the user to assume, that a default-constructed instance of the adapted iterator type is equivalent to the endpoint iterator.

With this in mind, let's see how to define a robust filtering iterator component.

42.5 `stlsoft::filter_iterator`

There's a fair bit to do in this class, so we'll tackle iterator refinements in a stepwise fashion. We'll start with input and forward iterators.

42.5.1 Forward Iterator Semantics

The handling of forward iterator semantics is shown in Listing 42.2.

Listing 42.2 Definition of `filter_iterator` Supporting Forward Iteration

```
template< typename I // The underlying iterator
        , typename P // The unary predicate that will select the items
        , typename T = adapted_iterator_traits<I>
        >
class filter_iterator
{
public: // Member Types
  . . . // All usual member types, most via T (adapted_iterator_traits)
public: // Construction
  filter_iterator(I begin, I end, P pr)
    : m_it(begin)
    , m_end(end)
    , m_pr(pr)
  {
    for(; m_it != m_end; ++m_it)
    {
      if(m_pr(*m_it))
      {
        break;
      }
    }
  }
public: // Forward Iterator Methods
  class_type& operator ++()
  {
    STLSOFT_MESSAGE_ASSERT( "Attempting to increment an endpoint
iterator", m_it != m_end);
```

```
    for(++m_it; m_it != m_end; ++m_it)
    {
      if(m_pr(*m_it))
      {
        break;
      }
    }
    return *this;
  }
  class_type& operator ++(int); // Canonical implementation
  effective_reference operator *()
  {
    return *m_it;
  }
  effective_const_reference operator *() const; // Same as operator *()
  effective_pointer operator ->()
  {
    enum { is_iterator_pointer_type
                    = is_pointer_type<base_iterator_type>::value };
    typedef typename
     value_to_yesno_type<is_iterator_pointer_type>::type  yesno_t;
    return invoke_member_selection_operator_(yesno_t());
  }
  effective_const_pointer operator ->() const; // Same as operator ->()
  . . .
private: // Member Variables
  I m_it;
  I m_end;
  P m_pr;
};
```

All member types are defined in terms of those provided by `adapted_iterator_traits` (just as is the case with `index_iterator`, described on the CD). The constructor takes the [`begin`, `end`) iterator pair defining the iterable range, followed by the predicate used for filtering. Note that the predicate comes last, as a reminder that defaulting the endpoint iterator is a crazy thing to attempt.

The constructor has to assume that the given base iterator instance specifying the start of the iterable range may not be one acceptable to the filter predicate and so tests it, possibly incrementing until finding one that is. Contrast this with the implementation of `operator ++()`, which knows that the current iteration point is acceptable to the filter and increments before it starts the loop. This is because the user *must* have previously tested it against the known endpoint filtered instance. This follows the basic idiom in STL that an iterator is determined to be viable by testing its equality against one that is known to not be.

The necessity to move to an acceptable point in the iteration implies the curious relationship whereby different start point iterators evaluate, in their filtered form, to be the same. Consider the sequence of integers 0, 2, 4, 5, 6, 7, 8, 9. Using a filter, `is_odd`, which selects odd numbers, there are several ways to specify equivalent iterators:

```
int ints[] = { 0, 2, 4, 5, 6, 7, 8, 9 };

stlsoft::filter(&ints[0], &ints[0] + 8, is_odd());  // Is equivalent to:
stlsoft::filter(&ints[1], &ints[0] + 8, is_odd());  //  this
stlsoft::filter(&ints[2], &ints[0] + 8, is_odd());  //  and this
stlsoft::filter(&ints[3], &ints[0] + 8, is_odd());  //  and this
```

Each of the iterators in that case actually refers to the element at index 3, whose value is 5, since that's the first one in the series that has an odd value.

The remainder of the implementation shown is entirely normal, given what we learned in Section 36.4.5 about handling the member selection operator. So far, so good.

42.5.2 Bidirectional Iterator Semantics

I expect you're ahead of me here. Given the current member variables, we cannot implement bidirectional iterator semantics because we stand the same risk of stepping outside the iterable range as discussed in Section 42.2, only this time we would step out of the beginning rather than the end. The remedy in this case is to remember the starting point. Hence, we add another member variable of the base iterator type, `m_begin`, and adjust the implementation of the constructor accordingly, as shown in Listing 42.3.

Listing 42.3 Member Variables Supporting Bidirectional Iteration

```
template <typename I, typename P, typename T>
class filter_iterator
{
  . . .
public: // Construction
  filter_iterator(I begin, I end, P pr)
    : m_it(begin)
    , m_begin(begin)
    , m_end(end)
    , m_pr(pr)
  {
    . . .
  }
  . . .
private: // Member Variables
  I m_it;
  I m_begin;
  I m_end;
  P m_pr;
};
```

Using this member, the implementation of the bidirectional iterator methods is surprisingly simple, as shown in Listing 42.4.

Listing 42.4 Predecrement Operators

```
  . . .
public: // Bidirectional Iterator Methods
  class_type& operator --()
  {
    STLSOFT_MESSAGE_ASSERT( "Attempting to increment an endpoint
iterator", m_it != m_begin);
    for(--m_it; m_it != m_begin; --m_it)
    {
      if(m_pr(*m_it))
      {
        break;
      }
    }
    return *this;
  }
  class_type& operator --(int); // Canonical implementation
  . . .
```

Now we can enumerate forwards as well as backwards:

```
template <typename I>
void fn(I from, I to)
{
  ++it;
  --it;
}

struct is_odd;

int ints[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

fn(stlsoft::filter(&ints[0], &ints[0] + 8, is_odd())); // Well-formed
```

42.5.3 Random Access Iterator Semantics

This one is very simple. There's no reasonable way to implement random access iterator semantics in a filtering iterator, so we don't do it. The only conceivable way would be extremely expensive since each apparent random access operation would involve an element-by-element iteration to identify which elements match the predicate. Even if this weren't the case, I simply can't conceive of a scenario in which random access filtering is meaningful. I may be wrong, of course, in which case please write to me and disabuse me of my erroneous assumptions.

Recommendation: Eschew support for random access (and higher) iterators in filtering iterators.

42.6 Constraining the Iterator Category

Given that we've chosen to eschew random access iterator semantics, we actually have a problem on our hands. If we adapt a random access iterator, the adapted form will think it's a random access iterator—the `iterator_category` member type of the adapted type will be `std::random_access_iterator_tag`—even though we've supplied it only with bidirectional iterator semantics. This is a problem. As soon as we try to pass this off to an algorithm that has a specialized form for handling random access iterators, things are going to get ugly. What you tend to see in this case is an enormous list of error messages, and ensconced within, if you're lucky enough to spot it, will be some mention of a missing `operator -()`, or `operator +()`, or some other operation specific to random access iterators.

You might wonder why we've not come across this with the other adaptors. Well, `transform_iterator` (Chapter 36), `member_selector_iterator` (Chapter 38), and `index_iterator` (extra chapter on the CD) are all able to exhibit the iterator category of their base type; `filter_iterator`, by its very nature, cannot.

Thus, the final act of cunning is to use the `min_iterator_category` template and its 16 full specializations, each of which corresponds to a permutation of two standard iterator categories. The primary template and several of the specializations are shown in Listing 42.5. In each permutation, the member type `iterator_category` is defined as the lesser refinement of the two specializing types.

Listing 42.5 Primary Template and Some Specializations of `min_iterator_category`

```
template< typename C1 // First category
        , typename C2 // Second category
        >
struct min_iterator_category;

template <>
struct min_iterator_category< std::input_iterator_tag
                            , std::input_iterator_tag
                            >
{
  typedef std::input_iterator_tag iterator_category;
};
template <>
struct min_iterator_category< std::forward_iterator_tag
                            , std::input_iterator_tag
                            >
{
  typedef std::input_iterator_tag iterator_category;
};
. . .
```

```
template <>
struct min_iterator_category< std::bidirectional_iterator_tag
                            , std::random_access_iterator_tag
                            >
{
  typedef std::bidirectional_iterator_tag iterator_category;
};
template <>
struct min_iterator_category< std::random_access_iterator_tag
                            , std::random_access_iterator_tag
                            >
{
  typedef std::random_access_iterator_tag iterator_category;
};
```

The traits class is used to limit the `iterator_category` member type to the maximum sensible refinement that is supportable (Listing 42.6).

Listing 42.6 Definition of `filter_iterator`

```
template< typename I // The underlying iterator
        , typename P // The unary predicate that will select the items
        , typename T = adapted_iterator_traits<I>
        >
class filter_iterator
{
public: // Member Types
  . . .
  typedef filter_iterator<I, P, T>             class_type;
  typedef typename min_iterator_category<
                        typename traits_type::iterator_category
                      , std::bidirectional_iterator_tag
                      >::iterator_category      iterator_category;
  . . .
```

42.7 Summary

We've seen that a filtering iterator adaptor *must* be instantiated from an iterator pair defining the viable range of the adapted range. It's slightly inconvenient to the user but is the only workable solution. We've also seen that by applying the `adapted_iterator_traits` traits class, we can achieve a simple definition for what is a sophisticated iterator adaptation.

42.8 On the CD

The CD contains a preview of how filtering can be more simply achieved using the *ranges* concept, which will be described in Volume 2.

CHAPTER 43

Composite Iterator Adaptations

Show me your flow charts and conceal your tables and I shall continue to be mystified. Show me your tables and I usually won't need your flow charts; they'll be obvious.

—Frederick P. Brooks

But if less is more, think how much more more would be!

—Dr. Frasier Crane, *Frasier*

43.1 Introduction

Naturally, you will now be pondering whether, and how, iterator adaptations may be used in concert. Well, the good news is that they can. The even better news is that you don't need to write a new iterator adaptor in order to combine them!

43.2 Transforming a Filtered Iterator

Let's imagine that we have a set of integers (`int`) and that we want to filter out the odd ones and then transform the remaining elements to `double` and take their square roots. Let's say we have the two function classes `is_even()` and `sqroot()`, as follows:

```
struct is_even
  : std::unary_function<int, bool>
{
  bool operator ()(int i) const
  {
    return 0 == (i % 2) /* && 0 != i */;
  }
};

struct sqroot
  : std::unary_function<int, double>
{
  double operator ()(int i) const
  {
    return ::sqrt(static_cast<double>(i));
  }
};
```

Using the `transformer()` (Section 36.3.1) and `filter()` (Chapter 42) creator functions, we can perform the filtration and transformation in a single statement:

```
std::list<int>  ints = . . .

std::copy(stlsoft::transformer(stlsoft::filter(ints.begin(), ints.end()
                                                , is_even())
                                    , sqroot())
        , stlsoft::transformer(stlsoft::filter(ints.end(), ints.end()
                                                , is_even())
                                    , sqroot())
        , std::ostream_iterator<double>(std::cout, "\n"));
```

This is effective, but it's quite a mouthful, and we've got the same kind of semibenign *DRY SPOT* (Chapter 5) violation seen in Section 36.3. It's not very nice.

43.2.1 Creator Function

We can simplify this, syntactically anyway, by defining a combining creator function, `transform_filter()`:

```
template< typename I  // The adapted iterator type
        , typename TF // Transformation function
        , typename FP // Filtering predicate
        >
transform_iterator<filter_iterator<I, FP>, TF>
 transform_filter(I from, I to, TF fn, FP pr)
{
  return transformer(filter(from, to, pr), fn);
}
```

The naming game (Chapter 37) becomes even more challenging when we get down to combining adaptors. (It's possible that this is the hardest part of effecting such combining functions!) Should it be `filter_transformer()`, `transforming_filter()`, `transformer_filter()` . . . ? I opted for the one that sounded best (to this English speaker), `transform_filter()`. Plugging this into our example relieves the matter somewhat, as shown in the following code. (Naturally, there's a predictable longhand equivalent: `make_transform_filter_iterator()`.)

```
std::copy(stlsoft::transform_filter(ints.begin(), ints.end(), sqroot()
                                    , is_even())
        , stlsoft::transform_filter(ints.end(), ints.end(), sqroot()
                                    , is_even())
        , std::ostream_iterator<double>(std::cout, "\n"));
```

Note that the transforming function comes before the filtering predicate, in the same order as the component names in the creator function name. This is by no means a matter of one convention being supremely superior to another. You might equally take the view that the function objects should appear in the order in which they are applied in the adaptations.

I've added a little compile-time constraint and a suitable comment in the implementation of `transform_filter()` to catch mistakes in the order of the function object arguments:

```
template <typename I, typename TF, typename FP>
transform_iterator<filter_iterator<I, FP>, TF>
    transform_filter(I from, I to, TF fn, FP pr)
{
  typedef typename FP::result_type    pred_res_t;
  // If this fires, either you've specified the transforming function
  // and the filtering predicate in the wrong order, or your predicate
  // has a nonintegral return type (which would be decidedly odd).
  STLSOFT_STATIC_ASSERT(0 != is_integral_type<pred_res_t>::value);
  return transformer(filter(from, to, pr), fn);
}
```

43.3 Filtering a Transformed Iterator

The obvious question now is whether we can filter a transformed iterator, and the answer, naturally enough, is that we can. It's as simple as reversing the order of application of the adaptors, using a creator function, as follows:

```
template< typename I  // The adapted iterator type
        , typename TF // Transformation function
        , typename FP // Filtering predicate
        >
filter_iterator<transform_iterator<I, TF>, FP>
 filter_transformer(I from, I to, FP pr, TF fn)
{
  typedef typename FP::result_type     pred_res_t;
  // If this fires, either you've specified the transforming function
  // and the filtering predicate in the wrong order, or your predicate
  // has a nonintegral return type (which would be decidedly odd).
  STLSOFT_STATIC_ASSERT(0 != is_integral_type<pred_res_t>::value);
  filter(transformer(from, fn), transformer(to, fn), pr);
}
```

The only significant difference, besides the definition of the return type, is that the `transformer()` creator function is invoked twice, once each on `from` and `to`, in order to give `filter_iterator` (Chapter 42) the iterator pair it requires.

A less important issue is the inconsistency between the order of template parameters. When I was implementing this function, I naturally did a copy/paste from `transform_filter()`. Because of this, the template parameters for both functions are in the order `I`, `TF`, `FP`, rather than `I`, `FP`, `TF` for the second case, which would be the more consistent way. Although to a borderline perfectionist (like me) this niggles, it really is entirely irrelevant. The compiler doesn't care one whit about the order of the template parameters in cases like this since all template parameters are unambiguously inferable from the function's arguments. Which is nice.

43.4 Hedging Our Bets

These functions are located within the *<stlsoft/iterators/transform_filter_iterator.hpp>* and *<stlsoft/iterators/filter_transform_iterator.hpp>* files, respectively. The names of these files imply to the user that bona fide template classes `transform_filter_iterator` and `filter_transform_iterator` are defined, instances of which are returned by the respective creator functions. The reason for doing this is straightforward and deliberate. Compilers exhibit different aptitudes when it comes to the optimization of templates, and adapted templates in particular. Following the **STLSoft** naming conventions for the files, and defining creator functions whose names imply (or at least allow for) the existence of a single composite iterator adaptor, affords us the ability to change the implementation in the future without disrupting extant client code.

43.5 Summary

This chapter illustrated the relative simplicity of combining nontrivial iterator adaptation functionality with the simple technique of defining creator functions. This is one of the real powerhouses of STL extension and also one of the most cost effective.

Epilogue

Times are bad. Children no longer obey their parents, and everyone is writing a book.

—Cicero

I have no idea what I'm going to write until the moment I sit down at the typewriter.

—Larry McMurtry

I hope you've enjoyed the first part of our journey into STL extension. I know I have. Well, sort of; as with my previous book-writing activities, it has taken about four times as long as I expected. Notwithstanding, I'm silly enough to be trying again, and not once, but twice! So, as long as my lovely wife doesn't (deservedly) throttle me for again risking penury, one of the following portents is true.

Matthew Wilson will return in *Breaking Up the Monolith*

wherein the *shim* concept, the ***Type Tunneling*** and ***Handle::Ref+Wrapper*** patterns, and the principles of *Intersecting Conformance* and *Irrecoverability* will join forces to address the balance between cohesion, expressiveness, flexibility, performance, and coupling.

Or:

Matthew Wilson will return in *Extended STL, Volume 2*

wherein we'll continue our journey into the world of STL extension by visiting the lands of *function objects*, *algorithms*, *adaptors*, *allocators*, and much more besides.

TTFN.

Bibliography

Self-education is, I firmly believe, the only kind of education there is.
—Isaac Asimov

Beware of people who only have one book.
—Billy Connolly

All of the following were helpful in the preparation of *Extended STL, Volume 1*.

Publications about STL:

- *Effective STL*, Scott Meyers (Boston, MA: Addison-Wesley, 2001)
- *Generic Programming and the STL*, Matthew Austern (Reading, MA: Addison-Wesley, 1999)
- *STL Tutorial and Reference Guide*, Second Edition, David R. Musser, Gillmer J. Derge, and Atul Saini (Boston, MA: Addison-Wesley, 2001)
- *The C++ Standard*, Second Edition, ISO/EIC 14882 (New York: American National Standards Institute, 2003)

Books about other things:

- *Advanced Programming in the UNIX Environment*, W. Richard Stevens (Reading, MA: Addison-Wesley, 1993)
- *Advanced Windows Programming*, Third Edition, Jeffrey Richter (Redmond, WA: Microsoft Press, 1997)
- *The Art of UNIX Programming*, Eric Raymond (Boston, MA: Addison-Wesley, 2004)
- *Beyond the C++ Standard Library: An Introduction to Boost*, Björn Karlsson (Boston, MA: Addison-Wesley, 2005)
- *The Cathedral and the Bazaar*, Eric Raymond (Sebastopol, CA: O'Reilly, 2001)
- *C++ Common Knowledge*, Stephen C. Dewhurst (Boston, MA: Addison-Wesley, 2005)
- *The C++ Programming Language*, Third Edition, Bjarne Stroustrup (Reading, MA: Addison-Wesley, 1997)
- *The Design and Evolution of C++*, Bjarne Stroustrup (Reading, MA: Addison-Wesley, 1994)

- *Don't Make Me Think*, Steve Krug (Indianapolis, IN: New Riders, 2000)
- *Effective C++*, Third Edition, Scott Meyers (Boston, MA: Addison-Wesley, 2005)
- *Facts and Fallacies of Software Engineering*, Robert L. Glass (Boston, MA: Addison-Wesley, 2003)
- *Imperfect C++*, Matthew Wilson (Boston, MA: Addison-Wesley, 2004) (Cracking read, this one!)
- *Joel on Software*, Joel Spolsky (Berkeley, CA: Apress, 2004)
- *More Effective C++*, Scott Meyers (Reading, MA: Addison-Wesley, 1996)
- *Object-Oriented Software Construction*, Bertrand Meyer (Upper Saddle River, NJ: Prentice Hall, 1997)
- *The Pragmatic Programmer*, Andrew Hunt and David Thomas (Boston, MA: Addison-Wesley, 2000)
- *Small Things Considered*, Henry Petroski (New York: Knopf, 2003)
- *UNIX Network Programming, Volume 1*, W. Richard Stevens (Reading, MA: Addison-Wesley, 1998)

Index