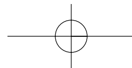


Prologue

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

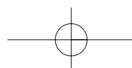
Late in the preparation of my first book, *Imperfect C++*, I proposed this book to my editor and confidently pronounced that it would be easy, would consist entirely of readily digestible content, would take me less than six months, and would be so thin it would slide easily between two layers of abstraction. 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.

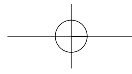
1. STL is not intuitive and requires a significant mental investment in order to acquire a position of comfort in using it.
2. For all its technical sophistication and brilliant cohesiveness, STL is limited in outlook and does not adequately address abstractions outside its sometimes narrow conceptual 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



fm01.qxd 5/24/07 11:16 AM Page xxv



some degree, are leaky." Simply, this means that successful use of nontrivial abstractions requires some knowledge of what's *under* the abstraction.

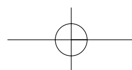
This is but one of the reasons why many C++ developers roll their own libraries. It's not just due to *Not Invented Here* syndrome; it's because you can, and often do, find yourself using a component written by a third party and find that, while you know and can use and understand, say, 80% of its functionality, 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.

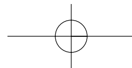
Principles of UNIX Programming

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.



fm01.qxd 5/24/07 11:16 AM Page xxvi



- *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.

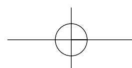
These days, I am a little wiser, and I recognize that the software engineering landscape is a broad one with many differing requirements. Execution time is not always an important factor, let alone the overriding one (*Principle of Economy*). When writing a system script, using Python or (perplexingly) 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.

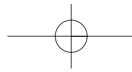
(You should still buy it, though!)

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



fm01.qxd 5/24/07 11:16 AM Page xxvii



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

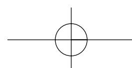
While efficiency is a strong motivating factor for the use of C++, it is tempered by deficiencies in two necessary attributes of any libraries you may consider using: discoverability and transparency. Extensive definitions of these two aspects of software in general are given in Raymond's *Art of UNIX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

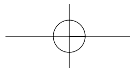
Definition: Discoverability is how easy it is to understand a component in order to be able to use it.

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.

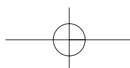
Expressiveness

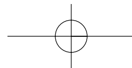
Another reason people use C++ is that it is extremely expressive (powerful).

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.

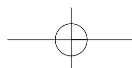
Now let's look at a C++/STL form, using the class `unixstl::readdir_sequence` (Chapter 19).

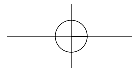
```
readdir_sequence entries(".", readdir_sequence::files);

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

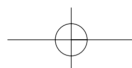
If something doesn't work, it's not likely to be successful. C++ is sometimes accused of being a language that too easily facilitates bad practice. Naturally, I don't agree, and I contrarily assert that, with appropriate discipline, C++ programs can be extremely robust. (My favorite real-world paying-job 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.

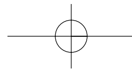
The principles of *Clarity*, *Composition*, *Modularity*, *Separation*, and *Simplicity* all apply to robustness; it is featured strongly in this book.

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.



fm01.qxd 5/24/07 11:16 AM Page xxxi




```
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.

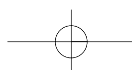
```
std::string const&  pathName = . . .
dynamic_library     dl(pathName.c_str());
```

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.

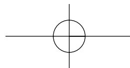
Modularity

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.



fm01.qxd 5/24/07 11:16 AM Page xxxii



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.

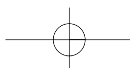
STL extensions by nature need a high degree of portability, from handling different operating systems right through to handling compiler bugs and language dialects, so this characteristic also receives a strong focus throughout the book.

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

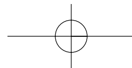
It should be no great surprise to learn that very few libraries can be said to score highly on all seven signs of success. Only very small libraries with very tightly defined functionality are likely to do so. For all the rest, the challenge in writing such libraries lies in finding the appropriate balance. 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 soft-



fm01.qxd 5/24/07 11:16 AM Page xxxiii



ware 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&);
};
```

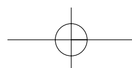
This is a widely recognized practice, so much so that it is part of the common lore of C++. It is an idiom. Similarly, STL is one gigantic idiom. Once you are experienced in STL, using the basic STL provided by the standard library becomes idiomatic. Where STL extension introduces new concepts and practices,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

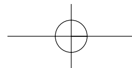
Identifying and reinforcing established idioms, describing useful new idioms, warning against faux/anti-idioms, and looking inside the black box are the tactical measures of this book (and Volume 2), with which we navigate the balance of the seven signs of successful C++ software libraries.

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



fm01.qxd 5/24/07 11:16 AM Page xxxiv



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.

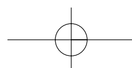
Although it offers many useful features for STL and non-STL programming alike, the main purpose of **STLSoft** is to provide general-purpose components and facilities, at a moderately low level of abstraction, to support commercial projects and other open-source libraries. It is high on efficiency, flexibility, 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`,



fm01.qxd 5/24/07 11:16 AM Page xxxv

`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.

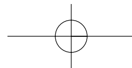
Although each **STLSoft** subproject (apart from the **STLSoft** main project, which resides in `stlsoft`) has its own apparently distinct top-level namespace, these namespaces are actually aliases for nested namespaces within the `stlsoft` namespace. For example, the `comstl` namespace is actually defined 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

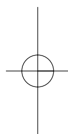
Boost is an open-source organization whose focus is the development of libraries that integrate with the standard library and may be proposed as future contributions to the standard. It has a large contributor base, including several members of the C++ standards committee.

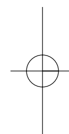
I'm not a **Boost** user or contributor, so **Boost** components aren't covered in detail in Volume 1; only the `boost::tokenizer` component is discussed (Section 27.5.4). If you want to learn how to *use* **Boost**, you should check out the book *Beyond the C++ Standard Library: An Introduction to Boost*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