Tools.h++ User's Guide

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-7666-10
Revision A, December 1996

Contents

About Tools.h++ 1

Overview and Features of Tools.h++

Tools.h++ is a rich, robust, and versatile C++ foundation class library: a set of software parts you can use to build virtually any application.

Tools.h++ is an industry standard. It is shipped by a wide variety of compiler vendors with every copy of their compilers. Preferred by thousands of users world wide, it is ported to numerous compilers and operating systems. *Tools.h++* is available on almost any development platform you choose.

This new version of *Tools.h++* is built on the Standard C++ Library. To aid your transition into this technology, *Tools.h++* provides a familiar object-oriented interface, and a reliable upward migration path. You can count on *Tools.h++* to track and incorporate revisions of the Standard C++ Library as they are approved.

Your new *Tools.h++* package includes:

- *Powerful single, multibyte, and wide character support*
  You can manipulate single and multibyte strings with class *RWCString*'s full suite of operators and functions, or choose class *RWWString* for wide character strings. Both classes make it easy to do concatenation, comparison, indexing (with optional bounds checking), I/O, case changes, stripping, and many other functions. In addition, classes *RWCSubString* and *RWWSubString* allow extraction and assignment to substrings; classes

RWCRegexp and *RWCRExpr* support regular expression pattern searches; and classes *RWCTokenizer* and *RWWTokenizer* break single and wide character strings, respectively, into separate tokens.

- *Extended regular expressions*
  Here's a richer set of pattern matching tools you can use to search for information in strings. The new *Tools.h++* extended regular expression features are a subset of those found in the ANSI/ISO standard POSIX.2 (Portable Operating System Interface), and require the presence of the Standard C++ Library.

- *Time and date handling classes*
  You can calculate the number of days between two dates, or the day of the week a date represents. Read and write days or times in arbitrary formats, or whatever you need to do. *Tools.h++* helps you master time.
- *Internationalization support*
  You can internationalize your software with the convenient and easy-to-use framework of class *RWLocale*, and use class *RWTimeZone* to manipulate time zones and daylight-saving time. The entire library is eight-bit clean, so you can use it with any eight-bit character set. Embedded nulls are fully supported.
  - *Endian streams*
    You can transfer information between operating systems with the efficiency of a binary stream. The endian streams mechanism, which keeps a record of the operating environment where information originates, allows the stream to be read on any system regardless of its native size or byte order.
- *Multithread safe*
  You can count on multithread safety. When compiled with a multithread option, the library uses multithread safe system facilities, with enough internal locking to maintain its internal integrity. See the release notes for your compiler.
  - *Persistent store*
    This new version of *Tools.h++* enhances an already powerful and sophisticated store facility. Isomorphic persistence, which maintains an object's pointer relationships, is now supported for most *Tools.h++* collections, including the template-based collections. You can also implement isomorphic persistence on your own classes. Objects that inherit from *RWCollectable* have polymorphic persistence, which not only maintains pointer-relationships, but also allows processes to restore objects without knowing their types.

- *Template based classes*
  Twenty-eight new or re-engineered class templates based on the Standard C++ Library container classes. You can use the full interface to these classes if your development environment supports the Standard C++ Library. If you don't have the Standard C++ Library, *Tools.h++* supplies template-based classes with a subset of the same interfaces.
- *Generic collection classes*
  If your compiler does not yet support templates, *Tools.h++* includes a set of template-like classes that use the C++ preprocessor and `<generic.h>`, a header file included with most compilers. The interface to these generic classes is similar to the template-based classes, so you can make an easy transition.
- *Smalltalk-like collection classes*
  You get a complete library of collection classes, modeled after the Smalltalk-80 programming environment, including *Set*, *Bag*, *Queue*, *Stack*, *OrderedCollection*, *SortedCollection*, *Dictionary*, and more.
- *Many other features:* *RWFile Class* encapsulates standard file operations. *B-tree disk retrieval* uses B-trees for efficient keyed access to disk records. *File Space Manager* allocates, deallocates and coalesces free space within a file. *A complete error handling facility*, which takes advantage of C++ exceptions if they are available. *Still more classes, including:* bit vectors, virtual I/O streams, cache managers, and virtual arrays.

Tools.h++ and the C++ Philosophy

If you're familiar with C++, you'll feel comfortable with *Tools.h++*. As a C++ class library, *Tools.h++* shares many design goals with the C++ language itself. These mutual goals include:

- *Efficiency.* In general, you will find no feature in *Tools.h++* that impairs non-users of the feature. As many decisions as possible are made at compile time, consistent with the C++ philosophy of static type checking. In most cases, *Tools.h++* offers you a choice between classes with extreme simplicity, but little generality, and complex classes with more generality.
- *Simplicity.* To maintain simplicity, *Tools.h++* uses few subclasses. Although the overall architecture is sophisticated and integrated, each class usually plays just one well-defined role. Many functions are also simple, consisting of a few lines of code. New features are added sparingly: in general, if there is already a way to do it, we leave it out!

- *Compactness.* Like C++, *Tools.h++* aims to make programs compile small. Always a desirable design goal, it also facilitates using programs in embedded systems. Templates have a mixed effect in this regard: encouraging compact source code, but in many cases compromising compactness when compiling.
- *Predictability.* All of the familiar operators work just as you might expect—there are no surprises, no esoteric overloaded operators. And *Tools.h++* offers you great symmetry, making it possible to do things like change the implementation of a dictionary from a hash table to a B-tree with impunity.

Tools.h++ and the Standardization of C++

Almost everybody sees the benefits of standardizing the C++ language and the Standard C++ Library. The trick is to keep working during the process. We call this a period of transition, and the C++ community is engaged in it now.

Here is what the transition looks like: a standard nearing completion, but not yet fully stable. Although the standard itself is unlikely to be substantially revised, the fine tuning and ratification will continue into 1997.

And here is what the transition looks like: compilers evolving toward the standard at various rates. For a time, you will find new language features—such as namespaces, default template arguments, member function templates, nested class templates—supported on some compilers and not others. Some compilers may not even include a version of the Standard C++ Library; many will offer versions which conform to the standard only so far as they support the necessary language features. It will be some time before commercial compilers actually implement the exact C++ language, or include the Standard C++ Library as described in the standard.

Finally, here is what the transition looks like: you, the developer, and what you're going through now. You are the one evolving designs and implementations toward the emerging standard. Change will come at rates determined by your development environment, application domain, and corporate culture.

Our goal for *Tools.h++* is to help you to maintain consistency in your development while moving, at your own pace, along the path of the latest C++ technology.

Harnessing the Standard

The primary challenge of this new version of *Tools.h++* was to establish our relationship with the ANSI/ISO Standard C++ Library. Rogue Wave is committed not only to bringing our products into compliance with the standard, but to harnessing its full power. The object is to provide you with even more useful and efficient class libraries. The process of integrating our libraries with the C++ standard begins here with *Tools.h++ Version 7*.

For this version of *Tools.h++*, we have concentrated our integration efforts on the Standard C++ Library containers, often referred to as the STL or Standard Template Library. Each of the standard containers has been wrapped with a new or re-engineered *Tools.h++* collection class template. You'll find a full explanation of templates in Chapter 11. Following are the major design goals for our integration of *Tools.h++* and the Standard C++ Library, along with examples of how they are reflected in this version:

- Design Goal: Leverage
  To offer greater value by taking advantage of the Standard C++ Library to build upon a higher foundation than the base C++ language.

  For example, *Tools.h++* offers collections that use Standard C++ Library containers for their implementations. Building *Tools.h++* upon the standard enables these collections to easily supply standard iterators, which in turn allows them to be used with the rich set of Standard C++ Library algorithms. At the same time, you retain the safe, easy-to-use, object-oriented interface that *Tools.h++* collections have always provided.

- Design Goals: Interoperability
  To support one of the primary benefits of the C++ standard, which is to allow libraries, modules, classes, and algorithms from diverse providers to easily work together at a high level.

  For example, we made sure you can safely and efficiently pass a *Tools.h++* doubly-linked list where a Standard C++ Library *list* is expected.

- Design Goal: Freedom
  To maintain access to the Standard C++ Library.

  For example, when using a *Tools.h++* collection implemented with a Standard C++ Library container, you are always free to drop down to the level of the implementation that takes advantage of the non-object-oriented features of the Standard C++ Library.

- Design Goal: Object-orientation
  To enhance the Standard C++ Library with efficient, object-oriented interfaces.

  All our new and re-engineered collection class templates exemplify this goal. In each case, we have put an efficient wrapper around a corresponding Standard C++ Library container to provide a familiar, though expanded, *Tools.h++* collection interface.

- Design Goals: Simplicity and Safety
  To enhance the Standard C++ Library with a simpler interface, which reduces risk and makes client code easily maintainable.

  The object-oriented interface helps achieve this goal. Unlike the Standard C++ Library, the *Tools.h++* container methods know what data they control, freeing the user from the need to specify iterators and algorithms.

- Design Goal: Compatibility
  To protect our customers' investment in code written with previous versions of Tools.h++.

  For example, we have re-engineered the *Tools.h++ Version 6.1* collection class templates to base them on Standard C++ Library containers. In almost all cases, your existing source code that used classes in the previous version of the library will compile with the new library without modification.

- Design Goal: Smooth Transition
  To provide the means for developers to begin moving along the path toward standard C++ at their own pace and with minimal hassle.

  For example, you can use *Tools.h++* with or without the Standard C++ Library. If your development environment supports a version of the Standard C++ Library certified for use with *Tools.h++*, we offer 28 new or re-engineered class templates implemented using the Standard C++ Library container classes. If you don't have the Standard C++ Library, we offer you a subset interface to many of the same class templates, implemented using the technology of previous versions of *Tools.h++*. The appropriate implementation is selected automatically and transparently at compile time. By coding to the more restricted interface, you will be able to take full advantage of the Standard C++ Library as soon as it becomes available to you.

What We Didn't Do

Future versions of *Tools.h++* will make full use of the Standard C++ Library and other newly added features of the C++ language. This version includes several areas where we have elected to wait before incorporating the latest available technology. In some cases, we're waiting until the standard library or language feature is more widely available. In other cases, frankly, we're waiting until we gain more experience with the new features to see how we can best mold them into a unified and effective whole. We want to be careful not to commit ourselves and our customers to less than optimal patterns of usage. In the meantime, we'd like to draw your attention to the following areas:

- *RWCString* and *RWWString*
  Tools.h++ continues to use classes *RWCString* and *RWWString*. These classes, along with their substring classes and collaborating regular expression and tokenizer classes, have long been considered among the most useful and powerful classes in the library. This suite of functionality is not offered by the Standard C++ Library. You may use the C++ standard *string* and *wstring* in your applications, but you may occasionally incur the overhead of copying if you must convert between *Tools.h++* and standard strings.
- *RWLocale*
  Tools.h++ continues to use class *RWLocale*. At the time of this release, the C++ standard locale class specification is still undergoing review by the ANSI/ISO standards committees.
- *Exception Hierarchy*
  Tools.h++ continues to use its own exception hierarchy, which is similar to the exception hierarchy in the draft C++ Standard. We don't expect to change over until the standard exception hierarchy is more widely available. You are free to use standard exceptions in your application, but you must be prepared also to catch *Tools.h++* exceptions when making calls into the *Tools.h++* library from within your try blocks.
- *Namespaces*
  Tools.h++ is not yet using or attempting to use namespaces specified by the current draft. For now, we continue to use the *RW* prefix to distinguish our classes within the global namespace. Of course, this does not preclude you from using namespaces in your own application, if your compiler allows it.

1

Reading This Manual

This manual is an introduction to using *Tools.h++*, Rogue Wave's foundation class library. It assumes that you are familiar with C++. If you are not, you will find several books of interest in the Bibliography.

If you're an advanced C++ user, you may want to accompany this manual with Stroustrup [1991], Lippman [1991], or Ellis and Stroustrup [1990]. The latter is sometimes referred to as "The ARM," the Annotated Reference Manual. The terse but precise style of these works makes them excellent references to the language.

Special Conventions

When reading this manual, you'll notice the following special conventions:

- `courier font`—Used for disk directories, file names, examples, operating system commands, function names, code and code fragments (for example, `RWTPtrHashSet<int RWDefHArgs(int)> hset;`, `deque<T>`, `isEqual`). Most function names start with a lower case letter, but subsequent words are capitalized (for example, `compareTo()`).
- *Bold Sans Serif Italic*–Used for classes (for example, *RWCollectable or RWCString*). Most Rogue Wave classes include a prefix *RW* which is de-emphasized. The class name conveys what the class does, and distinguishes a Rogue Wave class from the generically named class of another vendor (for example, *RWIterator*, not just *Iterator*.) Class names begin with capital letters.
- *Sans Serif Italic*—Used for Rogue Wave product names (for example, *Tools.h++).*
- *Italic or* bold *only*—Conventional uses, such as emphasis or special terminology.
- Vertical ellipses—In code examples, they indicate that some part of the code is missing:

```
main()
{
.
.   //Something happens
.
}
```

Rogue Wave Professional Training

To help you get a head start on your project, Rogue Wave Professional Services provides training that can put the power of *Tools.h++*, or any other robust Rogue Wave library, into your hands in less than a week.

Rogue Wave training and mentoring is available for all levels of project development, from analysis and design to implementation. We also offer world-class courses in C++ and object-oriented programming.

For information on Rogue Wave's products and professional services, call us by phone, contact us by e-mail, or review the information on our World Wide Web site:

Telephone:	(541) 754-5010 (800) 487-3217
e-mail:	training@roguewave.com
WWW:	http://www.roguewave.com

On-line Documentation

Rogue Wave provides on-line documentation that supplements this manual. On-line documentation is in the `rogue\docs` directory. The docs directory contains important information regarding specific compilers and operating systems, how to use shared libraries and DLLs, and information that became available after the manual was published. We urge you to read all the files of the docs directory, but especially `toolread.doc`, the *Tools.h++* readme file.

The Frequently Asked Questions (FAQ) document is a new feature of the Rogue Wave home page (`http://www.roguewave.com`). The home page also contains late-breaking information about Rogue Wave products.

Technical Support

Rogue Wave is proud of its reputation for superior technical support. Our support policies are described in the technical support brochure that accompanies this product. Extended technical support contracts can be purchased from Rogue Wave or authorized partners.

1

Your first line of technical support is the documentation provided with this product, both on-line and in the manual. Many times, you can find what you need there, and save us both a call.

If you do need to call technical support, the first thing we ask is your *name*, your *company*, and the *serial number* of your product. Look for this number on your disk, or contact your system administrator.

It would also save both your time and ours if you do the following before you call:

- *Review your information.* Read relevant portions of the manual, the `rogue\docs` files, especially `toolread.doc`, and the FAQ (Frequently Asked Questions) file in the directory or at our web site.

Collect your numbers. In addition to your product serial number, we need to know what version of *Tools.h++* you're using. You'll find this number at the top of `toolread.doc`. We'll also ask for your compiler and operating system, and their respective version numbers.

Isolate your problem to a small test case, if applicable. Short code is easier to understand, transmit, and verify on our systems.

How to Contact Technical Support

You can contact technical support via any of the following paths:

FAX:	(541) 758-4761
Telephone:	(541) 754-2311
BBS:	(541) 754-5011
Mail:	850 SW 35th Street Corvallis, OR 97333 USA
e-mail:	support@roguewave.com
World Wide Web:	http://www.roguewave.com

Class Overview 2

This section gives an overview of *Tools.h++*, and highlights some common points among the classes.

Tools.h++ provides *implementation*, not policy. Hence, it consists mostly of a large and rich set of *concrete classes* that are usable in isolation and independent of other classes for their implementation or semantics. They can be pulled out and used just one or two at a time. Concrete classes are the heart of *Tools.h++*.

Tools.h++ also includes a rich set of *abstract base classes*, which define an interface for persistence, internationalization, and other issues, and a number of *implementation classes* that implement these interfaces. Although public, the implementation classes act like private classes. They are not designed to be used, and are therefore not documented.

Some *Tools.h++* classes are further categorized as collection classes, or collections. A central feature of *Tools.h++*, the collection classes fall into three groups:

- Template-based collection classes, called collection class templates, or just templates, for short;
- Generic collection classes;
- Smalltalk-like collection classes.

Regardless of their implementation, collection classes generally follow the Smalltalk naming conventions and semantical model: *SortedCollection*, *Dictionaries*, *Bags*, *Sets*, and so on. They use similar interfaces, allowing them to

be interchanged easily. The template-based and generic collections will hold any kind of object; the Smalltalk-like collections require that all collected items inherit from *RWCollectable*.

Choosing which collection classes to use in your programs is not a trivial task. We have added an appendix called *Choosing a Collection* to help you decide which class is the best for your purposes.

Table 2-2 at the end of this chapter gives the class hierarchy of all the public *Tools.h++* classes. In addition to these public classes, *Tools.h++* contains other classes for its own internal use.

Concrete Classes

The concrete classes consist of:

- The *simple classes* representing dates, times, strings, and so on, discussed in Chapter 3 through Chapter 5;
- The *template-based collection classes*, discussed in Chapter 11;
- The *generic collection classes* using the preprocessor `<generic.h>` facilities, discussed in Chapter 12.

Simple Classes

Tools.h++ provides a rich set of lightweight *simple classes*. By lightweight, we mean classes with low-cost initializers and copy constructors. These classes include: *RWDate* (for dates); *RWTime* (for times, with support for various time zones and locales); *RWCString* (for single and multibyte strings); *RWWString* (for wide character strings); and *RWCRegexp* or *RWCRexpr* (for regular expressions). Most of these classes can be held in four bytes or less, and have very simple copy constructors (usually just a bit copy) and no virtual functions. See the *Class Reference*.

Template-based Collection Classes

Template-based collection classes, or templates for short, give you the advantages of speed and type-safe usage. When templates are used sparingly, their code size can be quite small. When templates are used with many different types, however, their code size can become large because each type effectively generates a whole new class. If your compiler is capable of using the Standard C++ Library, you can use the *Tools.h++* template-based collections

that are based on the Standard C++ Library. If your compiler does not have access to the Standard C++ Library, you can still use a subset of the templates, as described in “Standard C++ Library Not Required” on page 92 and “Using Templates Without the Standard Library” on page 106.

Generic Collection Classes

Generic collection classes are those which use the `<generic.h>` preprocessor macros supplied with your C++ compiler. They can approximate templates, in the sense that they are typesafe, for compilers that do not support templates, and so are highly portable. However, because they depend heavily on the preprocessor, it can be difficult to use a debugger on code that contains them. See Chapter 12 for more information.

Abstract Base Classes

Tools.h++ includes a set of abstract base classes and corresponding specializing classes that provides a framework for many issues. The list below identifies some of these issues and associates them with their respective abstract base classes. The description of each class in the *Class Reference* indicates if it is an abstract base class.

Table 2-1 Abstract Base Classes and Issues

Issue	Class	Section Where Discussed
Locale	*RWLocale*	“RWLocale and RWZone” on page 220
Time zones	*RWZone*	“RWLocale and RWZone” on page 220
Virtual streams	*RWvistream* *RWvostream*	Chapter 6
Polymorphic persistence	*RWCollectable*	Chapter 14
Virtual page heaps	*RWVirtualPageHeap*	Class Reference
Model-View-Controller abstraction	*RWModel* *RWModelClient*	Class Reference

2

Smalltalk-like Collection Classes

The Smalltalk-like collection classes of *Tools.h++* give you much of the functionality of such Smalltalk namesakes as *Bag* and *SortedCollection*, along with some of the strengths and weaknesses of C++. The greatest advantages of the Smalltalk-like collections are their simple programming interface, powerful I/O abilities, and high code reuse. Their biggest disadvantages are their relative lack of type-safety, and their relatively large object code size. Large code is typical even when these classes are used in only small doses because of their initially high overhead in code machinery. All objects to be used by the Smalltalk-like collection classes must inherit from the abstract base class *RWCollectable*.

Common Member Functions

Whatever their category, all classes have similar programming interfaces. This section highlights their common functionality.

Persistence

Tools.h++ uses the following member functions to store an object of type `ClassName` to and from an *RWFile*, and to and from the Rogue Wave virtual streams facility, and to restore it later:

```
RWFile&      operator<<(RWFile& file, const ClassName&);
RWFile&      operator>>(RWFile& file,       ClassName&);
Rwvostream&  operator<<(RWvostream& vstream,const ClassName&);
Rwvistream&  operator>>(RWvistream& vstream,  ClassName&);
```

Class *RWFile*, which encapsulates ANSI-C file I/O, saves objects in binary format. The result is efficient storage and retrieval to files. For more information on *RWFile*, see Chapter 7 and the *Class Reference*.

Classes *RWvistream* and *RWvostream* are abstract base classes used by the Rogue Wave virtual streams facility. The final output format is determined by the specializing class. For example, *RWpistream* and *RWpostream* are two classes that derive from *RWvistream* and *RWvostream*, respectively. They store and retrieve objects using a portable ASCII format. The results can be transferred between different operating systems. These classes are discussed in more detail in Chapter 6 and the *Class Reference*.

It's up to you to decide whether to store to *RWFile*s, or to Rogue Wave streams. Storing to *RWFile*s gives you speed, but limits portability of results to the host machine and operating system. Storing to Rogue Wave streams is not as fast, but you get several specializing classes that provide other useful features, including highly portable format between different machines, and XDR stream encapsulation for distributed computations.

Store Size

The following common member functions return the number of bytes of secondary storage necessary to store an object of type `ClassName` to an *RWFile*:

```
Rwspace  ClassName::binaryStoreSize() const;
Rwspace  ClassName::recursiveStoreSize() const;
```

The member functions use the function:

```
RWFile& operator<<(RWFile& file, const ClassName&);
```

The above member functions are good for storing objects using classes *RWFileManager* and *RWBTreeOnDisk*. For objects that inherit from *RWCollectable*, the second variant `recursiveStoreSize()`can calculate the number of bytes used in a recursive store. The variant uses the function:

```
RWFile& operator<<(RWFile& file, const RWCollectable&)
```

You can use class *RWAuditStreamBuffer* in conjuction with any stream to count the number of bytes that pass through the buffer. Therefore, this class gives you functionality for streams as the above member functions give you functionality for files. For more information on class *RWAuditStreamBuffer*, see the *Class Reference.*

Stream I/O

The overloaded left-shift operator `<<`, taking an `ostream` object as its first argument, will print the contents of an object in human-readable form. Conversely, the overloaded right-shift operator `>>`, taking an `istream` object as its first argument, will read and parse an object from the stream in a human-understandable format.

```
ostream&  operator<<(ostream& ostr, const ClassName& x);
istream&  operator>>(istream& istr, const ClassName& x);
```

The overloaded left-shift and right-shift operators contrast with the persistence operators:

```
Rwvostream&  operator<<(RWvostream& vstream, const ClassName&);
Rwvistream&  operator>>(RWvistream& vstream,       ClassName&);
```

Although the persistence shift operators may store and restore to and from a stream, they will not necessarily do so in a form that could be called "human-readable."

Comparisons

Finally, most classes have comparison and equality member functions:

```
int        compareTo(ClassName*) const;
RWBoolean  equalTo(ClassName*) const;
```

and their logical operator counterparts:

```
RWBoolean  operator==(const ClassName&) const;
RWBoolean  operator!=(const ClassName&) const;
RWBoolean  operator<=(const ClassName&) const;
RWBoolean  operator>=(const ClassName&) const;
RWBoolean  operator<(const ClassName&) const;
RWBoolean  operator>(const ClassName&) const;
```

Memory Allocation and Deallocation

When an object is allocated off the heap, who is responsible for deleting it? With some libraries, ownership can be a problem.

Most of the Rogue Wave classes take a very simple approach: if you allocate something off the heap, then you are responsible for deallocating it. If the Rogue Wave library allocates something off the heap, then it is responsible for deallocating it.

There are two exceptions for creation of objects. The first exception involves the operators:

```
RWFile&      operator>>(RWFile& file, RWCollectable*&);
Rwvistream&  operator>>(RWvistream& vstream, RWCollectable*&);
```

These operators restore an object inheriting from *RWCollectable* from an *RWFile* or *RWvistream*, respectively. They return a pointer to an object *allocated off the heap*: you are responsible for deleting it.

The second exception is member function:

```
RWCollection* RWCollection::select(RWtestCollectable,void*)const;
```

This function returns a pointer to a collection, allocated off the heap, with members satisfying some selection criterion. Again, you are responsible for deleting this collection when you are done with it.

There is also an exception for object deletion: As a service, many of the collection classes provide a method, `clearAndDestroy()`, which will remove all pointers from the collection and delete each. Even with `clearAndDestroy()`, however, it is still your responsibility to know that it is safe to delete all the pointers in that collection.

These methods are documented in detail in the *Class Reference*.

Information Flow

With the Rogue Wave libraries, information generally flows into a function via its arguments and out through a return value. Most functions do not modify their arguments. Indeed, if an argument is passed by value or as a const reference:

```
void  foo(const RWCString& a)
```

you can be confident that the argument will not be modified. However, if an argument is passed as a non-const reference, you may find that the function will modify it.

If an argument is passed in as a pointer, there is the strong possibility that the function will retain a copy of the pointer. This is typical of the collection classes:

```
RWOrdered::insert(RWCollectable*);
```

The function retains a copy of the pointer to remind you that the collection will be retaining a pointer to the object after the function returns[1].

1. An alternative design strategy would be to pass objects that are to be inserted into a collection by reference, as in The NIH Classes. We rejected this approach for two reasons: it looks so similar to pass-by-value that the programmer could forget about the retained reference; also, it becomes too easy to store a reference to a stack-based variable.

Multithread Safe

When compiled with the appropriate option according to the release notes for your compiler, *Tools.h++* is multithread safe. In other words, all *Tools.h++* functions behave the same in a multithreaded environment as in a single-threaded environment. Of course, this assumes the application program takes care either to avoid sharing individual objects between threads, or to perform locking around operations on objects shared across threads. *Tools.h++* does enough internal locking to maintain its own internal integrity, and uses appropriate multithread-safe systems calls.

Eight-bit Clean

All classes in *Tools.h++* are eight-bit clean. This means they can be used with eight-bit code sets such as ISO Latin-1.

Embedded Nulls

All classes in *Tools.h++*, including *RWCString* and *RWWString*, support character sets with embedded nulls. This allows them to be used with multibyte character sets.

Indexing

Indexes have type `size_t`, an unsigned integral type defined by your compiler, usually in `<stddef.h>`. Because `size_t` is unsigned, it allows indexes up to 64k minus one under 16-bit DOS.

Invalid indexes are signified by the special value `RW_NPOS`, defined in `<rw/defs.h>`.

Version

When programming, you may need to know the specific version number of *Tools.h++* to perform certain operations. This number is given by the macro `RWTOOLS`, expressed as a hexadecimal number. For example, version 1.2.3 would be 0x123. This can be used for conditional compilations.

If the version is needed at run time, you can find it via the function `rwToolsVersion()`, declared in header file `<rw/tooldefs.h>`.

Table 2-2 The public class hierarchy of the *Tools.h++* classes.

Note that this is the public class hierarchy–the class implementations may use private inheritance. Classes that have multiple inheritance are shown in both places in the hierarchy; their other base is shown in italics to the right.

RWBench

RWBitVec

RWBTreeOnDisk

RWCacheManager

RWCollectable

RWCollection

RWBag

RWBinaryTree

RWBTree

RWBTreeDictionary

RWHashTable

RWSet

RWFactory

RWHashDictionary

RWIdentityDictionary

RWIdentitySet

RWSequenceable

RWDlistCollectables

RWOrdered

RWSortedVector

RWSlistCollectables

RWSlistCollectablesQueue

RWSlistCollectablesStack

RWCollectableDate (&RWDate)

RWCollectableInt (&RWInteger)

RWCollectableString (&RWCString)

RWCollectableTime (&RWTime)

- *RWModelClient*
- *RWCRegexp*
- *RWCRExp*
- *RWCString*
  - *RWCollectableString (&RWCollectable)*
- *RWCSubString*
- *RWCTokenizer*
- *RWDate*
  - *RWCollectableDate (&RWCollectable)*
- *RWErrObject*
- *RWFile*
  - *RWFileManager*
- *RWGBitVec(size)*
- *RWGDlist(type)*
- *RWGDlistIterator(type)*
- *RWGOrderedVector(val)*
- *RWGQueue(type)*
- *RWGSlist(type)*
- *RWGSlistIterator(type)*
- *RWGStack(type)*
- *RWGVector(val)*
  - *RWGSortedVector(val)*
- *RWInteger*
  - *RWCollectableInt (&RWCollectable)*
- *RWIterator*
  - *RWBagIterator*
  - *RWBinaryTreeIterator*
  - *RWDlistCollectablesIterator*
  - *RWHashDictionaryIterator*
  - *RWHashTableIterator*
    - *RWSetIterator*
  - *RWOrderedIterator*
  - *RWSlistCollectablesIterator*
- *RWLocale*

RWLocaleSnapshot

RWMessage

RWModel

RWTime

RWCollectableTime (&RWCollectable)

RWTimer

RWTBitVec<size>

RWTIsvDlist<T>

RWTIsvDlistIterator<TL>

RWTIsvSlist<T>

RWTIsvSlistIterator<TL>

RWTPtrDeque<T>

RWTPtrDlist<T>

RWTPtrDlistIterator<T>

RWTPtrHashMap<Key,Type,Hash,EQ>

RWTPtrHashMapIterator<Key,Type,Hash,EQ>

RWTPtrHashMultiMap<Key,Type,Hash,EQ>

RWTPtrHashMultiMapIterator<Key,Type,Hash,EQ>

RWTPtrHashMultiSet<T,Hash,EQ>

RWTPtrHashMultiSetIterator<T,Hash,EQ>

RWTPtrHashSet<T,Hash,EQ>

RWTPtrHashSetIterator<T,Hash,EQ>

RWTPtrMap<Key,Type,Compare>

RWTPtrMapIterator<Key,Type,Compare>

RWTPtrMultiMap<Key,Type,Compare>

RWTPtrMultiMapIterator<Key,Type,Compare>

RWTPtrMultiSet<T,Compare>

RWTPtrMultiSetIterator<T,Compare>

RWTPtrOrderedVector<T>

RWTPtrSet<T,Compare>

RWTPtrSetIterator<T,Compare>

RWTPtrSlist<T>

RWTPtrSlistIterator<T>

RWTPtrSlistDictionary<KeyP,ValP>

RWTPtrSlistDictionaryIterator<KeyP,ValP>
RWTPtrSortedDlist<T,Compare>
RWTPtrSortedDlistIterator<T,Compare>
RWTPtrSortedVector<T,Compare>
RWTPtrVector<T>
RWTQueue<T,Container>
RWTRegularExpression<charT>
RWTStack<T,Container>
RWTValDeque<T>
RWTValDlist<T>
RWTValDlistIterator<T>
RWTValHashMap<Key,Type,Hash,EQ>
RWTValHashMapIterator<Key,Type,Hash,EQ>
RWTValHashMultiMap<Key,Type,Hash,EQ>
RWTValHashMultiMapIterator<Key,Type,Hash,EQ>
RWTValHashMultiSet<T,Hash,EQ>
RWTValHashMultiSetIterator<T,Hash,EQ>
RWTValHashSet<T,Hash,EQ>
RWTValHashSetIterator<T,Hash,EQ>
RWTValMap<Key,Type,Compare>
RWTValMapIterator<Key,Type,Compare>
RWTValMultiMap<Key,Type,Compare>
RWTValMultiMapIterator<Key,Type,Compare>
RWTValMultiSet<T,Compare>
RWTValMultiSetIterator<T,Compare>
RWTValOrderedVector<T>
RWTValSet<T,C>
RWTValSetIterator<T,C>
RWTValSlist<T>
RWTValSlistIterator<T>
RWTValSlistDictionary<Key,V>
RWTValSlistDictionaryIterator<Key,V>
RWTValSortedDlist<T,Compare>
RWTValSortedDlistIterator<T,Compare>

RWTValSortedVector<T>

RWTValVector<T>

RWTValVirtualArray<T>

RWvios

RWios (virtual)

RWvistream

RWbistream (&ios: virtual)

RWeistream

RWpistream

RWXDRistream (&RWios)

RWvostream

RWbostream (&ios: virtual)

RWeostream

RWpostream

RWXDRostream (&RWios)

RWVirtualPageHeap

RWBufferedPageHeap

RWDiskPageHeap

RWWString

RWWSubString

RWWTokenizer

RWZone

RWZoneSimple

streambuf

RWAuditStreamBuffer

RWCLIPstreambuf

RWDDEstreambuf

xmsg

RWxmsg

RWExternalErr

RWFileErr

RWStreamErr

RWInternalErr

RWBoundsErr

2

RWxalloc

Using the String Classes 3

Manipulating strings is probably one of your most common tasks. Many developers say it is also the most error-prone. The *Tools.h++* classes *RWCString* and *RWWString* give you the constructors, operators, and member functions you need to create, manipulate, and delete strings easily.

The member functions of class *RWCString* read, compare, store, restore, concatenate, prepend, and append *RWCString* objects and `char*`s. Its operators allow access to individual characters, with or without bounds checking. And the class automatically takes care of memory management: you never need to create or delete storage for the string's characters.

Class *RWWString* is similar to *RWCString*, except that *RWWString* works with wide characters. Since the interfaces of the two classes are similar, they can be easily interchanged. Details of these classes are described in the *Class Reference*. This chapter gives you some general examples of how *RWCString* works, followed by discussions of selected features of the string classes.

3

An Introductory Example

The following example calls on several essential features of the string classes. Basically, it shows the steps *RWCString* would take to substitute a new version number for the old ones in a piece of documentation.

```
#include <rw/cstring.h>
#include <rw/regexp.h>
#include <rw/rstream.h>

main(){
  RWCString a;                      //1 create string object a

  RWCRegexp re("V[0-9]\\.[0-9]+"); //2 define regular expression
  while( a.readLine(cin) ){         //3 read standard input into a
    a(re) = "V4.0";                 //4 replace matched expression
    cout << a << endl;
  }
  return 0;
}
```

Program Input:

```
This text describes V1.2.  For more
information see the file install.doc.
The current version V1.2 implements...
```

Program Output:

```
This text describes V4.0.  For more
information see the file install.doc.
The current version V4.0 implements...
```

The code here describes the activity of the class. *RWCString* creates a string object `a`, reads lines from standard input into `a`, and searches `a` for a pattern matching the defined regular expression `"V[0-9]\\.[0-9]+"`. A match would be a version number between `V0` and `V9`; for example, `V1.2` and `V1.22`, but not `V12.3`. When a match is found, it is replaced with the string "`V4.0`"

The power of this operation lies in the expression:

```
a(re) = "V4.0";
```

where`()` is an example of an overloaded operator. As you know, an overloaded operator is one which can perform more than one function, depending on context or argument.

In the example, the function call operator `RWCString::operator()`is overloaded to take an argument of type *RWCRegexp*, the regular expression. The operator returns either a substring that delimits the regular expression, or a null substring if a matching expression cannot be found. The program then calls the substring assignment operator, which replaces the delimited string with the contents of the right hand side, or does nothing if this is the null substring. Because *Tools.h++* provides the overloaded operator, you can do a search and replace on the defined regular expression all in a single line.

You will notice that you need two backlashes in "`V[0-9]\\.[0-9]+`" to indicate that the special character "`.`" is to be read literally as a decimal point. That's because the compiler removes one backslash when it evaluates a literal string. The remaining backslash alerts the regular expression evaluator to read whatever character follows literally.

In the next example, *RWCString* uses another overloaded operator, `+` , to concatenate the strings `s1` and `s2`. The `toUpper` member function converts the strings from lower to upper case, and the results are sent to `cout`:

```
RWCString s1, s2;
cin >> s1 >> s2;
cout << toUpper(s1+s2);
```

See the *Class Reference* for details on the string classes.

Lexicographic Comparisons

If you're putting together a dictionary, you'll find the lexicographics comparison operators of *RWCString* particularly useful. They are:

```
RWBoolean operator==(const RWCString&, const RWCString&);
RWBoolean operator!=(const RWCString&, const RWCString&);
RWBoolean operator< (const RWCString&, const RWCString&);
RWBoolean operator<=(const RWCString&, const RWCString&);
RWBoolean operator> (const RWCString&, const RWCString&);
RWBoolean operator>=(const RWCString&, const RWCString&);
```

These operators are case sensitive. If you wish to make case insensitive comparisons, you can use the member function:

```
int RWCString::compareTo(const RWCString& str,
                         caseCompare = RWCString::exact) const;
```

Here the function returns an integer less than zero, equal to zero, or greater than zero, depending on whether `str` is lexicographically less than, equal to, or greater than self. The type `caseCompare` is an enum with values:

`exact`	Case sensitive
`ignoreCase`	Case insensitive

Its default setting is `exact`, which gives the same result as the logical operators `==`, `!=`, etc.

For locale-specific string collations, you would use the member function:

```
int RWCString::collate(const RWCString& str) const;
```

which is an encapsulation of the Standard C library function `strcoll()`. This function returns results computed according to the locale-specific collating conventions set by category `LC_COLLATE` of the Standard C library function `setlocale()`. Because this is a relatively expensive calculation, you may want to pretransform one or more strings using the global function:

```
RWCString strXForm(const RWCString&);
```

then use `compareTo()` or one of the logical operators, `==`, `!=`, etc., on the results. See the *Class Reference* entry for *RWCString*: the function `strxForm` appears under related global functions.

Substrings

A separate *RWCSubString* class supports substring extraction and modification. There are no public constructors; *RWCSubString* are constructed indirectly by various member functions of *RWCString*, and destroyed at the first opportunity.

You can use substrings in a variety of situations. For example, you can create a substring with `RWCString::operator()`, then use it to initialize an *RWCString*:

```
RWCString s("this is a string");
// Construct an RWCString from a substring:
RWCString s2 = s(0, 4);                                  // "this"
```

The result is a string `s2` that contains a copy of the first four characters of `s`.

You can also use *RWSubString* as lvalues in an assignment to a character string, or to an *RWCString* or *RWCSubString*:

```
// Construct an RWCString:
RWCString article("the");
RWCString s("this is a string");
s(0, 4) = "that";                        // "that is a string"
s(8, 1) = article;                      // "that is the string"
```

Note that assignment to a substring is *not* a conformal operation: the two sides of the assignment operator need not have the same number of characters.

Pattern Matching

Class *RWCString* supports a convenient interface for string searches. In the example below, the code fragment:

```
RWCString s("curiouser and curiouser.");
size_t i = s.index("curious");
```

will find the start of the first occurrence of `curious` in `s`. The comparison will be case sensitive, and the result will be that `i` is set to 0. To find the index of the next occurrence, you would use:

```
i = s.index("curious", ++i);
```

which will result in `i` set to 14. You can make a case-insensitive comparison with:

```
RWCString s("Curiouser and curiouser.");
size_t i = s.index("curious", 0, RWCString::ignoreCase);
```

which will also result in `i` set to 0.

If the pattern does not occur in the string, the `index()` will return the special value `RW_NPOS`.

Simple Regular Expressions

As part of its pattern matching capability, the *Tools.h++* Class Library supports regular expression searches. See the *Class Reference*, under *RWCRegexp*, for details of the regular expression syntax. You can use a regular expression to return a substring; for example, here's how you might match all Windows messages (prefix `WM_`):

```
#include <rw/cstring.h>
#include <rw/regexp.h>
#include <rw/rstream.h>

main(){
  RWCString a("A message named WM_CREATE");

  // Construct a Regular Expression to match Windows messages:
  RWCRegexp re("WM_[A-Z]*");
  cout << a(re) << endl;

  return 0;
}
```

Program Output:

```
WM_CREATE
```

The function call operator for *RWCString* has been overloaded to take an argument of type *RWCRegexp*. It returns an *RWCSubString* matching the expression, or the null substring if there is no such expression.

Extended Regular Expressions

This version of the *Tools.h++* class library supports extended regular expression searches based on the POSIX.2 standard. (See the Bibliography.) Extended regular expressions are the regular expressions used in the UNIX utilities lex and awk. You will find details of the regular expression syntax in the *Class Reference* under *RWCRExpr*.

Note – *RWCRExpr* is available only if your compiler supports exception handling and the C++ Standard Library.

Extended regular expressions can be any length, although limited by available memory. You can use parentheses to group subexpressions, and the symbol | to create either/or regular expressions for pattern matching.

The following example shows some of the capabilities of extended regular expressions:

```
#include "rw/rstream.h"
#include "rw/re.h"

main (){
  RWCRExpr  re("Lisa|Betty|Eliza");
  RWCString s("Betty II is the Queen of England.");

  s.replace(re, "Elizabeth");
  cout << s << endl;

  s = "Leg Leg Hurrah!";
  re = "Leg";
  s.replace(re, "Hip", RWCString::all);
  cout << s << endl;
}
```

Program Output:

```
Elizabeth II is the Queen of England.
Hip Hip Hurrah!
```

Note that the function call operator for *RWCString* has been overloaded to take an argument of type *RWCRExpr*. It returns an *RWCSubString* matching the expression, or the null substring if there is no such expression.

3

String I/O

Class *RWCString* offers a rich I/O facility to and from both iostreams and Rogue Wave virtual streams.

iostreams

The standard left-shift and right-shift operators have been overloaded to work with iostreams and *RWCString*s:

```
ostream&operator<<(ostream& stream, const RWCString& cstr);
istream&operator>>(istream& stream, RWCString& cstr);
```

The semantics parallel the operators:

```
ostream&operator<<(ostream& stream, const char*);
istream&operator>>(istream& stream, char* p);
```

which are defined by the Standard C++ Library that comes with your compiler. In other words, the left-shift operator << writes a null-terminated string to the given output stream. The right-shift operator >> reads a single token, delimited by white space, from the input stream into the *RWCString*, replacing the previous contents.

Other functions allow finer tuning of *RWCString* input[1]. For instance, function `readline()` reads strings separated by newlines. It has an optional parameter controlling whether white space is skipped before storing characters. You can see the difference skipping white space makes in the following example:

```
#include <rw/cstring.h>
#include <iostream.h>
#include <fstream.h>

main(){
   RWCString line;

  { int count = 0;
    ifstream istr("testfile.dat");

    while (line.readLine(istr))            // Use default value:
                                              // skip whitespace
      count++;
    cout << count << " lines, skipping whitespace.\n";
  }

  { int count = 0;
    ifstream istr("testfile.dat");
    while (line.readLine(istr, FALSE))       // NB: Do not skip
                                                   // whitespace
      count++;
    cout << count << " lines, not skipping whitespace.\n";
  }

  return 0;
}
```

1. Details about methods `readFile(); readLine(); readString(istream&); readToDelim();` and `readToken()` may be found in the *RWCString* section of the *Class Reference*.

Program Input:

```
line 1



line 5
```

Program Output:

```
2 lines, skipping whitespace.
5 lines, not skipping whitespace.
```

Virtual Streams

String operators to and from Rogue Wave virtual streams are also supported:

```
Rwvistream&  operator>>(RWvistream& vstream, RWCString& cstr);
Rwvostream&  operator<<(RWvostream& vstream,
                        const RWCString& cstr);
```

By using these operators, you can save and restore a string without knowing its formatting. See Chapter 6 for details on virtual streams.

Tokenizer

You can use the class *RWCTokenizer* to break up a string into tokens separated by arbitrary white spaces. Here's an example:

```
#include <rw/ctoken.h>
#include <rw/cstring.h>
#include <rw/rstream.h>

main(){
  RWCString a("a string with five tokens");

  RWCTokenizer next(a);

  int i = 0;

  // Advance until the null string is returned:
  while( !next().isNull() ) i++;

  cout << i << endl;
  return 0;
}
```

Program Output:

```
5
```

This program counts the number of tokens in the string. The function call operator for class *RWCTokenizer* has been overloaded to mean "advance to the next token and return it as an *RWCSubString*," much like other *Tools.h++* iterators. When there are no more tokens, it returns the null substring. Class *RWCSubString* has a member function `isNull()` which returns `TRUE` if the substring is the null substring. Hence, the loop is broken. See the *Class Reference* under *RWCTokenizer* for details.

3

Multibyte Strings

Class *RWCString* provides limited support for multibyte strings, sometimes used in representing various alphabets (see “Localizing Alphabets with RWCString and RWWString” on page 217). Because a multibyte character can consist of two or more bytes, the length of a string in bytes may be greater than or equal to the number of actual characters in the string.

If the *RWCString* contains multibyte characters, you should use member function `mbLength()` to return the number of characters. On the other hand, if you know that the *RWCString* does not contain any multibyte characters, then the results of `length()` and `mbLength()` will be the same, and you may want to use `length()` because it is much faster. Here's an example using a multibyte string in `Sun`:

```
RWCString Sun("\306\374\315\313\306\374");
cout << Sun.length();                              // Prints "6"
cout << Sun.mbLength();                            // Prints "3"
```

The string in `Sun` is the name of the day Sunday in Kanji, using the EUC (Extended UNIX Code) multibyte code set. With the EUC, a single character may be 1 to 4 bytes long. In this example, the string `Sun` consists of 6 bytes, but only 3 characters.

In general, the second or later byte of a multibyte character may be null. This means the length in bytes of a character string may or may not match the length given by `strlen()`. Internally, *RWCString* makes no assumptions[1] about embedded nulls, and hence can be used safely with character sets that use null bytes. You should also keep in mind that while `RWCString::data()` always returns a null-terminated string, there may be earlier nulls in the string. All of these effects are summarized in the following program:

1. However, system functions to transfer multibyte strings may make such assumptions. *RWCString* simply calls such functions to provide such transformations.

```
#include <rw/cstring.h>
#include <rw/rstream.h>
#include <string.h>
main() {
RWCString a("abc");                                          // 1
RWCString b("abc\0def");                                     // 2
RWCString c("abc\0def", 7);                                  // 3

cout << a.length();                                 // Prints "3"
cout << strlen(a.data());                           // Prints "3"

cout << b.length();                                 // Prints "3"
cout << strlen(b.data());                           // Prints "3"

cout << c.length();                                 // Prints "7"
cout << strlen(c.data());                           // Prints "3"
return 0; }
```

You will notice that two different constructors are used above. The constructor in lines 1 and 2 takes a single argument of `const char*`, a null-terminated string. Because it takes a single argument, it may be used in type conversion (ARM 12.3.1). The length of the results is determined the usual way, by the number of bytes before the null. The constructor in line 3 takes a `const char*` and a run length. The constructor will copy this many bytes, *including any embedded nulls.*

The length of an *RWCString* in bytes is always given by `RWCString::length()`. Because the string may include embedded nulls, this length may not match the results given by `strlen()`.

Remember that indexing and other operators—basically, all functions using an argument of type `size_t`—work in *bytes.* Hence, these operators will not work for *RWCString*s containing multibyte strings.

3

Wide Character Strings

Class *RWWString*, also used in representing various alphabets, is similar to *RWCString* except it works with wide characters. These are much easier to manipulate than multibyte characters because they are all the same size: the size of a `wchar_t`.

Tools.h++ makes it easy to convert back and forth between multibyte and wide character strings. Here's an example of how to do it, built on the `Sun` example in the previous section:

```
#include <rw/cstring.h>
#include <rw/wstring.h>
#include <assert.h>
main() {
RWCString Sun("\306\374\315\313\306\374");
RWWString wSun(Sun, RWWString::multiByte); // MBCS to wide string

RWCString check = wSun.toMultiByte();
assert(Sun==check);                                    // OK
return 0; }
```

Basically, you convert from a multibyte string to a wide string by using the special *RWWString* constructor:

```
RWWString(const char*, multiByte_);
```

The parameter `multiByte_` is an enum with a single possible value, `multiByte`, as shown in the example. The `multiByte` argument ensures that this relatively expensive conversion is not done inadvertently. The conversion from a wide character string back to a multibyte string, using the function `toMultiByte()`, is similarly expensive.

If you know that your *RWCString* consists entirely of ASCII characters, you can greatly reduce the cost of the conversion in both directions. This is because the conversion involves a simple manipulation of high-order bits:

```
#include <rw/cstring.h>
#include <rw/wstring.h>
#include <assert.h>
main() {
RWCString EnglishSun("Sunday");                  // ASCII string
assert(EnglishSun.isAscii());                              // OK

// Now convert from Ascii to wide characters:
RWWString wEnglishSun(EnglishSun, RWWString::ascii);

assert(wEnglishSun.isAscii());                             // OK
RWCString check = wEnglishSun.toAscii();
assert(check==EnglishSun);                                 // OK
return 0; }
```

Note how the member functions `RWCString::isAscii()` and `RWWString::isAscii()` are used to ensure that the strings consist entirely of ASCII characters. The *RWWString* constructor:

```
RWWString(const char*, ascii_);
```

is used to convert from ASCII to wide characters. The parameter `ascii_` is an enum with a single possible value, `ascii`.

The member function `RWWString::toAscii()` is used to convert back.

3

Using Class RWDate 4

Class *RWDate* represents a date, stored as a Julian day number. Commonly used in software, this compact representation allows rapid calendar calculations, shields you from details such as leap years, and performs easy conversions to and from conventional calendar formats.

You don't need to know Julian day numbers to benefit from their use in *Tools.h++*. If you are interested, the algorithm *Tools.h++* uses to convert common calendar dates to Julian day numbers is given in "Algorithm 199" from *Communications of the ACM*, Volume 6, No. 8, Aug. 1963, p. 444.Gregorian calendar

The Gregorian calendar now used nearly world-wide was introduced by Pope Gregory XIII in 1582, and adopted in various places at various times. It was adopted by England on September 14, 1752, and thus came to the United States. We mention this because an *RWDate* for a day prior to the adoption of the Gregorian calendar is only valid in the sense that it is an extrapolation back from the Gregorian system. Printing such an *RWDate* , or using its methods to deal with specific day or month names, may have unexpected results.

Example

The point is that *RWDate* allows you to quickly and easily manipulate the calendar dates you're most likely to use. Here is an example that demonstrates the virtuosity of the class.

Let's print out the date when ENIAC first started, 14 February 1945, then calculate and print the date of the previous Sunday, using the global locale:

```
#include <rw/rwdate.h>
#include <rw/rstream.h>

int main(){
  // ENIAC start date
  RWDate d(14, "February", 1945);

  // Today
  RWDate today;

  cout << d.asString("%A, %B %d 19%y")
       << " was the day the ENIAC computer was" << endl
       << "first turned on.  "
      << today - d << " days have gone by since then. " << endl;

  return 0;
}
```

Program Output:

```
Wednesday February 14, 1945 was the day the ENIAC computer was
first turned on.  18636 days have gone by since then.
```

In this calculation, notice that the number of days that have passed depends on when you run the program.

Constructors

You can construct an *RWDate* in several ways. For example:

1. Construct an *RWDate* with the current date[1]:

```
RWDate d;
```

1. Because the default constructor for *RWDate* fills in today's date, constructing a large array of *RWDate* may be slow. If this is an issue, declare your arrays with a class derived from *RWDate* that provides a faster constructor, or use *RWTValOrderedVector<RWDate>*.

2. Construct an *RWDate* for a given day of the year (1–365) and a given year, e.g., 1989 or 89. Although the class supports 2-digit year specifiers, we urge you to use the 4-digit variety if possible to avoid difficulties at the turn of the century.

```
RWDate d1(24, 2001);                    // 1/24/2001
RWDate d2(24, 01);                    // 1/24/1901 (oops)
```

3. Construct an *RWDate* for a given day of the month (1–31), month number (1–12), and year:

```
RWDate d(10, 3, 2015);                          // 3/10/2015
```

4. Construct an *RWDate* from an *RWTime*:

```
RWTime t;                          // Current time.
RWDate d(t);
```

In addition, you can construct a date using locale-specific strings. If you do nothing, a default locale using United States conventions and names is applied:

```
RWDate d1(10, "June", 2001);                          // 6/10/2001
RWDate d2(10, "JUN", 2001);                           // 6/10/2001
```

But suppose you need to use French month names. Assuming your system supports a French locale, here's how you might do it:

```
dateloc.cpp
#include <rw/rwdate.h>
#include <rw/rstream.h>
#include <rw/locale.h>
#include <rw/cstring.h>

main()
{
  RWLocaleSnapshot us("C");
  RWLocaleSnapshot french("fr");      // or vendor specific // 1

  RWCString americanDate("10 June 2025");
  RWCString frenchDate("10 Juin 2025");

  RWDate d(frenchDate, french);       // OK                // 2
```

```
  cout << frenchDate << ((d.isValid()) ? " IS " : " IS NOT ")
       << "a valid date (french locale)." << endl << endl;

  RWDate bad = RWDate(frenchDate);                          // 3
  cout << frenchDate;
  cout << ((bad.isValid() && bad == d) ? " IS " : " IS NOT ")
       << "a valid date (default locale)." << endl << endl;

  bad = RWDate(americanDate, french);                       // 4
  cout << americanDate;
  cout << ((bad.isValid() && bad == d) ? " IS " : " IS NOT ")
       << "a valid date (french locale)." << endl << endl;

  cout << d << endl;                                        // 5
  cout << d.asString() << endl;                             // 6
  cout << d.asString('x', french) << endl;                  // 7

  return 0;
}
```

Here's a line-by-line description of the previous code:

1. A snapshot is taken of locale `fr`. This assumes your system supports the locale. Another common name for this locale is `fr_FR`.

2. A date is constructed using the constructor:

   ```
   RWDate(unsigned day,const char* month,unsigned year,
          const RWLocale& locale = RWLocale::global());
   ```

   Note that the second argument `month` is meaningful only within the context of a locale. In this case, we are using the locale constructed at line 1. The result is the date known in English as June 10, 2002.

3. Here we attempt to construct the same date using the default locale. This locale recognizes C formatting conventions only. Hence, the date 10 June 2002 should be meaningless. Just in case, though, compare with a known valid date.

4. For the same reason, constructing a date using United States names with a French locale should fail. Just in case, though, compare with a known valid date.

5. The date constructed at line 2 is printed using the default locale, i.e., United States formatting conventions. The results are:

   06/10/25

6. The date is converted to a string, then printed. Again, the default locale is used. The results are the same:

   06/10/25

7. The date is converted to a string, this time using the locale constructed at line 1. The results are now[1]:

   10.06.25

1. Your system's locale files determine the format used.

4

Using Class RWTime 5

Class *RWTime* represents time, stored as the number of seconds since 1 January 1901 UTC. UTC is sometimes called GMT, for Greenwich Meridian Time. The number of seconds that can be stored is limited by the size of a `long` on your system. The last date and time that can be represented with a four-byte (32-bit) `long` is 22:28:15 February 5, 2037 UTC.

Class *RWTime* uses UTC because it is a widely accepted standard, useful in calculations, but it is not the usual time reference people use in their daily lives. We tell time with a local time which may or may not observe daylight-saving time (DST) conventions; in fact, DST may or may not be in effect.

When we create an *RWTime* object to represent the current time, the library obtains the current UTC time directly from the operating system. However, when we create an *RWTime* object for some specific time, we are unlikely to do so with UTC. More likely, the time we give it will be with respect to some other time zone, and we must specify which time zone for *RWTime* to do its job, or even print out the time. So by default, *RWTime* uses a global local time, set by `RWZone::local()`.

Setting the Time Zone

The question naturally arises, how does the library determine this local time?

The UNIX operating system provides for setting the local time zone and for establishing whether DST is locally observed. Class *RWTime* uses various system calls to determine these values and sets itself accordingly. Class

RWTime should function properly in North America or places where DST is not observed. In places not governed by United States DST rules, you may need to re-initialize the local time zone—see *RWZone* in the *Class Reference*.

Users of the various Windows operating systems may have to set the time switches manually. How you do this depends on your compiler. If you do nothing, the class will function properly for local time, but may not give the proper GMT because the computer has no way of knowing the offset from local time to GMT.

If you use Borland, MetaWare, Microsoft, Symantec, or Watcom, you must set your environment variable `TZ` to the appropriate time zone. For example:

```
set TZ=PST8PDT
```

For further information, see the documentation for function `tzset()`or `_tzset()` in your compiler's run-time library reference.

Finally, it is essential that your computer's system clock be set and functioning correctly. If you are using a PC, be sure the batteries that power the system clock are charged.

Constructors

An *RWTime* may be constructed in several ways:

1. Construct an *RWTime* with the current time:

   ```
   RWTime t;
   ```

2. Construct an *RWTime* with today's date, at the specified local hour (0-23), minute (0-59), and second (0-59):

   ```
   RWTime t(16, 45, 0);                          // today, 16:45:00
   ```

3. Construct an *RWTime* for a given date and local time:

   ```
   RWDate d(2, "June", 1952);
   RWTime t(d, 16, 45, 0);                     // 6/2/52 16:45:003
   ```

4. Construct an *RWTime* for a given date and time zone:

   ```
   RWDate d(2, "June", 1952);
   RWTime t(d, 16, 45, 0, RWZone::utc());       // 6/2/52 16:45:00
   ```

Member Functions

Class *RWTime* has member functions to compare, store, restore, add, and subtract *RWTimes*. An *RWTime* may return hours, minutes or seconds, or fill a *struct tm* for any time zone. A complete list of member functions is included in the *Class Reference.*

For example, here is a code fragment that outputs the hour in local and UTC zones, and then the complete local time and date:

```
RWTime t;
cout << t.hour() << endl;                          // Local hour
cout << t.hour(RWZone::utc()) << endl;               // UTC hour
cout << t.asString('c') << endl;           // Local time and date
```

See the definition for `c` and other format characters for time under the entry for *RWLocale* in the *Class Reference.* The next example shows how you find out when daylight-saving time starts for the current year and local time zone:

```
RWDate today;                                      // Current date
RWTime dstStart = RWTime::beginDST(today.year(), RWZone::local());
```

In order to ensure that this will give the right results for time zones outside of North America, you should reset `RWZone::local()` with an *RWZoneSimple* equipped with an appropriate daylight-saving time rule. For more information see *RWZoneSimple* in the *Class Reference*. See the entry for `c` and other format characters for time in *RWLocale* in the *Class Reference.*

5

Using Virtual Streams 6

The iostream facility that comes with every C++ compiler is a resource that should be familiar to you as a C++ developer. Among its advantages are type-safe insertion and extraction into and out of streams, extensibility to new types, and transparency to the user of the source and sink of the stream bytes, which are set by the class *streambuf*.

But the iostream facility suffers from a number of limitations. Formatting abilities are particularly weak; for example, if you insert a double into an *ostream*, there is no type-safe way to insert it as binary. Furthermore, not all byte sources and sinks fit into the *streambuf* model. For many protocols, such as XDR, the format is intrinsically wedded to the byte stream and cannot be separated.

The Rogue Wave virtual streams facility overcomes these limitations by offering an idealized model of a stream. No assumptions are made about formatting, or stream models. At the root of the virtual streams class hierarchy is class *RWvios*. This is an abstract base class with an interface similar to the standard library class *ios:*

```
class RWvios{
public:
  virtual int   eof()            = 0;
  virtual int   fail()           = 0;
  virtual int   bad()            = 0;
  virtual int   good()           = 0;
  virtual int   rdstate()        = 0;
  virtual int   clear(int v = 0)  = 0;
};
```

Classes derived from *RWvios* will define these functions.

Inheriting from *RWvios* are the abstract base classes *RWvistream* and *RWvostream*. These classes declare a suite of pure virtual functions such as `operator<<(), put(), get()`, and the like, for all the basic built-in types and arrays of built-in types:

```
class RWvistream : public RWvios {
public:
  virtual Rwvistream&  operator>>(char&)       = 0;
  virtual Rwvistream&  operator>>(double&)     = 0;
  virtual int          get()                   = 0;
  virtual Rwvistream&  get(char&)              = 0;
  virtual Rwvistream&  get(double&)            = 0;
  virtual Rwvistream&  get(char*, size_t N)    = 0;
  virtual Rwvistream&  get(double*, size_t N)  = 0;
  .
  .
  .
};

class RWvostream : public RWvios {
public:
  virtual Rwvostream&  operator<<(char)             = 0;
  virtual Rwvostream&  operator<<(double)           = 0;
  virtual Rwvostream&  put(char)                    = 0;
  virtual Rwvostream&  put(double)                  = 0;
  virtual Rwvostream&  put(const char*, size_t N)   = 0;
  virtual Rwvostream&  put(const double*, size_t N) = 0;
  .
  .
  .
};
```

Streams that inherit from *RWvistream* and *RWvostream* are intended to store built-ins to specialized streams in a format that is transparent to the user of the classes.

The basic abstraction of the virtual streams facility is that built-ins are inserted into a virtual output stream, and extracted from a virtual input stream, *without any regard for formatting*. In other words, there is no need to pad output with whitespace, commas, or any other kind of formatting. You are effectively telling *RWvostream*, “Here is a double. Please store it for me in whatever format is convenient, and give it back to me in good shape when I ask for it.”

The results are extremely powerful. You can write and use streaming operators without knowing anything about the final output medium or formatting to be used. For example, the output medium could be a disk, memory allocation, or even a network. The formatting could be in binary, ASCII, or network packet. In all of these cases, you use the same streaming operators.

Specializing Virtual Streams

The Rogue Wave classes include four types of classes that specialize *RWvistream* and *RWvostream*. The first uses a portable ASCII formatting, the second and third a binary formatting, and the fourth an XDR formatting (eXternal Data Representation, a Sun Microsytems standard):

	Input class	**Output class**
Abstract base class	*RWvistream*	*RWvostream*
Portable ASCII	*RWpistream*	*RWpostream*
Binary	*RWbistream*	*RWbostream*
Endian	*RWeistream*	*RWeostream*
XDR	*RWXDRistream*	*RWXDRostream*

The portable ASCII versions store their inserted items in an ASCII format that escapes special characters (such as tabs, newlines, etc.) in such a manner that they will be restored properly, even under a different operating system. The binary versions do not reformat inserted items, but store them instead in their native format. The endian versions allow for the space and time efficiency of binary format, but can store or retrieve the information in big endian, little endian, or native format. XDR versions send their items to an XDR stream, to be transmitted remotely over a network.

None of these versions retain any state: they can be freely interchanged with regular streams, including XDR. Using them does not lock you into doing all your file I/O with them. For more information, see the respective entries in the *Class Reference*.

6

Simple Example

Here's a simple example that exercises *RWbostream* and *RWbistream* through their respective abstract base classes, *RWvostream* and *RWvistream*:

```
#include <rw/bstream.h>
#include <rw/cstring.h>
#include <fstream.h>

#ifdef __BORLANDC__
# define MODE ios::binary                                        // 1
#else
# define MODE 0
#endif

void save(const RWCString& a, RWvostream& v){
  v << a;                   // Save to the virtual output stream
}

RWCString recover(RWvistream& v) {
   RWCString dupe;
   v >> dupe;           // Restore from the virtual input stream
   return dupe;
}

main(){
   RWCString a("A string with\ttabs and a\nnewline.");

   {
     ofstream f("junk.dat", ios::out|MODE);                      // 2
     RWbostream bostr(f);                                        // 3
     save(a, bostr);
   }                                                             // 4

   ifstream f("junk.dat", ios::in|MODE);                         // 5
   RWbistream bistr(f);                                          // 6
   RWCString b = recover(bistr);                                 // 7

   cout << a << endl;  // Compare the two strings                // 8
   cout << b << endl;
   return 0;
}
```

Program Output:

```
A string with   tabs and a
newline.
A string with   tabs and a
newline.
```

The job of function `save(const RWCString& a, RWvostream& v)` is to save the string `a` to the virtual output stream `v`. Function `recover(RWvistream&)` restores the results. These functions do not know the ultimate format with which the string will be stored. Here are some additional comments on particular lines:

`//1, //2` On these lines, a file output stream `f` is created for the file `junk.dat`. The default file open mode for many PC compilers is `text`, requiring that the explicit flag `ios::binary` be used to avoid automatic DOS new line conversion[1].

`//3` On this line, an *RWbostream* is created from `f`.

`//4` Because this clause is enclosed in braces `{ ... }`, the destructor for `f` will be called here. This will cause the file to be closed.

`//5` The file is reopened, this time for input.

`//6` Now an *RWbistream* is created from it.

`//7` The string is recovered from the file.

`//8` Finally, both the original and recovered strings are printed for comparison.

1. With many PC compilers, even `ostream::write()` and `istream::read()` perform a text conversion unless the file is opened with the `ios::binary` flag.

You could simplify this program by using class *fstream*, which multiply inherits *ofstream* and *ifstream*, for both output *and* input. A seek to beginning-of-file would occur before reading the results back in. Since some early implementations of `seekg()` have not proven reliable, the simpler approach was not chosen for this example.

6

Windows Clipboard and DDE Streambufs

In the previous section, you saw how the virtual streams facility abstracts the formatting of items inserted into the stream. The disposition of the items inserted into the streams has also been made abstract: it is set by the type of *streambuf* used.

Class *streambuf* is the underlying sequencing layer of the iostreams facility. It is responsible for producing and consuming sequences of characters. Your compiler comes with several versions. For example, class *filebuf* ultimately gets and puts its characters to a file. Class *strstreambuf* gets and puts to memory-based character streams; you can think of it as the *iostream* equivalent to ANSI-C's `sprintf()` function. Now *Tools.h++* adds two Windows-based extensions:

- Class *RWCLIPstreambuf* for getting and putting to the Windows Clipboard;
- Class *RWDDEstreambuf* for getting and putting through the Windows Dynamic Data Exchange (DDE) facility.

These classes take care of the details of allocating and reallocating memory from Windows as buffers overflow and underflow. In the case of class *RWDDEstreambuf*, the associated `DDEDATA` header is also filled in for you. Any class that inherits from class *ios* can be used with these *streambuf*s, including the familiar *istream* and *ostream*, as well as the Rogue Wave virtual stream classes.

The result is that the same code that is used to store a complex structure to a conventional disk-based file, for example, can also be used to transfer that structure through the DDE facility to another application!

DDE Example

Let's look at a more complicated example of how you might use class *RWDDEstreambuf* to exchange an *RWBinaryTree* through the Windows DDE facility. You would use a similar technique for the Windows Clipboard.

```
#include <rw/bintree.h>
#include <rw/collstr.h>
#include <rw/bstream.h>
#include <rw/winstrea.h>
#include <windows.h>
#include <dde.h>

BOOL
PostCollection(HWND hwndServer, WORD cFormat){
   RWBinaryTree sc;                                          // 1
   sc.insert(new RWCollectableString("Mary"));
   sc.insert(new RWCollectableString("Bill"));
   sc.insert(new RWCollectableString("Pierre"));

   // Allocate an RWDDEstreambuf and use it to initialize
   // an RWbostream:
   RWDDEstreambuf* sbuf = new RWDDEstreambuf(cFormat,        // 2
                                             FALSE,          // 3
                                             TRUE,           // 4
                                             TRUE);          // 5
   RWbostream bostr( sbuf );                                 // 6

   // Store the collection to the RWbostream:
   bostr << sc;                                              // 7

   // Lock the output stream, and get its handle:
   HANDLE hDDEData = sbuf->str();                            // 8

   // Get an atom to identify the DDE Message:
   ATOM atom = GlobalAddAtom("SortedNames");                 // 9

   // Post the DDE response:
   return PostMessage(0xFFFF, WM_DDE_DATA, hwndServer,       //10
                      MAKELONG(hDDEData, atom));
}
```

In the code above, the large memory model has been assumed. Here's the line-by-line description:

`//1` An *RWBinaryTree* is built and some items inserted into it.

`//2-//5` An *RWDDEstreambuf* is allocated. The constructor takes several arguments. The first argument is the Windows Clipboard format. In this example, the format type has been passed in as an argument, but in general, you will probably want to register a format with Windows (using `RegisterClipboardFormat()`) and use that.

The other arguments have to do with the intricacies of DDE data exchange acknowledgments and memory management. See the *Class Reference* for the list of arguments; for their meanings, see Petzold (1990), Chapter 17, or the *Microsoft Windows Guide to Programming*.

`//6` An *RWbostream* is constructed from the supplied *RWDDEstreambuf*. We could have used an *RWpostream* here, but DDE exchanges are done within the same machine architecture so, presumably, it is not worth the extra overhead of using the portable ASCII formats. Nevertheless, note how the disposition of the bytes, which is set by the type of *streambuf*, is cleanly separated from their formatting, which is set by the type of *RWvostream*.

`//7` The collection is saved to the *RWbostream*. Because the *streambuf* associated with *RWbostream* is actually an *RWDDEstreambuf*, the collection is actually being saved to a Windows global memory allocation with characteristic `GMEM_DDESHARE`. This allocation is resized automatically if it overflows. Like any other *strstreambuf*, you can change the size of the allocation chunks using member function `setbuf()`.

`//8` The *RWDDEstreambuf* is locked. Once locked using `str()`, this *streambuf*, like any other *strstreambuf*, cannot be used again. Note, however, that `RWDDEstreambuf::str()` returns a handle, rather than a `char*`. The handle is unlocked before returning it.

`//9` An atom is constructed to identify this DDE data.

`//10` The handle returned by `RWDDEstreambuf::str()`, along with its identifying atom, is posted.

A similar and actually simpler technique can be used for Clipboard exchanges.

Note that there is nothing that constrains you to use the specialized *streambuf*s R*WCLIPstreambuf* and *RWDDEstreambuf* with only the Rogue Wave virtual streams facility. You could quite easily use them with regular *istreams* and *ostreams*; you just wouldn't be able to set the formatting at run time.

RWAuditStreamBuffer

Classes `RWDDEstreambuf` and `RWCLIPstreambuf` specialize *streambuf* to hand off the characters according to the Windows API. But there are other useful specializations of a `streambuf`. Class *RWAuditStreamBuffer* allows you to count the bytes of any stream, while optionally calling a function of your choice for each character. See the code example in the *Class Reference*.

Recap

In this section, you have seen how an object can be stored to and recovered from a stream without regard for the final destination of the bytes of that stream, whether memory or disk. You have also seen that you need not be concerned with the final formatting of the stream, whether ASCII or binary.

You can also write your own specializing virtual stream class, much like *RWpostream* and *RWpistream*. The great advantage of the virtual streams facility is that, if you do write your own specialized virtual stream, you don't have to modify any of the code of the client classes—you just use your stream class as an argument to:

```
RWvostream& operator<<(RWvostream&, const ClassName&);
RWvistream& operator>>(RWvistream&,       ClassName&);
```

In addition to storing and retrieving an object to and from virtual streams, all of the classes can store and retrieve themselves in binary to and from an *RWFile*. This file encapsulates ANSI-C style file I/O. Although more limited in its abilities than stream I/O, this form of storage and retrieval is slightly faster to and from disk because the virtual dispatching machinery is not needed.

6

Using Class RWFile 7

Class *RWFile* encapsulates the standard C file operations for binary read and write, using the ANSI-C functions `fopen()`, `fwrite()`, `fread()`, etc. This class is patterned on class *PFile* of the *Interviews Class Library* (Stanford University, 1987), but has been modernized by Rogue Wave to use `const` modifiers, and to port to various operating systems. The member function names begin with upper case letters in order to maintain compatibility with class *PFile*.

The constructor for class *RWFile* has the prototype:

```
RWFile(const char* filename, const char* mode = 0);
```

This constructor will open or create a binary file called `filename` with mode set to `mode` (for example, `r+`), as defined by the Standard C function `fopen()`. If `mode` is zero, which is the default, an existing file will be opened for update (mode `r+` for UNIX, `rb+` for Windows environments). If `filename` does not exist, it will be created (mode `w+` for UNIX, `wb+` for Windows environments). The destructor for this class closes the file.

After constructing an *RWFile*, you should use member function `isValid()` to check whether opening the file was successful.

There are member functions to flush the file, and to test whether the file has had an error, or is empty or at the end-of-file.

Example

Class *RWFile* also has member functions to determine the status of a file, and to read and write a wide variety of built-in types, either one at a time, or as arrays. The file pointer may be repositioned with functions `SeekTo()`, `SeekToBegin()`, and `SeekToEnd()`. The details of the *RWFile* class capabilities are summarized in the *Class Reference*.

The following example creates an *RWFile* with filename `test.dat`. The code reads an `int` (if the file is not empty), increments it, and writes it back to the file:

```
#include <rw/rwfile.h>

main(){
   RWFile file("test.dat");             // Construct the RWFile.

   // Check that the file exists, and that it has
   // read/write permission:
   if ( file.Exists() )
{
     int i = 0;
     // Read the int if the file is not empty:
     if ( !file.IsEmpty() ) file.Read(i);
     i++;
     file.SeekToBegin();
     file.Write(i);                          // Rewrite the int.
   }
   return 0;
}
```

Using Class RWFileManager 8

Class *RWFileManager* allocates, deallocates, and coalesces free space in a disk file. This is done internally by maintaining on disk a linked-list of free space blocks.

Two typedefs are used:

```
typedef long           RWoffset;
typedef unsigned long  RWspace;
```

The type `RWoffset` is used for the offset within the file to the start of a storage space; `RWspace` is the amount of storage space required. The actual typedef may vary depending on the system you are using.

Class *RWFile* is a public base class of class *RWFileManager*; therefore, the public member functions of class *RWFile* are available to class *RWFileManager.*

Construction

The *RWFileManager* constructor has the prototype:

```
RWFileManager(const char* filename);
```

The argument is the name of the file that the *RWFileManager* is to manage. If it exists, it must contain a valid *RWFileManager*; otherwise, one will be created.

8

Member Functions

The class *RWFileManager* adds four additional member functions to those of class *RWFile*. They are:

1. `RWoffset allocate(RWspace s);`
   Allocate `s` bytes of storage in the file, returning the offset to the start of the allocation.

2. `void deallocate(RWoffset t);`
   Deallocate (free) the storage space starting at offset `t`. This space must have been previously allocated by the function `allocate()`:

3. `RWOffset endData();`
   Return the offset to the last data in the file.

4. `RWoffset start();`
   Return the offset from the start of the file to the first space ever allocated by *RWFileManager*, or return `RWNIL`[1] if no space has been allocated, which implies that this is a new file.

The statement:

```
RWoffset a = F.allocate(sizeof(double));
```

uses `F` of *RWFileManager* to allocate the space required to store an object with the size of a double, and returns the offset to that space. To write the object to the disk file, you should seek to the allocated location and use `Write()`. It is an error to read or write to an unallocated location in the file.

It is your responsibility to maintain a record of the offsets necessary to read the stored object. To help you do this, the first allocation ever made by an *RWFileManager* is considered special and can be returned by member function `start()` at any time. The *RWFileManager* will not allow you to deallocate it. This first block will typically hold information necessary to read the remaining data, perhaps the offset of a root node, or the head of a linked-list.

The following example shows the use of class *RWFileManager* to construct a linked-list of `int`s on disk. The source code is included in the `toolexam` subdirectory as `fmgrsave.cpp` and `fmgrrtrv.cpp`.

1. `RWNIL` is a macro whose actual value is system dependent. Typically, it is `-1L`.

When using this example, you must type a carriage return after the last item you want to insert in order to guarantee that it will be added to the list. This is because different compilers handle the occurrence of an EOF on the `cin` stream differently.

```
#include <rw/filemgr.h>                                          // 1
#include <rw/rstream.h>

struct DiskNode {                                                // 2
  int       data;                                                // 3
  Rwoffset  nextNode;                                            // 4
};

main(){
   RWFileManager fm("linklist.dat");                             // 5

   // Allocate space for offset to start of the linked list:
   fm.allocate(sizeof(RWoffset));                                // 6
   // Allocate space for the first link:
   RWoffset thisNode = fm.allocate(sizeof(DiskNode));            // 7

   fm.SeekTo(fm.start());                                        // 8
   fm.Write(thisNode);                                           // 9

   DiskNode n;
   int temp;
   RWoffset lastNode;
   cout << "Input a series of integers, ";
   cout << "then EOF to end:\n";
   while (cin >> temp) {                                       // 10
     n.data = temp;
     n.nextNode = fm.allocate(sizeof(DiskNode));               // 11
     fm.SeekTo(thisNode);                                      // 12
     fm.Write(n.data);                                         // 13
     fm.Write(n.nextNode);
     lastNode = thisNode;                                      // 14
     thisNode = n.nextNode;
   }
```

```
   fm.deallocate(n.nextNode);                                    // 15
   n.nextNode = RWNIL;                                           // 16
   fm.SeekTo(lastNode);
   fm.Write(n.data);
   fm.Write(n.nextNode);
   return 0;
}                                                                // 17
```

Here's a line-by-line description of the program:

`//1` Include the declarations for the class *RWFileManager*.

`//2` `Struct` *DiskNode* is a link in the linked-list. It contains:

`//3` the data (an `int`), and:

`//4` the offset to the next link. `RWoffset` is typically `typedef`'d to a `long int`.

`//5` This is the constructor for an *RWFileManager*. It will create a new file, called `linklist.dat`.

`//6` Allocate space on the file to store the offset to the first link. This first allocation is considered special and will be saved by the *RWFileManager*. It can be retrieved at any time by using the member function `start()`.

`//7` Allocate space to store the first link. The member function `allocate()` returns the offset to this space. Since each *DiskNode* needs the offset to the next *DiskNode*, space for the next link must be allocated before the current link is written.

`//8` Seek to the position to write the offset to the first link. Note that the offset to this position is returned by the member function `start()`. Note also that `fm` has access to public member functions of class *RWFile*, since class *RWFileManager* is derived from class *RWFile*.

`//9` Write the offset to the first link.

`//10` A loop to read integers and store them in a linked-list.

`//11` Allocate space for the next link, storing the offset to it in the `nextNode` field of this link.

`//12` Seek to the proper offset to store this link

`//13` Write this link.

`//14` Since we allocate the next link before we write the current link, the final link in the list will have an offset to an allocated block that is not used. It must be handled as a special case.

`//15` First, deallocate the final unused block.

`//16` Next, reassign the offset of the final link to be `RWNIL`. When the list is read, this will indicate the end of the linked list. Finally, rewrite the final link with the correct information.

`//17` The destructor for class *RWFileManager*, which closes the file, will be called here.

Having created the linked-list on disk, how might you read it? Here is a program that reads the list and prints the stored integer field:

```
#include <rw/filemgr.h>
#include <rw/rstream.h>

struct DiskNode {
  int       data;
  Rwoffset  nextNode;
};

main(){
   RWFileManager fm("linklist.dat");                          // 1

   fm.SeekTo(fm.start());                                     // 2
   RWoffset next;
   fm.Read(next);                                             // 3

   DiskNode n;
   while (next != RWNIL) {                                    // 4
```

```
    fm.SeekTo(next);                                    // 5
    fm.Read(n.data);                                    // 6
    fm.Read(n.nextNode);
    cout << n.data << "\n";                             // 7
    next = n.nextNode;                                  // 8
  }
  return 0;
}                                                       // 9
```

And this is a line-by-line description of the program:

`//1` The *RWFileManager* has been constructed with an old File.

`//2` The member function `start()` returns the offset to the first space ever allocated in the file. In this case, that space will contain an offset to the start of the linked-list.

`//3` Read the offset to the first link.

`//4` A loop to read through the linked-list and print each entry.

`//5` Seek to the next link.

`//6` Read the next link.

`//7` Print the integer.

`//8` Get the offset to the next link.

`//9` The destructor for class *RWFileManager*, which closes the file, will be called here.

Using Class RWBTreeOnDisk 9

Class *RWBTreeOnDisk* has been designed to manage a B-tree in a disk file. The class represents an ordered collection of associations of keys and values, where the ordering is determined internally by comparing keys. Given a key, a value can be retrieved. Duplicate keys are not allowed.

Keys are arrays of `char`. The key length is set by the constructor. The ordering in the B-tree is determined by comparing keys with an external function, which you can change.

The type of the values is:

```
typedef long RWstoredValue;
```

The values typically represent an offset to a location in a file where an object is stored. Given a key, you can find where an object is stored and retrieve it. As far as class *RWBTreeOnDisk* is concerned, however, the value has no special meaning—it is up to you to interpret it.

The class *RWBTreeOnDisk* uses class *RWFileManager* to manage the allocation and deallocation of space for the nodes of the B-tree. You can use the same *RWFileManager* to manage the space for the objects themselves if the B-tree and data are to be in the same file. Alternatively, you could use a different *RWFileManager*, managing a different file, to store the B-tree and data in separate files.

The member functions associated with class *RWBTreeOnDisk* are similar to those of the in-memory class *RWBTreeDictionary*, except that keys are arrays of `char` rather than *RWCollectable*s. There are member functions to add a key-

value pair, remove a pair, replace a value associated with a key, query for information associated with a key, operate on all key-value pairs in order, return the number of entries in the tree, and determine if a key is contained in the tree.

Construction

An *RWBTreeOnDisk* is always constructed from an *RWFileManager.* If the *RWFileManager* is managing a new file, then the *RWBTreeOnDisk* will initialize it with an empty root node. For example, the following code fragment constructs an *RWFileManager* for a new file called `filename.dat` and then constructs an *RWBTreeOnDisk* from it:

```
#include <rw/disktree.h>
#include <rw/filemgr.h>

main(){
  RWFileManager fm("filename.dat");

  // Initializes filename.dat with an empty root:
  RWBTreeOnDisk bt(fm);
}
```

Example

In this example, key-value pairs of character strings and offsets to *RWDate*s representing birthdays are stored. Given a name, you can retrieve a birthdate from disk.

```
#include <rw/disktree.h>
#include <rw/filemgr.h>
#include <rw/cstring.h>
#include <rw/rwdate.h>
#include <rw/rstream.h>

main(){
   RWCString name;
   RWDate birthday;

   RWFileManager fm("birthday.dat");
   RWBTreeOnDisk btree(fm);                                    // 1

   while (cin >> name)                                         // 2
   {
     cin >> birthday;                                          // 3
    RWoffset loc =  fm.allocate(birthday.binaryStoreSize());// 4
     fm.SeekTo(loc);                                           // 5
     fm << birthday;                                           // 6
     btree.insertKeyAndValue(name, loc);                       // 7
   }
   return 0;
}
```

Here's the line-by-line description:

`//1` Construct a B-tree. The default constructor is used, resulting in a key length of 16 characters.

`//2` Read the name from standard input. This loop will exit when EOF is reached.

`//3` Read the corresponding birthday.

`//4` Allocate enough space from the *RWFileManager* to store the birthday. Function `binaryStoreSize()` is a member function in most Rogue Wave classes. It returns the number of bytes necessary to store an object in an *RWFile*. If you are storing an entire *RWCollection*, or using one of the methods `recursiveSaveOn()` or `operator<<(RWFile&, RWCollectable)`, be sure to use `recursiveStoreSize()` instead.

`//5` Seek to the location where the *RWDate* will be stored.

`//6` Store the date at that location. Most Rogue Wave classes have an overloaded version of the streaming operators `<<` and `>>`.

`//7` Insert the key and offset to the object in the B-tree.

Having stored the names and birthdates on a file, here's how you might retrieve them:

```
#include <rw/disktree.h>
#include <rw/filemgr.h>
#include <rw/cstring.h>
#include <rw/rwdate.h>
#include <rw/rstream.h>

main(){
   RWCString name;
   RWDate birthday;

   RWFileManager fm("birthday.dat");
   RWBTreeOnDisk btree(fm);

   while(1)
   {
     cout << "Give name: ";
     if (!( cin >> name)) break;                          // 1
     RWoffset loc = btree.findValue(name);                // 2
     if (loc==RWNIL)                                      // 3
       cerr << "Not found.\n";
     else
```

```
      {
        fm.SeekTo(loc);                                    // 4
        fm >> birthday;                                    // 5
        cout << "Birthday is " << birthday << endl;        // 6
      }
    }
    return 0;
}
```

Here is a description of the program:

`//1` The program accepts names until encountering an EOF.

`//2` The name is used as a key to *RWBTreeOnDisk*, which returns the associated value, an offset, into the file.

`//3` Check to see whether the name was found.

`//4` If the name is valid, use the value to seek to the spot where the associated birthdate is stored.

`//5` Read the birthdate from the file.

`//6` Print it out.

With a little effort, you can easily have more than one B-tree active in the same file. This allows you to maintain indexes on more than one key. Here's how you would create three B-trees in the same file:

```
#include <rw/disktree.h>
#include <rw/filemgr.h>

main(){
   RWoffset rootArray[3];

   RWFileManager fm("index.dat");
   RWoffset rootArrayOffset =    fm.allocate(sizeof(rootArray));

   for (int itree=0; itree<3; itree++)
   {
     RWBTreeOnDisk btree(fm, 10, RWBTreeOnDisk::create);
     rootArray[itree] = btree.baseLocation();
   }
   fm.SeekTo(fm.start());
   fm.Write(rootArray, 3);
   return 0;
}
```

And here is how you could open the three B-trees:

```
#include <rw/disktree.h>
#include <rw/filemgr.h>

main(){
   RWoffset rootArray[3];          // Location of the tree roots
   RWBTreeOnDisk* treeArray[3]; // Pointers to the RWBTreeOnDisks
    RWFileManager fm("index.dat");
    fm.SeekTo(fm.start());     // Recover locations of root nodes
    fm.Read(rootArray, 3);

    for (int itree=0; itree<3; itree++)
    {
      // Initialize the three trees:
      treeArray[itree] = new RWBTreeOnDisk(fm,
                         10,          // Max. nodes cached
        RWBTreeOnDisk::autoCreate,  // Will read old tree
                         16,          // Key length
                         FALSE,       // Do not ignore nulls
                         rootArray[itree] // Location of root
                                          );
    }

    .
    .
    .
    for (itree=0; itree<3; itree++)  // Free heap memory
       delete treeArray[itree];

    return 0;
}
```

9

Collection Classes 10

The *Tools.h++* class library includes three types of collection classes:

- The template-based collection classes, which we call collection class templates, or just templates, for short;
- The generic collection classes, modelkjKevin Johnsrudeed after Stroustrup (1986), Chapter 7.3.5;
- The Smalltalk-like collection classes.

Despite their different implementations, their functionality and user interfaces (member function names, etc.) are similar.

In the following sections, we'll discuss each type of collection class in turn. In this chapter, we'll discuss basic concepts of collection classes, and translate some of the jargon you may encounter here and in the literature of C++.

Storage Methods of Collection Classes

The general objective of collection classes, called *collections* for short, is to store and retrieve objects. In fact, you can classify collection classes according to how they store objects. *Value-based* collections store the object itself; *reference-based* collections store a pointer or reference to the object. The difference between the two will influence how you use some features of collection classes in *Tools.h++*.

Value-based collection classes are simpler to understand and manipulate. You create a linked list of integers or doubles, for example, or a hash table of shorts. Stored types can be more complicated, like the *RWCString*s, but the important

point is that they *act* just like values, even though they may contain pointers to other objects. When an object is inserted into a value-based collection class, a *copy* is made. The procedure is similar to C's pass-by-value semantics in function calls.

In a reference-based collection class, you store and retrieve *pointers* to other objects. For example, you could create a linked list of pointers to integers or doubles, or a hash table of pointers to *RWCString*s.

Let's look at two code fragments that demonstrate the difference between the value-based and the reference-based procedures:

Value-based Example	**Reference-based Example**

`/* A vector of RWCStrings: */`
`RWTValOrderedVector<RWCString> v;`
`RWCString s("A string");`
`v.insert(s);`
	`/* A vector of pointers to RWCStrings: */`
	`RWTPtrOrderedVector<RWCString> v;`
	`RWCString* p = new RWCString("A string");`
	`v.insert(p);`

Both code fragments insert an *RWCString* into vector `v`. In the first example, `s` is an *RWCString* object containing "A string". The statement `v.insert(s)` copies the value of `s` into the vector. The object that lies within the vector is distinct and separate from the original object `s`. In the second example, `p` is a pointer to the *RWCString* object. When the procedure `v.insert(p)` is complete, the new element in `v` will refer to the same *RWCString* object as `p`.

A Note on Memory Management

A reference-based collection can be very efficient because pointers are small and inexpensive to manipulate. However, with a reference-based collection, you must always remember that you are responsible for memory management: the creation, maintenance, and destruction of the actual objects themselves. If you create two pointers to the same object and prematurely delete the object, you'll leave the second pointer pointing into nonsense. By the same token, you must never insert a nil pointer into a reference-based collection, since the collection has methods which must dereference its contained values.

Despite the added responsibility, don't avoid reference-based collections when you need them. *Tools.h++* classes have member functions to help you, and in most cases, the ownership of the contained objects is obvious anyway. You should choose a reference-based collection if you need performance and size

advantages: here the size of all pointers is the same, allowing a large degree of code reuse. Also choose the reference-based collection if you just *want* to point to an object rather than contain it (a set of selected objects in a dialog list, for example). Finally, for certain heterogeneous collections, the reference-based approach may be the only one viable.

Copying Collection Classes

Copying classes is a common software procedure. It happens every time a copy constructor is applied, or whenever a process needs a copy to work on. Copying value-based collection classes is straightforward. But special considerations arise in copying reference-based classes, and we deal with them here.

Copying Reference-based Collection Classes

What happens when you make a copy of a reference-based collection class, or any class that references another object, for that matter? It depends which of the two general approaches you choose: *shallow copying* or *deep copying*.

1. A *shallow copy* of an object is a new object whose instance variables are identical to the old object. For example, a shallow copy of a *Set* has the same members as the old *Set*, and shares objects with the old *Set* through pointers. Shallow copies are sometimes said to use *reference semantics*.

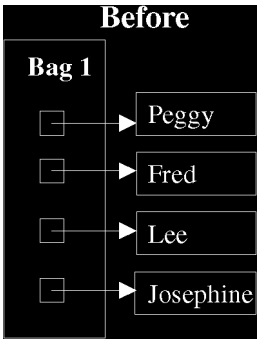
Note – The copy constructors of all reference-based Rogue Wave collection classes make shallow copies.

---

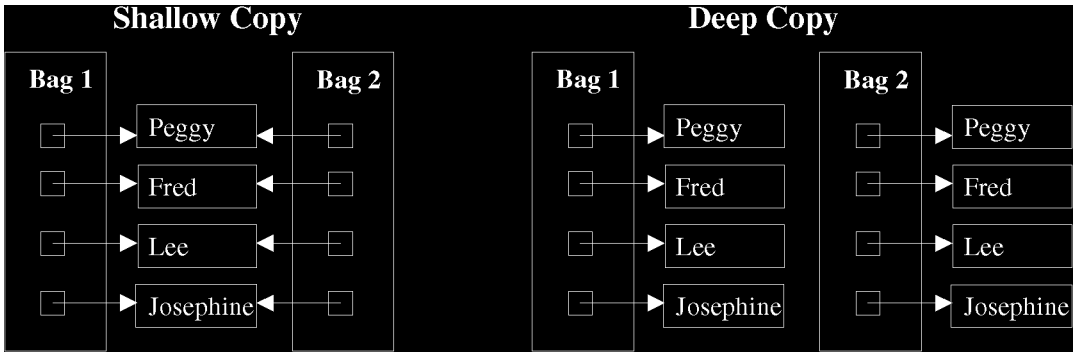
2. A *deep copy* of an object is a new object with entirely new instance variables; it does not share objects with the old. For example, a deep copy of a *Set* not only makes a new *Set*, but also inserts items that are copies of the old items. In a true deep copy, this copying is done recursively. Deep copies are sometimes said to use *value semantics*.

Note that some reference-based collection classes have a `copy()` member function that returns a new object with entirely new instance variables. This copying is not done recursively, and the new instance variables are shallow copies of the old instance variables.

Here's a graphical example of the differences between shallow and deep copies. Imagine *Bag*, an unordered collection class of objects with duplicates allowed, that looks like this before a copy :



Making a shallow copy and a deep copy of *Bag* would produce the following results:



You can see that the deep copy copies not only the bag itself, but recursively all objects within it.

The copying approach you choose is important. For example, shallow copies can be useful and fast, because less copying is done, but you must be careful because *two* collections now reference the same object. If you delete all the items in one collection, you will leave the other collection pointing into nonsense.

You also need to consider the approach when writing an object to disk. If an object includes two or more pointers or references to the same object, it is important to preserve this *morphology* when the object is restored. Classes that inherit from *RWCollectable* inherit algorithms that guarantee to preserve an object's morphology. You'll see more on this in Chapter 14.

Copying Value-based Collection Classes

Let us now contrast the results of copying the reference-based collection with the value-based collection. Consider the class:

RWTValOrderedVector<RWCString>

that is, an ordered vector template instantiated for *RWCString*. In this case, each string is embedded within the collection. When a copy of the collection class is made, not only the collection class itself is copied, but also the objects in it. This results in distinct new copies of the collected objects:

Retrieving Objects in Collections

We have defined the major objective of collection classes as storing and retrieving objects. How you retrieve or find an object depends on its properties. Every object you create has three properties associated with it:

1. *Type*: for example, an *RWCString* or a `double`. In C++, the type of an object is set at creation, and cannot change.

2. *State*: the value of the string. The values of all the instance variables or attributes of an object determine its state. These can change.

3. *Identity*: the unique definition of the object for all time. Languages use different methods for establishing an object's identity. C++ always uses the object's address. Each object is associated with one and only one address. Note that the reverse is not always true, because of inheritance. Generally, an address and a type[1] are both necessary to disambiguate the object you mean within an inheritance hierarchy.

Retrieval Methods

Based on the properties of an object, there are two general methods for finding or retrieving it. Some collection classes can support either, some only one. The important thing for you to keep in mind is which one you mean. The two methods are:

1. Find an object with a particular state. For example, test two strings for the same value. In the literature, this is variously referred to as two objects testing `isEqual`, having equality, `compares equal`, having the same value, or testing true for the `==` operator. Here, we refer to the two objects testing equal as `isEqual`. In general, we need some knowledge of the type of each objectæor subtype, in the case of inheritanceæin order to find the appropriate instance variables to test for equality[2].

2. Find a particular object; that is, one with the same identity as the object being compared. In the literature, this is referred to as two objects testing `isSame`, having the same identity, or testing true for the `==` operator. We refer to this as two objects having the same identity. Note that because value-based collection classes make a copy of an inserted object, finding an object in a value-based collection class with a particular identity is meaningless.

In C++, to test for identity—that is, to test whether two objects are the same object—you must see if they have the same address. Because of multiple inheritance, the address of a base class and its associated derived class may not be the same. Therefore, if you compare two pointers (addresses) to test for identity, the types of the two pointers should be the same.

1. Because of multiple inheritance, it may be necessary to know not only an object's type, but also its location within an inheritance tree in order to disambiguate which object you mean.

2. The Rogue Wave collection classes allow a generalized test of equality; it is up to you to define what it means for two objects to "be equal". A bit-by-bit comparison of the two objects is not done. You could define "equality" to mean that a panda is the same as a deer because, in your context, they are both mammals.

Smalltalk uses the operator = to test for equality, and the operator == to test for identity. In the C++ world, however, operator = is firmly attached to assignment, and operator == to some kind of equality of values. We have taken the C++ approach. At Rogue Wave, the operator == generally means test for equality of values (`isEqual`) when applied to two classes, and test for identity when applied to two pointers.

Whether to test for equality or identity depends on the context of your problem. Here are some examples that can clarify which to choose.

Here's an example when you should test for equality. Suppose you are maintaining a mailing list. Given a person's name, you want to find his or her address. In this case, you search for a name that is equal to the name at hand. An *RWHashDictionary* would be appropriate. The key to the dictionary would be the name, the value would be the address.

In the next example, you would test for identity. Suppose you are writing a hypertext application, and need to know in which document a particular graphic occurs. You could keep an *RWHashDictionary* of graphics and their corresponding documents. In this case, however, you need an *RWIdentityDictionary* because you need to know in which document a *particular* graphic occurs. The graphic acts as the key, the document as the value.

Maintaining a disk cache? You might want to know whether a particular object is resident in memory. In this case, an *RWIdentitySet* is appropriate. Given an object, you can check to see whether it exists in memory—another identity test.

Iterators in Collection Classes

Many of the collection classes have an associated iterator. The advantage of the iterator is that it maintains its own internal state, thus allowing two important benefits:

- More than one iterator can be constructed from the same collection class;
- All of the items need not be visited in a single sweep.

Iterators are always constructed from the collection class itself, as in the following example:

```
RWBinaryTree bt;
.
.
```

```
.
RWBinaryTreeIterator bti(bt);
```

Immediately after construction, or after `reset()`is called, the state of the iterator is undefined. You must either advance it or position it before using its current state or position.

For traditional *Tools.h++* iterators—those declared as a distinct class related to the collection class—the rule is "advance and then return."[1] However, iterators obtained directly from classes implemented using the Standard C++ Library differ. In keeping with the standard for container classes, they follow the precept: If you obtain an iterator using the `begin()` or `end()` method, or using an algorithm which returns an iterator, you have a "Standard Library" iterator.[2] A Standard Library iterator must always be compared against that collection's `end()` iterator to discover if it references an item in the container.

Traditional Tools.h++ Iterators

Traditional *Tools.h++* iterators have a number of unique features.

You recall that the state of the iterator is undefined immediately following construction or the calling of `reset()`. You also trigger the undefined state if you change the collection class directly[3] by adding or deleting objects while an iterator is active. Using an iterator at that point can bring unpredictable results. You must then use the member function `reset()` to restart the iterator, as if it had just been constructed.

At any given moment, the iterator marks an object in the collection class—think of it as the current object. There are various methods for moving this mark. For example, most of the time you will probably be using member function `operator()`. In *Tools.h++*, it is designed to *always* advance to the next object, then return either `TRUE` or a pointer to the next object, depending on whether the associated collection class is value-based or reference-based, respectively. It always returns `FALSE` (i.e., zero) when the end of the collection class is reached. Hence, a simple canonical form for using an iterator is:

```
RWSlistCollectable list;
.
```

1. This is actually patterned after Stroustrup (1986, Section 7.3.2).
2. The draft ANSI standard describes container iterators in great detail. Briefly, such iterators are valid in the range "first-element" to "one-past-last-element", which are returned respectively by the methods `begin()` and `end()`. The "end" iterator, however, does not reference an item in the container, but acts as a sentinel.
3. It's OK to change a collection via the iterator itself.

```
.
.
RWSlistCollectableIterator iterator(list);
RWCollectable* next;
while (next = iterator()) {
  .
  .                                                    // (use next)
  .
}
```

As an alternative, you can also use the prefix increment operator `++X`. Some iterators have other member functions for manipulating the mark, such as `findNext()` or `removeNext()`.

Member function `key()` always returns either the current object or a pointer to the current object, again depending on whether the collection class is value-based or reference-based, respectively.

For most collection classes, using member function `apply()` to access every member is much faster than using an iterator. This is particularly true for the sorted collection classes—usually a tree has to be traversed here, requiring that the parent of a node be stored on a stack. Function `apply()` uses the program's stack, while the sorted collection class iterator must maintain its own. The former is much faster.

10

Collection Class Templates 11

Introduction

As a developer, you no doubt periodically ask yourself, “Haven’t I coded this before?” Clearly, one of the primary attractions of the C++ language is the promise of *reuse*, the lure of avoiding rewrites of the same old code, over and over again. The *Tools.h++* collection classes take advantage of several C++ language features that support reuse.

The Smalltalk-like collection classes, discussed later in detail, effect object-code reuse through polymorphism and inheritance. In this chapter, we demonstrate reuse in the *Tools.h++* collection class templates, called templates in our jargon.

A template is a class declaration parameterized on a type: a prescription for how a particular type of collection class should behave. For example, a `Vector` template would describe such things as how to index an element, how long it is, how to resize it, and so on. The actual type of the elements is independent of these larger, more general issues.

With templates, you achieve extreme *source*-code reuse by writing code for your collections without regard to the particular type or types of elements being collected. The template mechanism allows you to represent these types using formal template parameters, which act as place holders. Later, when you want to make use of your collection class template, you *instantiate* the template with an actual type. At that point, the compiler goes back to the class template and fills it in with that type, as if you had written it that way in the first place.

11

Without templates, you would have to write class `VectorOfInt`, `VectorOfDouble`, `VectorOfFoo`, and so on; with templates, you simply code one class, `Vector<T>` , where `T` can stand for any type. From there, you're free to create `Vector<int>`, `Vector<double>`, `Vector<Foo>`, or a vector of some type never conceived of when the class template was written originally.

Template Overview

To gain some perspective, let's begin with a general example that shows how templates work. We'll explain concepts from the example throughout the section, though you'll probably follow this without difficulty now:

```
#include <iostream.h>
#include <rw/cstring.h>
#include <rw/regexp.h>
#include <rw/tvdlist.h>

int main()
{
  // Declare a linked-list of strings:
  RWTValDlist<RWCString> stringList;
  RWTValDlist<RWCString>::iterator iter;
  RWCString sentence;

  // Add words to the list:
  stringList.insert("Templates");
  stringList.insert("are");
  stringList.insert("fun");

  // Now use standard iterators to build a sentence:
  iter = stringList.begin();
  while (iter != stringList.end()) {
    sentence += (*iter++ + " ");
  }
  // Replace trailing blank with some oomph!
  sentence(RWCRegexp(" $")) = "!"

  // Display the result:
  cout << sentence << endl;
  return 0;
}
```

Output:

```
Templates are fun!
```

The preceding example demonstrates the basic operation of templates. Using the collection class template `RWTValDList`, we instantiate the object `stringList`, simply by specifying type `RWCString`. The template gives us complete flexibility in specifying the type of the list; we don't write code for the object, and *Tools.h++* doesn't complicate its design with a separate class *RWTValDListofRWCString*. Without the template, we would be limited to types provided by the program, or forced to write the code ourselves.

Template Naming Convention

You'll notice that the collection class template *RWTValDlist* in the example follows a unique format. In *Tools.h++*, all templates have class names starting with *RWT* , for *R*ogue *W*ave *T*emplate, followed by a three letter code:

Isv	Intrusive lists
Val	Value-based
Ptr	Pointer-based

Hence, *RWTValOrderedVector<T>* is a value-based template for an ordered vector of type-name `T`. *RWTPtrMultiMap<Key,T,C>* is a pointer-based template based on the Standard C++ Library `multimap` class. Special characteristics may also modify the name, as in *RWTValSortedDlist<T,C>*, a value-based doubly-linked template list that automatically maintains its elements in sorted order.

Value vs. Reference Semantics in Templates

Tools.h++ collection class templates can be either value-based or pointer-based. Value-based collections use *value semantics,* maintaining copies of inserted objects and returning copies of retrieved objects. In contrast, pointer-based collections use *reference semantics,* dealing with pointers to objects as opposed to the objects themselves. See "Storage Methods of Collection Classes" on page 77 for other examples of value and reference semantics.

Templates offer you a choice between value and reference semantics. In fact, in most cases, you must choose between a value-based or a pointer-based class; for example, either *RWOrderedVectorVal*, or *RWOrderedVectorPtr*.

Your choice depends on the requirements of your application. Pointer-based templates are a good choice for maximizing efficiency for large objects, or if you need to have the same group of objects referred to in several ways, requiring that each collection class *point* to the target objects, rather than wholly contain them.

An Important Distinction

There is a big difference between a value-based collection of pointers, and a pointer-based collection class. You can save yourself difficulty by understanding the distinction. For example, declaring:

```
// value-based list of RWDate pointers:
RWTValDlist<RWDate*> myBirthdayV;
```

gives you a value-based list, where the values are of type pointer to *RWDate*. The collection class will concern itself only with the pointers, never worrying about the actual *RWDate* objects they refer to. Now consider:

```
RWDate* d1 = new RWDate(29,12,55);  // December 29, 1955
myBirthdayV.insert(d1);
RWDate* d2 = new RWDate(29,12,55);  // Different object, same date
cout << myBirthdayV.occurrencesOf(d2);    // Prints 0
```

The above code prints `0` because the memory locations of the two date objects are different, and the collection class is comparing only the values of the pointers themselves (their addresses) when determining the number of occurrences.

Contrast that with the following:

```
RWTPtrDlist<RWDate> myBirthdayP;   // pointer-based list of RWDates
RWDate* d1 = new RWDate(29,12,55); // December 29,1955
myBirthdayP.insert(d1);
RWDate* d2 = new RWDate(29,12,55); // Different object, same date
cout << myBirthdayP.occurrencesOf(d2);    // Prints 1
```

Here the collection class is parameterized by *RWDate*, not *RWDate**, showing that only *RWDate* objects, not pointers, are of interest to the list. But because it is a pointer-based collection class, communicating objects of interest is done via

pointers to those objects. The collection class knows it must dereference these pointers, as well as those stored in the collection class, before comparing for equality.

Intrusive Lists in Templates

For a collection class of type-name `T`, intrusive lists are lists where type `T` inherits directly from the link type itself[1]. The results are optimal in space and time, but require you to honor the inheritance hierarchy. The disadvantage is that the inheritance hierarchy is inflexible, making it slightly more difficult to use with an existing class. For each intrusive list class, *Tools.h++* offers templatized value lists as alternative non-intrusive linked lists.

Note that when you insert an item into an intrusive list, the actual item , not a copy, is inserted. Because each item carries only one link field, the same item cannot be inserted into more than one list, nor can it be inserted into the same list more than once.

Tools.h++ Templates and the Standard C++ Library

Most of the *Tools.h++* collection class templates use the Standard C++ Library for their underlying implementation. The collection classes of the Standard C++ Library, called *containers*, act as an engine under the hood of the *Tools.h++* templates.

For example, the value-based *Tools.h++* double-ended queue *RWTValDeque<T>* *has-a* member of type `deque<T>` from the Standard C++ Library. This member serves as the implementation of the collection. Like an engine, it does the bulk of the work of adding and removing elements, and so on. *RWTValDeque<T>* is a *wrapper* class, like the hood protecting the engine. More than cosmetic, it functions as a simpler, object-oriented interface to the *deque* class, making it easier and more pleasant to deal with.

Thanks to inlining and the lack of any extra level of indirection, this wrapping incurs few, if any, performance penalties. If you need direct access to the implementation, the wrapper classes offer the member function `std()`, which returns a reference to the implementation. In addition, because the Rogue

1. See Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, 1991, for a description of intrusive lists.

Wave template collections supply standard iterators, you can use them with the Standard C++ Library algorithms as if they were Standard C++ Library collections themselves.

Standard C++ Library Not Required

A unique feature of this version of *Tools.h++* is that many of its template collections do not actually *require* the presence of the Standard C++ Library. Consider *RWTValDlist<T>*. If you are using *Tools.h++* on a platform that supports the Standard C++ Library, *RWTValDlist<T>* will be built around the Standard C++ Library `list` container. But if your platform does not support the Standard C++ Library, you may still use the class. *Tools.h++* accomplishes this feat transparently through an alternate implementation, not based on the Standard C++ Library. The appropriate implementation is selected at compile time based on the settings in the configuration header file, `rw/compiler.h`. In fact, the alternate implementations are exactly those that were employed in the previous version of *Tools.h++*.

When using one of these template collections without the Standard C++ Library, you will be restricted to a subset of the full interface. For example, the `std()` member function mentioned above is not available, nor are the `begin()` and `end()` functions, which return Standard C++ Library iterators. The *Tools.h++ Class Reference* contains entries for both the full and the subset interfaces for all of the templates that can be used either with or without the Standard C++ Library.

There are two reasons you may want to use the restricted subset interface for a collection class template:

1. You may be operating in an environment that does not yet support a version of the Standard C++ Library compatible with this version of *Tools.h++*. In that case, you have no choice but to use the restricted subset interface. The good news is that by using the interface, you will be ready to start using the full interface as soon as the Standard C++ Library becomes available on your platform.

2. Another reason to stick to the subset interface is that you want to write portable code—a class library, perhaps—that can be deployed on multiple platforms, some without support for the Standard C++ Library. Clients of that code can still take full advantage of their individual environments; you

aren't forced to inflict on them a "lowest common denominator." See "Using Templates Without the Standard Library" on page 106 toward the end of this chapter for more information on the restricted subset interface.

The Standard C++ Library Containers

There are seven Standard C++ Library containers: `deque`, `list`, `vector`, `set`, `multiset`, `map`, and `multimap`. *Tools.h++* extends these with five additional containers which are compliant with the Standard C++ Library: *rw_slist*, *rw_hashset*, *rw_hashmultiset*, *rw_hashmap*, and *rw_hashmultimap*. Each of these has value-based and pointer-based Rogue Wave wrapper templates. *Tools.h++* also offers always-sorted versions, both value-based and pointer-based, of the doubly-linked list and vector collections. The total: 28 new or re-engineered collection class templates, all based on the Standard C++ Library!

Tools.h++ Standard Library-Based Templates

Division / / Notes	Value-based	Pointer-based	Standard Library Required?
Sequence-based	*RWTValDlist*	*RWTPtrDlist*	No
	RWTValDeque	*RWTPtrDeque*	Yes
	RWTValOrderedVector	*RWTPtrOrderedVector*	No
/ /External ordering, access by index	*RWTValSlist*	*RWTPtrSlist*	No
Sorted sequence-based	*RWTValSortedDlist*	*RWTPtrSortedDlist*	Yes
/ /Internal ordering, access by index	*RWTValSortedVector*	*RWTPtrSortedVector*	No
Associative container-based	*RWTValSet*	*RWTPtrSet*	Yes
(set-based)	*RWTValMutliSet*	*RWTPtrMultiSet*	Yes
(map-based)	*RWTValMap*	*RWTPtrMap*	Yes
/ /Internal ordering, access by key	*RWTValMultiMap*	*RWTPtrMultiMap*	Yes
Associative hash -based	*RWTValHashSet*	*RWTPtrHashSet*	No

(set-based)	*RWTValHashMutliSet*	*RWTPtrHashMultiSet*	No
(map-based)	*RWTValHashMap*	*RWTPtrHashMap*	No
/ /No ordering, access by key	*RWTValHashMultiMap*	*RWTPtrHashMultiMap*	Yes

Commonality of Interface

To keep things simple and allow you to program with more flexibility, we have implemented common interfaces within the various divisions of standard-library based collection class templates. For example, the *RWTPtrSet* and *RWTPtrMultiSet* templates have interfaces identical to their value-based cousins; so do the map-based collection classes. All of the Sequence-based collections have nearly identical interfaces within the value and pointer-based subgroups. (An exception here is the set of *deque*-based classes, which contain `push` and `pop` member functions designed to enhance their abstraction of the queue data structure.)

There are pluses and minuses to this approach. The downside is that it puts slightly more of the burden on you, the developer, to choose the appropriate collection class. Had we chosen not to provide the `insertAt(size_type index)` member function for class *RWOrderedVectorVal<Type>*, we could have enforced the idea that vector-based templates are not a good choice for inserting into the middle of a collection class. Instead, it is up to you to be aware of your choices and use such member functions judiciously.

On the plus side, the common interface lowers the learning curve, allows flexibility in experimenting with different collections, and provides the capability of dealing with the Rogue Wave templates polymorphically via genericity.[1]

Real-life programming is seldom as neat as the exercises in a data structures textbook. You may find yourself in a situation where it is difficult to balance the trade-offs and determine just which collection class to use. With the common interface, you can easily benchmark code that uses an *RWTValDeque* and later benchmark it again, substituting an *RWTValOrderedVector* or *RWTValDlist*. You can also write class and function templates that are parameterized on the collection class type. For example:

1. For a discussion of genericity versus inheritance, see Meyer (1988).

```
template <class RWSeqBased>
void tossHiLo(RWSeqBased& coll) {
    // throw out the high and low values:
    assert(coll.entries() >= 2);  // precondition
    coll.sort();
    coll.removeFirst();
    coll.removeLast();
}
```

Thanks to the common interface, the above function template will work when instantiated with any of the Rogue Wave Sequence-based templates.

Parameter Requirements

In order to use a *Tools.h++* template collection class to collect objects of some type `T`, that type must satisfy certain minimal requirements. Unfortunately, some compilers may require the instantiating type or types to be more powerful than should be necessary.

According to the draft C++ standard, a compiler should only instantiate and compile those template functions that are actually used. Thus, if you avoid calling any member functions, such as `sort()`, that require valid `less-than` semantics, you should still be able to create a collection class of some type `U` for which, given instances `u1` and `u2`, the expression `(u1 < u2)` is ill-formed. If your compiler does not provide this selective instantiation, you may not be able to collect objects of type `U` without implementing `operator<(const U&, const U&)` for that type.

Comparators

The associative container-based and the sorted sequence-based collection classes maintain order internally. This ordering is based on a comparison object, an instance of a comparator class you must supply when instantiating the template. A comparator must contain a `const` member `operator()`, the function-call operator, which takes two potential elements of the collection class as arguments and returns a Boolean value. The returned value should be `true` if the first argument must precede the second within the collection class, and `false` otherwise. Often, it is easiest to use one of the function-object

classes provided by the Standard C++ Library in the header file <functional>. In particular, use less<T> to maintain elements in increasing order, or greater<T> to maintain them in decreasing order. For example:

```
#include <functional>
#include <rw/tvset.h>
#include <rw/rwdate.h>

RWTValSet<int, less<int> > mySet1;
RWTValSet<RWDate, greater<RWDate> > mySet2;
```

Here mySet1 is a set of integers kept in increasing order, while mySet2 is a set of dates held in decreasing order; that is, from the most recent to the oldest. You can use these comparators from the Standard C++ Library as long as the expression (x < y) for the case of less<T>, or (x > y) for the case of greater<T>, are valid expressions that induce a total ordering on objects of type T.

More on Total Ordering

As noted above, the comparator must induce a total ordering on the type of the items in the collection class. This means that the function-call operator of the comparator must satisfy the following two conditions[1], assuming that comp is the comparison object and x, y, and z are potential elements of the collection class, not necessarily distinct:

```
I.      Exactly one of the following statements is true:
        a) comp(x,y) is true  and comp(y,x) is false
        b) comp(x,y) is false and comp(y,x) is true
        c) comp(x,y) is false and comp(y,x) is false
        (or, in other words: not both comp(x,y) and comp(y,x)
         are true)
II.     If comp(x,y) and comp(y,z) are true, then so is comp(x,z)
(transitivity).
```

The truth of I.a implies that x must precede y within the collection class, while I.b says that y must precede x. More interesting is I.c. If this statement is true, we say that x and y are *equivalent*, and it doesn't matter in what order they occur within the collection class. This is the notion of equality that prevails for the templates that take a comparator as a parameter. For example, when the member function contains(T item) of an associative container-based template tests to see if the collection class contains an element equivalent to item, it is really looking for an element x in the collection class

1. Adapted from Knuth (1973).

such that `comp(x,item)` and `comp(item,x)` are both `false`. It is important to realize that the `==` operator is *not* used. Don't worry if at first it seems counter-intuitive that so much negativity can give rise to equivalence—you are not alone! You'll soon be comfortable with this flexible way of ensuring that everything has its proper place within the collection class.

Comparators are generally quite simple in their implementation. Take for example:

```
class IncreasingLength {
public:
    bool operator()(const RWCString& x, const RWCString& y)
    { return x.length() < y.length(); }
};
RWTValSet<RWCString,IncreasingLength> mySet;
```

Here `mySet` maintains a collection of strings, ordered from shortest to longest by the length of those strings. You can verify that an instance of the comparator satisfies the given requirements for total ordering. In the next example, `mySet2` maintains a collection class of integers in decreasing order:

```
class DecreasingInt {
public:
    bool operator()(int x, int y)
    { return x > y; }
};
RWTValSet<int, DecreasingInt> mySet2;
```

Although the sense of the comparison may seem backwards when you first look at it, the comparator says that `x` should precede `y` within the collection class if `x` is greater than `y`; hence, you have a decreasing sequence. Finally, let's look at a bad comparator:

```
// DON'T DO THIS:
class BadCompare {
public:
    bool operator()(int x, int y)
    { return x <= y; }    // OH-OH! Not a total ordering relation
};
RWSetVal<int, BadCompare> mySet3;  // ILLEGAL COMPARATOR!
```

To determine why it's bad, consider an instance `badcomp` of `BadCompare`. Note that when using the value `7` for both `x` and `y`, none of the three statements `I.a`, `I.b`, or `I.c` is `true`, which violates the first rule of a total ordering relation.[1]

11

Hash Functors and Equalitors

The associative hash-based templates use a hash function object to determine how to place and locate objects within the collection class. An advantage of using hash function objects is efficient, constant-time retrieval of objects. A disadvantage is that objects are maintained in an order determined by mapping the result of the hash function onto the physical layout of the collection class itself. Rarely does this ordering have a useful meaning beyond the internal machinations of the hash-based container.

To avoid complete chaos, associative hash-based containers make use of an equality object. Collections which allow multiple keys that are equivalent to one another use the equality object to ensure that equivalent objects are grouped together when iteration through the container occurs. Hash collections which do not allow multiple keys use the equality object to ensure that only unique items are admitted. To effect these procedures, we need two template arguments in addition to the type or types of the elements being collected: a hash functor, and an equalitor.

A *hash functor* is a class or `struct` that contains a function-call operator that takes a const reference to a potential element of the collection class, and returns an unsigned long hash value. An *equalitor* is a class or `struct` that contains a function-call operator that takes two potential elements of the collection class, and returns `true` if the elements should be considered equivalent for the purpose of grouping objects or ensuring uniqueness within the collection class. For example:

```
#include <rw/tvhset.h>   // contains RWTValHashSet
#include <rw/cstring.h>  // Contains RWCString

struct StringHash {
   unsigned long operator()(const RWCString& s)
      {  return s.hash(); }
};

struct StringEqual {
   unsigned long operator()(const RWCString& s, const RWCString& t)
      {  return s == t; }
};

RWTValHashSet<RWCString, StringHash, StringEqual> rwhset;
```

1. Unfortunately, the requirement for total ordering is a logical, not a semantic one, so the compiler cannot help by rejecting poorly chosen comparitors. In general, such code will compile, but probably have unexpected behavior.

Here we instantiate an *RWHashValSet* of *RWCString*s with our own hash functor and equalitor. The example shows the relative ease of creating these `struct`s for use as template arguments.

Iterators

Tools.h++ provides several distinct methods for iterating over a collection class. Most collections offer an `apply` member function, which applies your supplied function to every element of a collection class before returning. Another form of iteration is provided by separate collaborating iterator classes associated with many of the collections. For example, an *RWTPtrDlistIterator*<*T*> can be used to visit each element of an *RWTPtrDlist*<*T*> in turn. Iterators are described in "Iterators in Collection Classes" on page 83 of *Collection Classes*.

Standard C++ Library Iterators

All *Tools.h++* standard library-based collection class templates provide standard iterators. These iteratorsi are fully compliant with the Standard C++ Library requirements for iterators, making them a powerful tool for using the classes in conjunction with the Standard C++ Library—especially the algorithms. Although full treatment of iterators is beyond the scope of this guide, your Standard C++ Library reference and tutorials will provide ample information.

The standard library-based collection class templates provide three types of iterators: forward, bi-directional, and random-access. *Forward iterators* allow unidirectional traversal from beginning to end. As suggested by the name, *bidirectional iterators* allow traversal in both directions—front to back, and back to front. *Random-access* iterators are bidirectional as well, and further distinguished by their ability to advance over an arbitrary number of elements in constant time. All of these iterators allow access to the item at the current position via the dereference operator `*`.

Given iterator `iter` and an integral value `n`, the following basic operations are just some of those supported:

Expression	Meaning	Supported by:
`++iter;`	advance to next item and return	Forw, Bidir, Random
`iter++;`	advance to next item, return original value	Forw, Bidir, Random
`*iter;`	return reference to item at current position	Forw, Bidir, Random
`--iter;`	retreat to previous item and return	Bidir, Random
`iter--;`	retreat to previous item, return original value	Bidir, Random
`iter+=n;`	advance `n` items and return	Random
`iter-=n;`	retreat `n` items and return	Random

Again, your standard library documentation will describe all the operators and functions available for each type of iterator.

In addition to the iterators just described, the standard library-based collection class templates also provide two typedefs used to iterate over the items in a collection class: `iterator`, and `const_iterator`. You can use the `iterator` typedef to traverse a collection class and modify the elements within. You can use instances of `const_iterator` to traverse, but *not* modify, the collection class and access elements. For the associative container-based and sorted sequence-based collections, which do not allow modification of elements once they are in the collection class, the `iterator` and `const_iterator` types are the same.

Finally, the templates also provide two member functions that return actual iterators you can use for traversing their respective collection classes. These member functions are `begin()` and `end()`. Each of these member functions is overloaded by a const receiver so that the non-const version returns an instance of type `iterator`, and the const version returns an instance of type `const_iterator`.

Member function `begin()` always returns an iterator already positioned at the first item in the collection class. Member function `end()` returns an iterator which has a *past-the-end* value, the way a pointer to the NULL character of a null-terminated character string has a value that points "past the end." An iterator of past-the-end value can be used to compare with another iterator to see if you've finished visiting all the elements in the collection class. It can also be used as a starting point for moving backwards through collection classes that provide either bidirectional or random-access iterators. The one thing you *cannot* do with an `end()` iterator is dereference it. The one thing you cannot do with an `end()` iterator is deference it. Here's an example using iterators to move through a list and search for a match:

```
RWTValDlist<int> intCollection; // a list of integers

// ... < put stuff in the list >

// position iter at start:
RWTValDlist<int>::iterator iter = intCollection.begin();

// set another iterator past the end:
RWTValDlist<int>::iterator theEnd = intCollection.end();

// iterate through, looking for a 7:
while (iter != theEnd) {       // test for end of collection
  if (*iter == 7)            // use '*' to access current element
    return true;               // found a 7
  ++iter;                      // not a 7, try next element
}
return false;                  // never found a 7
```

Map-Based Iteration and Pairs

In the case of a map-based collection class, like *RWMapVal<K,T,Compare>*, iterators refer to instances of the Standard C++ Library structure `pair<const K, T>`. As you iterate over a map-based collection, you have access to both the key and its associated data at each step along the traversal. The `pair` structure provides members `first` and `second`, which allow you to individually access the key and its data, respectively. For example:

```
typedef RWTValMap<RWCString, RWDate, less<RWCString> > Datebook;

Datebook birthdays;
// ... < populate birthdays collection >

Datebook::iterator iter = birthdays.begin();

while (iter != birthdays.end()) {
  cout << (*iter).first              // the key
       << " was born on "
       << (*iter).second             // the data for that key
       << endl;

  ++iter;
};
```

Note that given a non-const reference to such a `pair`, you can still modify only the second element—the data associated with the key. This is because the first element is declared to be of type `const K`. Because the placement of objects within the collection class is maintained internally, based on the value of the key, declaring it as `const` protects the integrity of the collection class. In order to change a key within a map, you will have to remove the key and its data entirely, and replace them with a new entry.

Iterators as Generalized Pointers

It may not be obvious at first, but you can think of an iterator as a generalized pointer. Imagine a pointer to an array of `int`s. The array itself is a collection class, and a pointer to an element of that array is a random-access iterator. To advance to the next element, you simply use the unary operator `++`. To move back to a previous element, you use `--`. To access the element at the current position of the iterator, you use the unary operator `*`. Finally, it is important to

know when you have visited all the elements. C++ guarantees that you can always point to the first address past the end of an allocated array. For example:

```
int intCollection[10];         // an array of integers

// ... < put stuff in the array >

// position iter at start:
int* iter = intCollection;

// set another iterator past the end:
int* theEnd = intCollection + 10;

// iterate through, looking for a 7:
while (iter != theEnd) {       // test for end of array
  if (*iter == 7)            // use '*' to access current element
    return true;               // found a 7
  ++iter;                      // not a 7, try next element
}
return false;                  // never found a 7
```

If you compare this code fragment to the one using standard iterators in "Standard C++ Library Iterators" on page 99, you can see the similarities. If you need a bit of help imagining how the standard iterators work, you can always picture them as generalized pointers.

Iterators and the std() Gateway

The *Tools.h++* templates are meant to enhance the Standard C++ Library, not to stand as a barrier to it. The iterators described in the previous section are standard iterators, and you can use them in conjunction with any components offering a standard iterator-based interface. In particular, you can use all of the standard algorithms with the Rogue Wave standard library-based collections. For example:

```
RWTValOrderedVector<int> vec;

// ... < put stuff in vector >

// Set the first 5 elements to 0:
fill(vec.begin(), vec.begin() + 5, 0);
```

In addition, you are always free to access, and in some cases to manipulate, the underlying Standard C++ Library collection class. This is accomplished via the `std()` member function, which returns a reference to the implementation.

The Best of Both Worlds

The following example is a complete program that creates a deck of cards and shuffles it. The purpose of the example is to show how the *Tools.h++* template collections can be used in conjunction with the Standard C++ Library. See your Standard C++ Library documentation for more information on the features used in the example.

```
/* Note: This example requires the C++ Standard Library */

#include <iostream.h>
#include <algorithm>
#include <rw/tvordvec.h>

struct Card {
  char  rank;
  char  suit;

  bool operator==(const Card& c) const
    { return rank == c.rank && suit == c.suit; }

  Card() { }
  Card(char r, char s) : rank(r), suit(s) { }

 // print card: e.g. '3-C' = three of clubs, 'A-S' = ace of spades
  friend ostream& operator<<(ostream& ostr, const Card& c)
  { return (ostr << c.rank << "-" << c.suit << "  "); }
};
```

```
/*
 * A generator class - return Cards in sequence
 */
class DeckGen {
  int rankIdx;  // indexes into static arrays below
  int suitIdx;
  static const char Ranks[13];
  static const char Suits[4];
public:
  DeckGen() : rankIdx(-1), suitIdx(-1) { }

  // generate the next Card
  Card operator()() {
    rankIdx = (rankIdx + 1) % 13;
    if (rankIdx == 0)
    // cycled through ranks, move on to next suit:
      suitIdx = (suitIdx + 1) % 4;
    return Card(Ranks[rankIdx], Suits[suitIdx]);
  }
};

const char DeckGen::Suits[4]  = {'S', 'H', 'D', 'C' };
const char DeckGen::Ranks[13] = {'A', '2', '3', '4',
                                 '5', '6', '7', '8',
                                 '9', 'T', 'J', 'Q', 'K' };

int main(){
  // Tools.h++ collection:
  RWTValOrderedVector<Card> deck;
  RWTValOrderedVector<Card>::size_type pos;

  Card aceOfSpades('A','S');
  Card firstCard;

  // Use standard library algorithm to generate deck:
  generate_n(back_inserter(deck.std()), 52, DeckGen());
  cout << endl << "The deck has been created" << endl;

  // Use Tools.h++ member function to find card:
  pos = deck.index(aceOfSpades);
  cout << "The Ace of Spades is at position " << pos+1 << endl;
```

```
  // Use standard library algorithm to shuffle deck:
  random_shuffle(deck.begin(), deck.end());
  cout << endl << "The deck has been shuffled" << endl;

  // Use Tools.h++ member functions:
  pos = deck.index(aceOfSpades);
  firstCard = deck.first();

  cout << "Now the Ace of Spades is at position " << pos+1
       << endl << "and the first card is " << firstCard << endl;
}


/* Output (will vary because of the shuffle):

The deck has been created
The Ace of Spades is at position 1

The deck has been shuffled
Now the Ace of Spades is at position 37
and the first card is Q-D

*/
```

Using Templates Without the Standard Library

Several of the *Tools.h++* templates, such as *RWTValVector<T>*, *RWTPtrVector<T>*, *RWTIsvSlist<T>*, and *RWTIsvDlist<T>*, are not based on the Standard C++ Library. You can use them on any of our certified platforms. Also, as mentioned previously in the Introduction, you can use many of the so-called standard library-based templates without the Standard C++ Library, as long as you keep to a subset of the full interface.

Keeping the Standard C++ Library in Mind for Portability

The restricted subset interfaces are almost fully upward compatible with their corresponding standard library-based interfaces. The major difference you will find is that some collections take a different number of template parameters, depending on which underlying implementation they are using. For example, when *RWTPtrHashSet* is used with the Standard C++ Library, it takes three template arguments as described in “Hash Functors and Equalitors” on

page 98. However, when that same class is used without the Standard C++ Library, the restricted interface calls for only one template parameter, namely the type of item being contained. To help you write portable code that works with or without the Standard C++ Library, *Tools.h++* provides two macros:

1. Use the first macro, `RWDefHArgs(T)`, standing for *R*ogue *W*ave *D*efault *H*ash *A*rguments, for the hash-based template collections. For example, by declaring:

   ```
   RWTPtrHashSet<int   RWDefHArgs(int)> hset;
   ```

   you declare a hash-based set that will have the same semantics whether or not the Standard C++ Library is present. Note that you should not use a comma between the element type and the macro. Without the Standard C++ Library, the macro expands to nothing and it is as if you had declared:

   ```
   RWTPtrHashSet<int> hset;
   ```

   However, as soon as the Standard C++ Library becomes available, the macro expands as follows:

   ```
   RWTPtrHashSet<int ,RWTHasher<int>, equal_to<int> > hset;
   ```

   This declaration satisfies the full requirement of the standard library-based interface for all three parameters, and keeps the semantics consistent with the alternative non standard-library based implementation.

2. The second macro, `RWDefCArgs(T)`, is similar to the first. Standing for *R*ogue *W*ave *D*efault *C*omparison *A*rguments, `RWDefCArgs(T)` is available for use with *RWTPtrSortedVector* and *RWTValSortedVector*. For example:

   ```
   RWTValSortedVector<int   RWDefCArgs(int)> srtvec;
   ```

   is a portable declaration that will work with or without the Standard C++ Library. Again, do not use a comma to separate the element type from the macro.

An Example

Let's start with a simple example that uses *RWTValVector<T>*, one of the classes that is not based on the Standard C++ Library.

```
#include <rw/tvvector.h>             // 1

main() {
    RWTValVector<double> vec(20, 0.0);                    // 2

    int i;
    for (i=0; i<10; i++)     vec[i] = 1.0;                // 3
    for (i=11; i<20; i++)    vec(i) = 2.0;                // 4

    vec.reshape(30);                                      // 5
    for (i=21; i<30; i++)    vec[i] = 3.0;                // 6
    return 0;
    }
```

Each program line is detailed below.

`//1` This is where the template for *RWTValVector<T>* is defined.

`//2` A vector of doubles, 20 elements long and initialized to 0.0, is declared and defined.

`//3` The first 10 elements of the vector are set to 1.0. Here, `RWValVector<double>::operator[](int)` has been used. This operator always performs a bounds check on its argument.

`//4` The next 10 elements of the vector are set to 2.0. In this case, `RWValVector<double>::operator()(int)` has been used. This operator generally does not perform a bounds check.

`//5` Member function `reshape(int)` changes the length of the vector.

`//6` Finally, the last 10 elements are initialized to 3.0.

Another Example

The second example involves a hashing dictionary. By using the macro `RWDefHArgs(T)` when you declare the hashing dictionary, you insure that your code is portable with or without access to the Standard C++ Library.

```
#include <rw/tvhdict.h>
#include <rw/cstring.h>
#include <rw/rstream.h>
#include <iomanip.h>

class Count {                                                    // 1
  int  N;
public:
  Count() :    N(0) { }                                          // 2
  int  operator++()  { return ++N; }                             // 3
  operatorint() { return N; }                                   // 4
};

unsigned hashString ( const RWCString& str )                    // 5
  { return str.hash(); }

main() {

  RWTValHashDictionary<RWCString,
                       Count  /* Note: no comma here! */
                       RWDefHArgs(RWCString)> hmap(hashString); //6


  RWCString token;
  while ( cin >> token )                                         // 7
    ++hmap[token];                                               // 8

  RWTValHashDictionaryIterator<RWCString,Count> next(hmap);      // 9

  cout.setf(ios::left, ios::adjustfield);                       // 10
  while ( ++next )                                              // 11
    cout << setw(20) << next.key()
         << " " << setw(10) << next.value() << endl;            // 12

  return 0;
}
```

Program Input:

```
How much wood could a woodchuck chuck if a woodchuck could chuck
wood ?
```

Program Output:

```
much                    1
wood                    2
a                       2
if                      1
woodchuck               2
could                   2
chuck                   2
How                     1
?                       1
```

In the code above, the problem is to read an input file, break it up into tokens separated by white space, count the number of occurrences of each token, and then print the results. The general approach is to use a dictionary to map each token to its respective count. Here's a line-by-line description:

`//1` This is a class used as the value part of the dictionary.

`//2` A default constructor is supplied that zeros out the count.

`//3` We supply a prefix increment operator. This will be used to increment the count in a convenient and pleasant way.

`//4` A conversion operator is supplied that allows `Count` to be converted to an `int`. This will be used to print the results. Alternatively, we could have supplied an overloaded `operator<<()` to teach a `Count` how to print itself, but this is easier.

`//5` This is a function that must be supplied to the dictionary constructor. Its job is to return a hash value given an argument of the type of the key. Note that *Tools.h++* supplies static `hash` member functions for classes *RWCString*, *RWDate*, *RWTime*, and *RWWString* that can be used in place of a user-supplied function. To keep the example general, we chose a user-defined function rather than one of the static `hash` member functions defined by *Tools.h++*.

`//6` Here the dictionary is constructed. Given a key, the dictionary can be used to look up a value. In this case, the key will be of type *RWCString*, the value of type `Count`. The constructor requires a single argument: a pointer to a function that will return a hash value, given a key. This function was defined on line 5 above. Note that we used the `RWDefHArgs(T)` macro to ensure that the program will be portable among platforms with and without the Standard C++ Library.

`//7` Tokens are read from the input stream into an *RWCString*. This will continue until an EOF is encountered. How does this work? The expression `cin >> token` reads a single token and returns an `ostream&`. Class *ostream* has a type conversion operator to `void*`, which is what the while loop will actually be testing. `Operator void*` returns `this` if the stream state is "good", and zero otherwise. Because an EOF causes the stream state to turn to "not good", the while loop will be broken when an EOF is encountered. See the *RWCString* entry in the *Class Reference*, and the *ios* entry in the class reference guide that comes with your compiler.

`//8` Here's where all the magic occurs. Object `map` is the dictionary. It has an overloaded `operator[]` that takes an argument of the type of the key, and returns a reference to its associated value. Recall that the type of the value is a `Count`. Hence, `map[token]` will be of type `Count`. As we saw on line 3, `Count` has an overloaded prefix increment operator. This is invoked on the `Count`, thereby increasing its value.

What if the key isn't in the dictionary? Then the overloaded `operator[]` will insert it, along with a brand new value built using the default constructor of the value's class. This was defined on line 2 to initialize the count to zero.

`//9` Now it comes time to print the results. We start by defining an iterator that will sweep over the dictionary, returning each key and value.

`//10` The field width of the output stream is adjusted to make things pretty.

`//11`	The iterator is advanced until it reaches the end of the collection class. For all template iterators, the prefix increment operator advances the iterator, then tests whether it has gone past the end of the collection class.
`//12`	The key and value at the position of the iterator are printed.

Migration Guide: For Users of Previous Versions of Tools.h++

As we explained in the introduction to this manual, one of our primary goals for this version of *Tools.h++* is to protect your investment in existing code based on previous versions of the library. As you can see from this chapter, we have significantly re-engineered the collection class templates in order to bring them up to date with the Standard C++ Library. The following classes were re-engineered:

RWTPtrDlist	*RWTValDlist*
RWTPtrHashDictionary	*RWTValHashDictionary*
RWTPtrHashSet	*RWTValHashSet*
RWTPtrHashTable	*RWTValHashTable*
RWTPtrOrderedVector	*RWTValOrderedVector*
RWTPtrSlist	*RWTValSlist*
RWTPtrSortedVector	*RWTValSortedVector*

You have seen that you can now use all of these classes either with or without the Standard C++ Library. Used without the Standard C++ Library, they have the same interfaces and implementations as in the previous version of *Tools.h++*, updated with some bug fixes. These minor enhancements should not cause any source incompatibilities with existing code.

You may need to make a few changes to existing source code when using the above classes with the Standard C++ Library. The adjustments required for specific classes are outlined below.

- Extra template arguments are now required for:

RWTPtrHashDictionary	*RWTValHashDictionary*
RWTPtrHashSet	*RWTValHashSet*
RWTPtrHashTable	*RWTValHashTable*
RWTPtrSortedVector	*RWTValSortedVector*

Existing code using these templates will not provide the number of template arguments expected by this version of *Tools.h++* when used with the Standard C++ Library. The solution to this problem is to use the macros discussed in

"Keeping the Standard C++ Library in Mind for Portability" on page 106. Using the macros described there will satisfy the compiler and preserve the semantics of your existing code.

- The class hierarchy has changed for:

RWTPtrHashSet	*RWTValHashSet*
RWTPtrHashTable	*RWTValHashTable*
RWTPtrOrderedVector	*RWTValOrderedVector*
RWTPtrSortedVector	*RWTValSortedVector*

If you have existing code that makes use of any of the inheritance relationships among the collection class templates, that code will not compile with this version of *Tools.h++* when used with the Standard C++ Library. There are no inheritance relationships among the standard library-based implementations of the collection class templates. For example, in the previous version of *Tools.h++*, *RWTPtrHashSet* inherited from *RWTPtrHashTable*, *RWTValOrderedVector* inherited from *RWTValVector*, and *RWTValSortedVector* inherited from *RWTValOrderedVector*. The pointer-based versions of these templates followed a similar pattern. These relationships do not hold in the new version of *Tools.h++*. If you have code based on this inheritance, you will need to modify it .

The most likely place where you will find this problem in your existing code is when building an *RWTValHashTableIterator* from an *RWTValHashSet*, or an *RWTPtrHashTableIterator* from an *RWTPtrHashSet*. Because the constructor for *RWTValHashTableIterator* is expecting a reference to an *RWTValHashTable*, passing in an *RWTValHashSet* instead depends on the inheritance relationship.

The solution to this particular problem is found in the new class *RWTPtrHashSetIterator*. Wherever you find code which constructs an *RWTValHashTableIterator* from an *RWTValHashSet*, use an *RWTValHashSetIterator* instead. Note that *RWTValHashSetIterator* is provided whether or not the Standard C++ Library is available, so you can modify your code now in anticipation of migrating your code to the standard library-based implementations.

- Required *less-than* semantics for an element type may affect:

RWTPtrDlist	*RWTValDlist*
RWTPtrOrderedVector	*RWTValOrderedVector*
RWTPtrSlist	*RWTValSlist*
RWTPtrSortedVector	*RWTValSortedVector*

As mentioned above, some compilers will require that the expression `(t1 < t2)` be defined for two instances of your element type. This is due to the inclusion of convenient member functions, such as `sort()` and `min_element()`, combined with certain compilers that instantiate all member functions whether used or not. You might have existing code that instantiates one of these templates on a type `T` for which no `operator<()` is defined. If that is the case, you will have to define one.

The best thing would be to define it in a way you can really use, if you ever use those member functions which really do require it. The quick and dirty approach would be to globally define a dummy `operator<()` whose only purpose is to appease the compiler. Our experience is that code written "just to appease the compiler" constitutes a maintenance nightmare. Please avoid it if at all possible.

Generic Collection Classes 12

Generic collection classes are the second major category of collection classes included in *Tools.h++*. We call them generic because they use the macros defined in `<generic.h>`, an early approximation to parameterized types first described in Stroustrup (1986, p. 209). Generic collection classes are less manageable than true templates[1], but they are portable to any C++ compiler. You can use them even with older compilers.

Most of the generic collection classes use reference-based semantics; that is, they store and retrieve pointers to other objects, as described in "Storage Methods of Collection Classes" on page 77. With these classes, as with all Rogue Wave collection classes, you are responsible for the allocation and deallocation of the objects themselves.

Three vector-based generic collections use value-based semantics: *RWGVector(val)*, *RWGOrderedVector(val)*, and *RWGSortedVector(val)*. These classes store the type itself,which could be a pointer to an object.

The storage and retrieval methods and criteria differ from class to class.

1. Actually, the generic macros are easy to use, but difficult to write. They are also difficult to debug because they are preprocessor macros, and most debuggers cannot enter macro code.

12

Example

Here is an example that uses an *RWGStack* , a generic stack, to store a set of pointers to `ints` in a last-in, first-out (LIFO) stack. We will go through it line-by-line and explain what is happening:

```
#include <rw/gstack.h>                                          //1
#include <rw/rstream.h>                                         //2

declare(RWGStack, int)                                          //3

main(){
   RWGStack(int) gs;                                            //4
   gs.push(new int(1));                                         //5
   gs.push(new int(2));                                         //6
   gs.push(new int(3));                                         //7
   gs.push(new int(4));                                         //8

   cout << "Stack now has " << gs.entries()
        << " entries\n";                                        //9

   int* ip;                                                     //10
   while( ip = gs.pop() )                                       //11
   {
     cout << *ip << "\n";                                       //12
     delete ip;
   }
   return 0;
}
```

Program Output:

```
Stack now has 4 entries
4
3
2
1
```

Each line of the program is detailed below.

`//1` This `#include` defines the preprocessor macro `RWGStackdeclare(type)`. This macro is an elaborate and ugly-looking thing that continues for many lines and describes how a generic stack of objects of type `type` should behave. Mostly, the macro serves as a *restricted interface* to the underlying implementation, which is a singly-linked list, class *RWSlist*. It is *restricted* because it can use only those member functions of *RWSlist* appropriate to stacks, and insert into the stack only items of type `type`.

`//2` `<rw/rstream.h>` is a special Rogue Wave header file that includes `<iostream.h>` with a suffix that depends on your compiler.

`//3` This line invokes the macro `declare`, which is defined in the header file `<generic.h>` supplied with your compiler. If called with arguments `declare(Class, type)`, it calls the macro `Classdeclare` with argument `type`. In this case, the macro `RWGStackdeclare`, defined in `<rw/gstack.h>`, will be called with argument `int`.

In other words, the result of calling the `declare(RWGStack, int)` macro is to create a new class especially for your program. For all practical purposes, its name is *RWGStack(int)*, a stack of pointers to `int`s.

`//4` At this line an instance `gs` of the new class *RWGStack(int)* is created.

`//5-//8`

Four `int`s are created off the heap and inserted into the stack. After statement 8 executes, a pointer to the `int` 4 will be at the top of the stack, and a pointer to the `int` 1 at the bottom.

`//9` The member function `entries()` of class *RWGStack(int)* is called to verify how many items are on the stack.

`//10` A pointer to an `int` is declared and defined.

`//11` The stack is popped until empty. The member function `pop()` will return and remove a pointer to the item on the top of the stack. If there are no more items on the stack it will return zero, causing the `while` loop to terminate.

`//12` Each item is dereferenced and printed.

12

Declaring Generic Collection Classes

All the *Tools.h++* generic collection classes are declared as in the example above, using the `declare` macro defined in the header file `<generic.h>`. However, there is one important difference in how the *Tools.h++* classes are declared versus the pattern set by Stroustrup (1986, Section 7.3.5). The difference is summarized below:

```
typedef int* intP;
declare(RWGStack, intP)                          //Wrong!
declare(RWGStack, int)                           //Correct.
```

In Stroustrup, the class is declared using a typedef for a pointer to the collected item. The Rogue Wave generic classes are all declared using the item name itself. This is true for both the reference-semantics and value-semantics classes.

User-Defined Functions

Some of the member functions of the generic collection classes require a pointer to a user-defined function. There are two kinds of these user-defined functions, discussed in the following two sections.

Tester Functions

The first kind of user-defined function is a tester function. It has the form:

```
RWBoolean tester(const type* ty, const void* a)
```

where `tester` is the name of the function, `type` is the type of the members of the collection class, and `RWBoolean` is a typedef for an `int` whose only possible values are `TRUE` or `FALSE`. The job of the tester function is to signal when a certain member of the collection has been identified. The decision of how this is done, or what it means to have identified an object, is left to the user. You can choose to compare addresses (test for two objects being `identical`), or to look for certain values within the object (test for `isEqual`). The first variable `ty`, which can be thought of as a *candidate*, will point to a member of the collection. The second variable `a`, which can be thought of as *client data*, can be tested against `ty` for a match.

In the following example, which expands on the previous one, the problem is to test for `isEqual`. We push some values onto a stack to see if a certain value exists on the stack. The member function `contains()` of class *RWGStack(type)* has prototype:

```
RWBoolean contains(RWBoolean (*t)(const type*, const void*),
                   const void* a) const;
```

The first argument is `RWBoolean (*t)(const type*, const void*)`. This is a pointer to the tester function, for which we will have to provide an appropriate definition:

```
#include <rw/gstack.h>
#include <rw/rstream.h>

declare(RWGStack, int)

RWBoolean myTesterFunction(const int* jp, const void* a)     //1
{  return *jp == *(const int*)a;                             //2
}

main(){
   RWGStack(int) gs;                                         //3
   gs.push(new int(1));                                      //4
   gs.push(new int(2));                                      //5
   gs.push(new int(3));                                      //6
   gs.push(new int(4));                                      //7

   int aValue = 2;                                           //8

   if ( gs.contains(myTesterFunction, &aValue) )             //9
     cout << "Yup.\n";
   else
     cout << "Nope.\n";

5while(!gs.isEmpty())
   delete gs.pop();

return 0;
}
```

Program Output:

```
Yup.
```

A description of each program line follows.

`//1` This is the tester function. Note that the first argument is a pointer to the type of objects in the collection, `int`s in this case. The second argument points to an object that can be of any type–also an `int` in this example. Note that both arguments are declared `const` pointers; in general, the tester function should not change the value of the objects being pointed to.

`//2` The second argument is converted from a `const void*` to a `const int*`, then dereferenced. The result is a `const int`. This `const int` is compared to the dereferenced first argument, which is also a `const int`. The net result is that this tester function considers a match to occur when the two `int`s have the same values (i.e., they are *equal*). Note that we could also choose to identify a *particular* `int` (i.e., test for *identity*).

`//3-//7`

These lines are the same as in the example in “Example” on page 116. A generic stack of (pointers to) `int`s is declared and defined, then 4 values are pushed onto it.

`//8` This is the value (i.e., 2) that we will look for in the stack.

`//9` Here the member function `contains()` is called, using the tester function. The second argument of `contains()`, a pointer to the variable `aValue`, will appear as the second argument of the tester function. The function `contains()` traverses the entire stack, calling the tester function for each item in turn, and waiting for the tester function to signal a match. If it does, `contains()` returns `TRUE`; otherwise, `FALSE`.

Note that the second argument of the tester function does not necessarily have to be of the same type as the members of the collection, although it is in the example above. In the following example, the argument and members of the collection are of different types:

```
#include <rw/gstack.h>
#include <rw/rstream.h>

class Foo {
public:
  int data;
  Foo(int i) {data = i;}
};

declare(RWGStack, Foo)  // A stack of pointers to Foos

RWBoolean anotherTesterFunction(const Foo* fp, const void* a)
{  return fp->data == *(const int*)a;
}

main(){
   RWGStack(Foo) gs;
   gs.push(new Foo(1));
   gs.push(new Foo(2));
   gs.push(new Foo(3));
   gs.push(new Foo(4));

   int aValue = 2;
   if ( gs.contains(anotherTesterFunction, &aValue) )
     cout << "Yup.\n";
   else
     cout << "Nope.\n";

while(!gs.isEmpty())
     delete gs.pop();
   return 0;
}
```

In this example, a stack of (pointers to) *Foos* is declared and used, while the variable being passed as the second argument to the tester function is still a `const int*`. The tester function must take the different types into account.

Apply Functions

The second kind of user-defined function is an apply function. Its general form is:

`void` *`yourApplyFunction`*`(`*`type`*`* ty, void* a)`

where `yourApplyFunction` is the name of the function, and `type` is the type of the members of the collection. Apply functions give you the opportunity to perform some operation on each member of a collection, perhaps print it out or draw it on a screen. The second argument is designed to hold client data to be used by the function, perhaps the handle of a window on which the object is to be drawn.

In the following example, the apply function `printAFoo` is used to print out the value of each member in *RWGDlist(type)*, a generic doubly-linked list:

```
#include <rw/gdlist.h>
#include <rw/rstream.h>

class Foo {
public:
  int val;
  Foo(int i) {val = i;}
};

declare(RWGDlist, Foo)

void printAFoo(Foo* ty, void* sp){
   ostream* s = (ostream*)sp;
   (*s) << ty->val << "\n";
}

main(){
   RWGDlist(Foo) gd;
   gd.append(new Foo(1));
   gd.append(new Foo(2));
   gd.append(new Foo(3));
   gd.append(new Foo(4));
```

```
   gd.apply(printAFoo, &cout);

   while(!gd.isEmpty())
      delete gd.get();
   return 0;
}
```

Program Output:

```
1
2
3
4
```

The items are appended at the tail of the list. For each item, the `apply()` function calls the user-defined function `printAFoo()` with the address of the item as the first argument, and the address of an `ostream` (an output stream) as the second argument. The job of `printAFoo()` is to print out the value of member data `val`. Because `apply()` scans the list from beginning to end, the items will come out in the same order in which they were inserted. See the Class Reference for *RWGDlist(type)*.

With some care, you can use apply functions to change the objects in a collection. For example, you could change the value of member data `val` in the example above, or delete all member objects. In the latter case, however, you must be careful not to use the collection again.

12

Smalltalk-Like Collection Classes 13

The Smalltalk-like collection classes are the third general type of collection class provided by *Tools.h++*. Based on the language Smalltalk-80, these collections require that their collected objects singly or multiply inherit from the abstract base class *RWCollectable*.

These collection classes have a few disadvantages: they are slightly slower and not completely typesafe, and their objects are slightly larger. These disadvantages are easily outweighed by the power of these classes, and their clean programming interface. Most importantly, the Smalltalk-like collection classes are well-suited for heterogeneous collections and polymorphic persistence.

Many of the *Tools.h++* Smalltalk-like classes have a typedef to either the corresponding Smalltalk names, or to a generic name. This typedef is activated by defining the preprocessor macro `RW_STD_TYPEDEFS`. Although you are free to use typedefs, we do encourage you to use the actual class names to make your code more maintainable.

13

Tables of the Smalltalk-like Classes

The following two tables summarize the *Tools.h++* Smalltalk-like classes. Table 1 lists all 17 classes, along with their typedefs, iterators, and implementations. Table 2 illustrates the class hierachy.

Table 13-1 Smalltalk-like collection classes, their iterators, typedefs, and implementations

Class	Iterator	Smalltalk typedef (deprecated)	Implemented as
RWBag	*RWBagIterator*	*Bag*	Dictionary of occurrences
RWBinaryTree	*RWBinaryTreeIterator*	*SortedCollection*	Binary tree
RWBTree			B-tree in memory
RWBTreeDictionary			B-tree of associations
RWCollection	*RWIterator*	*Collection*	Abstract base class
RWDlistCollectables	*RWDlistCollectablesIterator*		Doubly-linked list
RWHashTable	*RWHashTableIterator*		Hash table
RWHashDictionary	*RWHashDictionaryIterator*	*Dictionary*	Hash table of associations
RWIdentityDictionary	*RWHashDictionaryIterator*	*IdentityDictionary*	Hash table of associations
RWIdentitySet	*RWSetIterator*	*IdentitySet*	Hash table
RWOrdered	*RWOrderedIterator*	*OrderedCollection*	Vector of pointers
RWSequenceable	*RWIterator*	*Sequenceable*	Abstract base class
RWSet	*RWSetIterator*	*Set*	Hash table
RWSlistCollectables	*RWSlistCollectablesIterator*	*LinkedList*	Singly-linked list
RWSlistCollectables-Queue	*(n/a)*	*Queue*	Singly-linked list
RWSlistCollectables-Stack	*(n/a)*	*Stack*	Singly-linked list
RWSortedVector	*RWSortedVectorIterator*		Vector of pointers, using insertion sort

Table 13-2 The class hierarchy of the Smalltalk-like collection classes.

Note that some of these classes use multiple-inheritance: this hierarchy is shown relative to the *RWCollectable* base class.

RWCollectable
- *RWCollection (abstract base class)*
  - *RWBinaryTree*
  - *RWBTree*
    - *RWBTreeDictionary*
  - *RWBag*
  - *RWSequenceable (abstract base class)*
    - *RWDlistCollectables (Doubly-linked lists)*
    - *RWOrdered*
      - *RWSortedVector*
    - *RWSlistCollectables (Singly-linked lists)*
      - *RWSlistCollectablesQueue*
      - *RWSlistCollectablesStack*
  - *RWHashTable*
    - *RWSet*
      - *RWIdentitySet*
      - *RWHashDictionary*
        - *RWIdentityDictionary*

Example

To orient ourselves, we always like to start with an example. The following example uses a *SortedCollection* to store and order a set of *RWCollectableStrings*. *SortedCollection* is actually a typedef for the Smalltalk-like collection class

RWBinaryTree. Objects inserted into it are stored in order according to their relative values as returned by the virtual function `compareTo()`. (See "Add Definitions for Virtual Functions" on page 198). Here is the code:

```
#define RW_STD_TYPEDEFS 1                                        //1
#include <rw/bintree.h>
#include <rw/collstr.h>                                          //2
#include <rw/rstream.h>

main(){
// Construct an empty SortedCollection
SortedCollection sc;                                             //3

// Insert some RWCollectableStrings:
sc.insert(new RWCollectableString("George"));                   // 4
sc.insert(new RWCollectableString("Mary"));                     // 5
sc.insert(new RWCollectableString("Bill"));                     // 6
sc.insert(new RWCollectableString("Throkmorton"));              // 7

// Now iterate through the collection printing all members:
RWCollectableString* str;                                       // 8
SortedCollectionIterator sci(sc);                               // 9
  while( str = (RWCollectableString*)sci() )                   // 10
    cout << *str << endl;                                      // 11

sc.clearAndDestroy();
return 0;
}
```

Program Output:

```
Bill
George
Mary
Throkmorton
```

Let's go through the code line-by-line and explain what is happening:

`//1` By defining the preprocessor macro `RW_STD_TYPEDEFS`, we enable the set of Smalltalk-like typedefs. We can then use the typedef *SortedCollection* instead of *RWBinaryTree*, its true identity.

`//2` The second `#include` declares class *RWCollectableString*, a derived class that multiply inherits from its base classes *RWCString* and *RWCollectable*. *RWCollectableString* inherits functionality from *RWCString*, and "ability to be collected" from class *RWCollectable*.

`//3` An empty *SortedCollection* was created at this line.

`//4-//7`
Four *RWCollectableString*s were created off the heap and inserted into the collection, in no particular order. See the *Class Reference* for details on constructors for *RWCollectableString*s. The objects allocated here normally should be deleted before the end of the program, but we omitted this step to make the example more concise.

`//8` A pointer to an *RWCollectableString* was declared and defined here.

`//9` An iterator was constructed from the *SortedCollection* `sc`.

`//10` The iterator is then used to step through the entire collection, retrieving each value in order. The function call operator `operator()`has been overloaded so that the iterator means "step to the next item and return a pointer to it." All *Tools.h++* iterators work this way. See Stroustrup (1986, Section 7.3.2) for an example and discussion of iterators, as well as "Iterators in Collection Classes" on page 83 1 of this manual. The typecast:

```
str = (RWCollectableString*)sci()
```

is necessary because the iterator returns an `RWCollectable*`; that is, a pointer to an *RWCollectable* which must then be cast into its actual identity.

`//11` Finally, the pointer `str` is dereferenced and printed. The ability of an *RWCollectableString* to be printed is inherited from its base class *RWCString*.

When run, the program prints out the four collected strings in order; for class *RWCollectableString*, this means lexicographical order.

13

Choosing a Smalltalk-like Collection Class

Now that you have reviewed the list of Smalltalk-like collection classes, their class hierarchy, and an example of how they work, you may be wondering how you can use them. This section gives an overview of the various Smalltalk-like collection classes to help you choose an appropriate one for your problem. You can also see Appendix A, on choosing.

Bags Versus Sets Versus Hash Tables

Class *RWHashTable* is the easiest Smalltalk-like collection class to understand. It uses a simple hashed lookup to find the *bucket* where a particular object occurs, then does a linear search of the bucket to find the object. A key concept is that multiple objects that test `isEqual` to each other can be inserted into a hash table.

Class *RWBag* is similar to *RWHashTable*, except that it *counts* occurrences of multiple objects with the same value; that is, it retains only the first occurrence and merely increments an occurrence count for subsequent ones. *RWBag* is implemented as a dictionary, where the key is the inserted object and the value is the occurrence count. This is the same way the Smalltalk *Bag* object is implemented. Note that this implementation differs significantly from many other C++ *Bag* classes which are closer to the *RWHashTable* class and not true *Bag*s.

Class *RWSet* is similar to its base class *RWHashTable*, except that it doesn't allow duplicates. If you try to insert an object that `isEqual` to an object already in *RWSet*, the object will be rejected.

Class *RWIdentitySet*, which inherits from *RWSet*, retrieves objects on the basis of *identity* instead of value. Because *RWIdentitySet* is a *Set*, it can take only one *instance* of a given object.

Note that the ordering of objects in any of these classes based on hash tables is not meaningful. If ordering is important, you should choose a sequenceable class.

Sequenceable Classes

Classes inheriting from *RWSequenceable* have an innate ordering. You can speak meaningfully of their first or last object, or of their 6th or `i`th object.

These classes are implemented generally as either a vector, or a singly-linked or doubly-linked list. Vector-based classes make good stacks and queues, but poor insertions in the middle. If you exceed the capacity of a vector-based collection class, it will automatically resize, but it may exact a significant performance penalty to do so.

Note that the binary and B-tree classes can be considered *sequenceable* in the sense that they are sorted, and therefore have an innate ordering. However, their ordering is determined internally by the relative value of the collected objects, rather than by an insertion order. In other words, you cannot arbitrarily insert an object into a sorted collection in any position you want because it might not remain sorted. Hence, these classes are subclassed separately.

Dictionaries

Sometimes referred to as *maps*, dictionaries use an external *key* to find a *value*. The key and value may be of different types, and in fact usually are. You can think of dictionaries as associating a given key with a given value. For example, if you were building a symbol table in a compiler, you might use the symbol name as the key, and its relocation address as the value. This approach would contrast with your approach for a *Set*, where the name and address would have to be encapsulated into *one* object.

Tools.h++ provides two dictionary classes: *RWHashDictionary*, implemented as a hash table, and *RWBTreeDictionary*, implemented as a B-tree. For these classes, both keys and values must inherit from the abstract base class *RWCollectable*.

Virtual Functions Inherited From RWCollection

The Smalltalk-like collection classes inherit from the abstract base class *RWCollection*, which in turn inherits from the abstract base class *RWCollectable*, described in “Tables of the Smalltalk-like Classes” on page 126 and Chapter 15. (Thus do we produce collections of collections, but that is another story.)

An *abstract base class* is a class intended to be inherited by some other class, not used as itself *per se*. If you think of it as a kind of virtual class, you can easily project the meaning of virtual functions. These virtual functions provide a

blueprint of functionality for the derived class. As an abstract base class, *RWCollection* provides a blueprint for collection classes by declaring various virtual functions, such as `insert()`, `remove()`, `entries()`, and so on.

This section describes the virtual functions inherited by the Smalltalk-like collections. Any of these collections can be expected to understand them.

insert()

You can put a pointer to an object into a collection by using the virtual function `insert()`:

```
virtual RWCollectable*  insert(RWCollectable*);
```

This function inserts in the way most natural for the collection. Faced with a stack, it pushes an item onto the stack. Faced with a queue, it appends the item to the queue. In a sorted collection, it inserts the new item so that items before it compare less than itself, items after it compare greater than itself, and items equal compare equal, if duplicates are allowed. See “Example” on page 127 for an example using `insert()`.

You must always insert pointers to real objects. Since all *RWCollection* classes need to dereference their contents for some methods such as `find()`, inserting a zero will cause such methods to crash. If you must store an empty object, we suggest you create and insert a default constructed object of the appropriate type, such as *RWCollectable**. If you know you won’t be deleting every object in the *RWCollection*, you could also choose the global *RWnilCollectable*, which is a pointer to a cached *RWCollectable* object. But beware! Since there is only one *RWnilCollectable*, if you delete it, any other reference to it will be referencing invalid memory.

find() and Friends

You can use the following virtual functions to test how many objects a collection contains, and whether it contains a particular object:

```
virtual RWBoolean       contains(const RWCollectable*) const;
virtual unsigned        entries() const;
virtual RWCollectable*  find(const RWCollectable*) const;
virtual RWBoolean       isEmpty() const;
virtual unsigned        occurrencesOf(const RWCollectable*) const;
```

The function `isEmpty()` returns `TRUE` if the collection contains no objects. The function `entries()` returns the total number of objects that the collection contains.

The function `contains()` returns `TRUE` if the argument is equal to an item within the collection. The meaning of *is equal to* depends on the collection and the type of object being tested. Hashing collections use the virtual function `isEqual()` to test for equality, after first hashing the argument to reduce the number of possible candidates to those in one hash bucket. (Here it is important that all items which are `isEqual` with each other hash to the same value!). Sorted collections search for an item that *compares equal* to the argument; in other words, an item for which `compareTo()` returns zero.

The virtual function `occurrencesOf()` is similar to `contains()`, but returns the number of items that are equal to the argument.

The virtual function `find()` returns a pointer to an item that is equal to its argument.

The following example, which builds on the example in “Example” on page 127, uses `find()` to find occurrences of Mary in the collection, and `occurrencesOf` to find the number of times Mary occurs:

```
#define RW_STD_TYPEDEFS 1
#include <rw/bintree.h>                                          //1
#include <rw/collstr.h>                                          //2
#include <rw/rstream.h>

main(){
// Construct an empty SortedCollection
SortedCollection sc;                                             //3

// Insert some RWCollectableStrings:
sc.insert(new RWCollectableString("George"));                    //4
sc.insert(new RWCollectableString("Mary"));                      //5
sc.insert(new RWCollectableString("Bill"));                      //6
sc.insert(new RWCollectableString("Throkmorton"));               //7
sc.insert(new RWCollectableString("Mary"));                      //8
```

```
cout << sc.entries() << "\n";                                    //9

RWCollectableString dummy("Mary");                              //10
RWCollectable* t = sc.find( &dummy );                           //11

if(t){                                                          //12
  if(t->isA() == dummy.isA())                                   //13
    cout << *(RWCollectableString*)t << "\n";                   //14
  }
else
  cout << "Object not found.\n";                                //15

cout << sc.occurrencesOf(&dummy) << "\n";                       //16

sc.clearAndDestroy();
return 0;
}
```

Program Output:

```
5
Mary
2
```

Here's the line-by-line description:

`//1-//7` These lines are from "Example" on page 127.

`//8` Insert another instance with the value `Mary`.

`//9` This statement prints out 5, the total number of entries in the sorted collection.

`//10` A throwaway variable `dummy` is constructed, to be used to test for the occurrences of strings containing `Mary`.

`//11` The collection is asked to return a pointer to the first object encountered that compares equal to the argument. A nil pointer (zero) is returned if there is no such object.

`//12` The pointer is tested to make sure it is not nil.

`//13` Paranoid check. In this example, it is obvious that the items in the collection must be of type *RWCollectableString*. In general, it may not be obvious.

`//14` Because of the results of step 13, the cast to an *RWCollectableString* pointer is safe. The pointer is then dereferenced and printed.

`//15` If the pointer `t` was nil, then an error message would have been printed here.

`//16` The call to `occurrencesOf()` returns the number of items that compare equal to its argument. In this case, two items are found, the two occurrences of `Mary`.

remove() Functions

To search for and remove particular items, you can use the functions `remove()` and `removeAndDestroy()`:

```
virtual RWCollectable*  remove(const RWCollectable*);
virtual void            removeAndDestroy(const RWCollectable*);
```

The function `remove()` looks for an item that is equal to its argument and removes it from the collection, returning a pointer to it. It returns nil if no item is found.

The function `removeAndDestroy()` is similar except it deletes the item instead of returning it, using the virtual destructor inherited by all *RWCollectable* items. You must be careful when using this function that the item was actually allocated off the heap, *not* the stack, and that it is *not* shared with another collection. Also note that *RWnilCollectable* references a cached default *RWCollectable*–be careful not to destroy it!

The following example, which expands on the previous one, demonstrates the use of the virtual function `removeAndDestroy()`:

```
RWCollectable* oust = sc.remove(&dummy);                    //17
delete oust;                                                //18

sc.removeAndDestroy(&dummy);                                //19
```

`//17` Removes the first occurrence of the string containing `Mary` and returns a pointer to it. This pointer will be nil if there is no such item.

`//18`	Deletes the item, which was originally allocated off the heap. There is no need to check the pointer against nil because the language guarantees that it is always OK to delete a nil pointer.
`//19`	In this statement, the remaining occurrence of `Mary` is both removed and deleted.

apply() Functions

To efficiently examine the members of a Smalltalk-like collection, use the member function `apply()`:

```
virtual void apply(RWapplyCollectable ap, void* x);
```

The first argument, `RWapplyCollectable`, is a typedef:

```
typedef void  (*RWapplyCollectable)(RWCollectable*, void*);
```

In other words, `RWapplyCollectable` is a pointer to a function with prototype:

```
void yourApplyFunction(RWCollectable* item, void* x)
```

where `yourApplyFunction` is the name of the function. You must supply this function. It will be called for each item in the collection, in whatever order is appropriate for the collection, and passed as a pointer to the item as its first argument. The second argument `x` is passed through from the call to `apply()`, and is available for your use. For example, you could use it to hold a handle to a window on which the object is to be drawn.

Note that the `apply()` functions of the Smalltalk-like collections and the generic collections are similar. (Compare "Apply Functions" on page 122.) The difference is in the type of the first argument of the user-supplied function: the Smalltalk-like collections use `RWCollectable*`, while the generic collections use `type*`. With both sets of collections, you must be careful that you cast the pointer `item` to the proper derived class.

The apply functions generally employ the most efficient method for examining all members of the collection. This is their great advantage. Their disadvantage is that they are slightly clumsy to use, requiring you to supply a separate function[1].

1. The functional equivalent to `apply()` in the Smalltalk world is `do`. It takes just one argument: a piece of code to be evaluated for each item in the collection. This keeps the method and the block to be evaluated together in one place, resulting in cleaner code. As usual, the C++ approach is messier.

Functions clear() and clearAndDestroy()

To remove all items from the collection, you can use the functions `clear()` and `clearAndDestroy()`:

```
virtual voidclear();
virtual voidclearAndDestroy();
```

The function `clearAndDestroy()` not only removes the items, but also calls the virtual destructor for each item. You must use this function with care. The function *does* check to see if the same item occurs more than once in a collection (by building an *RWIdentitySet* internally), and thereby deletes each item only once. However, it cannot check whether an item is shared between two *different* collections. In particular, you should never call `clearAndDestroy()` on a collection which holds an instance of `RWnilCollectable`, which will almost be shared. You must also be certain that every member of the collection was allocated off the heap.

Other Functions Shared by All RWCollections

There are several other functions that are shared by all classes that inherit from *RWCollection*. Note that these are *not* virtual functions.

Class Conversions

The following functions allow any collection class to be converted into an *RWBag*, *RWSet*, *RWOrdered*, or a *SortedCollection* (that is, an *RWBinaryTree*):

```
RWBag         asBag() const;
RWSet         asSet() const;
RWOrdered     asOrderedCollection() const;
RWBinaryTree  asSortedCollection() const
```

Note that since these functions mimic similar Smalltalk methods, they return a *copy* of the new collection by value. For large collections, this can be very expensive. Consider using `operator+=()` instead.

Inserting and Removing Other Collections

You can use these functions to respectively insert or remove the contents of their argument.

```
void  operator+=(const RWCollection&);
void  operator-=(const RWCollection&);
```

Selection

The function `select()`:

```
typedef RWBoolean (*RWtestCollectable)(const RWCollectable*,
                   const void*);
RWCollection*     select(RWtestCollectable tst, void*);
```

evaluates the function pointed to by `tst` for each item in the collection. It inserts those items for which the function returns `TRUE` into a new collection of the same type as self and returns a pointer to it. This new collection is allocated *off the heap*, so you are responsible for deleting it when done.

Virtual Functions Inherited from RWSequenceable

Collections that inherit from the abstract base class *RWSequenceable*, which inherits from *RWCollectable*, have an innate, meaningful ordering. This section describes the virtual functions inherited from *RWSequenceable* which make use of that ordering. For example, the following virtual functions allow access to the `i`th item in the collection:

```
virtual RWCollectable*&       at(size_t i);
virtual const RWCollectable*  at(size_t i) const;
```

Remember that the first item in any collection is at position `i=0`. The compiler chooses which function to use on the basis of whether or not your collection has been declared `const`: the second variant of the function is for `const` collections, the first for all others. The first variant can also be used as an lvalue, as in the following example:

```
RWOrdered od;
od.insert(new RWCollectableInt(0));                          // 0
od.insert(new RWCollectableInt(1));                        // 0 1
od.insert(new RWCollectableInt(2));                      // 0 1 2

delete od(1);              // Use variant available for RWOrdered
od.at(1) = new RWCollectableInt(3);                      // 0 3 2
```

As you might expect, the operations above are efficient for the class *RWOrdered*, which is implemented as a vector, but relatively inefficient for a class implemented as a linked-list, because the entire list must be traversed to find a particular index.

The following virtual functions return the first or last item in the collection, respectively, or nil if the collection is empty:

```
virtual RWCollectable*  first() const;
virtual RWCollectable*  last() const;
```

The next virtual function returns the index of the first object that is equal to the argument, or the special value `RW_NPOS` if there is no such object:

```
virtual size_t  index(const RWCollectable*) const;
```

Here's an example of the index function in use. The output shows that the index of the variable we were searching for was found at position 1.

```
RWOrdered od;
od.insert(new RWCollectableInt(6));                          // 6
od.insert(new RWCollectableInt(2));                        // 6 2
od.insert(new RWCollectableInt(4));                      // 6 2 4

RWCollectableInt dummy(2);
size_t inx = od.index(&dummy);
if (inx == RW_NPOS)
  cout << "Not found.\n";
else
  cout << "Found at index " << inx << endl;
```

Program Output:

```
Found at index 1
```

Finally, you can use the following function to insert an item at a particular index:

```
virtual RWCollectable* insertAt(size_t i, RWCollectable* c);
```

In the example below, the code uses the function `insertAt` to insert 4 at position 1.

```
RWOrdered od;
od.insert(new RWCollectableInt(6));                          // 6
od.insert(new RWCollectableInt(2));                        // 6 2
od.insertAt(1, new RWCollectableInt(4));                 // 6 4 2
```

A Note on How Objects are Found

You may save yourself some difficulty by remembering the following point: *the virtual functions of the object within the collection, not those of the target, are called when comparing or testing a target for equality.*

The following code fragment illustrates the point:

```
SortedCollection sc;
RWCollectableString member;

sc.insert(&member);

RWCollectableString target;
RWCollectableString* p = (RWCollectableString*)sc.find(&target);
```

In this example, the virtual functions of `member` within the collection *RWCollectableString* are called, not the virtual functions of `target`. In other words:

```
member.compareTo(&target);              //This will get called.
target.compareTo(&member);              //Not this.
```

Hashing

Hashing is an efficient method for finding an object within a collection. All the collection classes that use hashing follow the same general strategy. First, member function `hash()` of the target is called to find the proper bucket within the hash table. A buckets is implemented as a singly-linked list. Because all the members of a bucket have the same hash value, the bucket is linearly searched to find the exact match. This is done by calling member function `isEqual()` *of the candidate* (see above) with each member of the bucket as the argument. The first argument that returns `TRUE` is the chosen object. Be careful not to design your class so that two objects that test true for `isEqual()` can have different hash values, since this algorithm will fail for such objects.

In general, because of this combination of hashing and linear searching, as well as the complexity of most hashing algorithms, the ordering of the objects within a hash collection will not make a lot of sense. Hence, when the `apply()` function or an iterator scans through the hashing table, the objects will be visited in what appears to be random order.

13

Persistence 14

Persistence is the ability to save an object to a file or a stream and then restore that object from the file or stream. *Persistence is a very important feature of objects because it facilitates the exchange of objects between processes.* Using persistence and working through streams, you can send objects from one program to another, or from one user to another. You can also save a persistent object to a file on a disk, and restore it from disk at another time, or in another place.

Levels of Persistence

An object has one of four levels of persistence:

- *No persistence.* There is no mechanism for storage and retrieval of the object.
- *Simple persistence.* A level of persistence that provides storage and retrieval of individual objects to and from a stream or file. Simple persistence does not preserve pointer relationships among the persisted objects.
- *Isomorphic persistence.* A level of persistence that preserves the pointer relationships among the persisted objects.
- *Polymorphic persistence.* The highest level of persistence. Polymorphic persistence preserves pointer relationships among the persisted objects and allows the restoring process to restore an object without prior knowledge of that object's type.

The *Class Reference* indicates the level of persistence for each class. This section provides information about each level of persistence through descriptions, examples, and procedures for designing your own persistent classes.

14

A Note About Terminology

Tools.h++ provides input and output classes that let you save and restore objects. These classes are:

- *RWFile*—*RWFile* lets you save and restore objects to a file;
- *RWvostream*—Classes derived from *RWvostream*, such as *RWpostream*, *RWbostream*, and *RWeostream*, are used to save objects;
- *RWvistream*—Classes derived from *RWvistream*, such as *RWpostream*, *RWbistream*, and *RWeistream*, are used to restore objects.

To keep our explanations simple, we'll refer to all of these input and output classes as *streams*. For a discussion of the trade-offs in using *RWvostream* and *RWvistream* versus *RWFile*, see Chapter 6 and Chapter 7.

About the Examples in this Section

For your convenience, all examples listed in this section are provided on disk in the directory `rw/toolexam/manual`. Each of the examples in this chapter has the name `persist*.cpp`.

No Persistence

Some *Tools.h++* classes have no persistence. The *Class Reference* indicates "None" in the Persistence section for all such classes. Review the class reference entry for a particular class if you have a question about its level of persistence.

Simple Persistence

Simple persistence is the storage and retrieval of an object to and from a stream. Table 14-1 lists the classes in *Tools.h++* that use simple persistence.

Table 14-1 Classes with Simple Persistence

Category	Description
C++ fundamental types	`int`, `char`, `float`, ...
Rogue Wave date and time classes	*RWDate, RWTime*
Rogue Wave string classes	*RWCString, RWWString*
Miscellaneous Rogue Wave classes	*RWBitVec*

Because it is straightforward, simple persistence is a quick and easy way to save and restore objects that have neither pointers to other objects nor virtual member functions.

However, when objects that refer to each other are saved and then restored with simple persistence, the pointer relationships, or *morphology*, among the objects can change. This is because *simple persistence assumes that every pointer reference to an object in memory refers to a unique object*. Thus, when an object is saved with simple persistence, two references to the same memory location will cause two copies of the contents of that memory location to be saved. Not only does this use extra space in the stream, but it also causes the restored object to point to two distinct copies of the referenced object.

Two Examples of Simple Persistence

Let's look at a two examples of simple persistence. The first example illustrates successful persistence of fundamental datatypes, and demonstrates the *Tools.h++* overloaded operators `operator<<` and `operator>>`, which save and restore persistent objects. The second example illustrates one of the problems with simple persistence—its inability to maintain pointer relationships among objects.

Example One: Simple Persisting Objects of Fundamental Type

This example uses simple persistence to save two integers to an output stream `po`, which saves the integers to the file `int.dat`. Then the example restores the two integers from the stream `pi`, which reads the integers from the file `int.dat`.

The example uses the overloaded insertion operator `operator<<` to save the objects, and the overloaded extraction operator `operator>>` to restore the objects, much the same way as you use these operators to output and input objects in C++ streams.

Note that the saving stream and the restoring stream are put into separate blocks. This is so that opening `pi` will cause it to be positioned at the beginning of the file.

Here's the code:

```
#include <assert.h>
#include <fstream.h>
#include <rw/pstream.h>

main (){
  int j1 = 1;
  int k1 = 2;

  // Save integers to the file "int.dat"
    {
    // Open the stream to save to:
        ofstream          f("int.dat");
        RWpostream        po(f);

    // Use overloaded insertion operator
    // "RWpostream::operator<<(int)" to save integers:
        po << j1;
        po << k1;
    }

  // Restore integers from the file "int.dat"
    int j2 = 0;
    int k2 = 0;
    {
    // Open a separate stream to restore from:
        ifstream          f("int.dat");
        RWpistream        pi(f);

    // Use overloaded extraction operator
    // "RWpistream::operator>>(int)" to restore integers:
        pi >> j2;         // j1 == j2
        pi >> k2;         // k1 == k2
    }

  assert(j1 == j2);
  assert(k1 == k2);
  return 0;
}
```

The preceding example shows how easy it is to use overloaded operators to implement this level of persistence. So, what are some of the problems with using simple persistence? As mentioned above, one problem is that simple persistence will not maintain the pointer relationships among objects. We'll take a look at this problem in the next example.

Example Two: Simple Persistence and Pointers

This example shows one of the shortcomings of simple persistence: its inability to maintain the pointer relationships among persisted objects. Let's say that you have a class `Developer` that contains a pointer to other `Developer` objects:

```
Developer {
  public:
    Developer(const char* name, const Developer* anAlias = 0L)
      : name_(name), alias_(anAlias) {}

  RWCString        name_;  // Name of developer.
  const Developer* alias_;  // Alias points to another Developer.
};
```

Now let's say that you have another class, `Team`, that is an array of pointers to `Developer`s :

```
class Team {
  public:
    Developer*  member_[3];
};
```

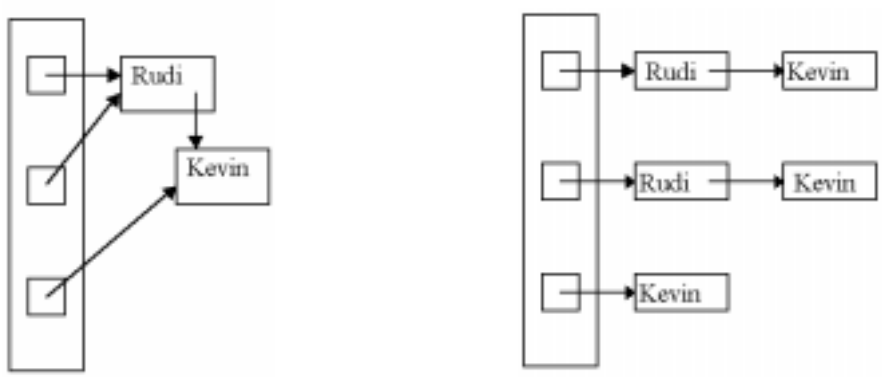
Note that `Team::member_` doesn't actually contain `Developer`s, but only pointers to `Developer`s .

We'll assume that you've written overloaded extraction and insertion operators that use simple persistence to save and restore `Developer`s and `Team`s . The example code for this is omitted to keep the explanation from getting cluttered.

When you save and restore a `Team` with simple persistence, what you restore may be different from what you saved. Let's look at the following code, which creates a team, then saves and restores it with simple persistence.

```
main (){
  Developer*  kevin   = new Developer("Kevin");
  Developer*  rudi    = new Developer("Rudi", kevin);
  Team        team1;

  team1.member_[0] = rudi;
  team1.member_[1] = rudi;
  team1.member_[2] = kevin;

  // Save with simple persistence:
  {
    RWFile    f("team.dat");
    f << team1;  // Simple persistence of team1.
  }

  // Restore with simple persistence:
  Team        team2;
  {
    RWFile            f("team.dat");
    f >> team2;
  }
  return 0;
}
```

Because this example uses simple persistence, which does not maintain pointer relationships, the restored `team` has different pointer relationships than the original `team`. Figure 14-1 shows what the created and restored `team`s look like in memory if you run the program.



Collection to be saved (`team1`). *Collection restored* (`team2`).

Figure 14-1 Simple Persistence

As you can see in Figure 14-1, when objects that refer to each other are saved and then are restored with simple persistence, the morphology among the objects can change. This is because simple persistence assumes that every pointer reference to an object in memory refers to a unique object. Thus, when such objects are saved, two references to the same memory location will cause two copies of the contents of that memory location to be saved, and later restored.

Isomorphic Persistence

Isomorphic persistence is the storage and retrieval of objects to and from a stream such that the pointer relationships between the objects are preserved. If there are no pointer relationships, isomorphic persistence effectively saves and restores objects the same way as simple persistence. When a collection is isomorphically persisted, all objects within that collection are assumed to have the same type. (A collection of objects of the same type is called a *homogeneous collection*.)

14

You can use isomorphic persistence to save and restore any *Tools.h++* class listed in Table 14-2. In addition, you can add isomorphic persistence to a class by following the technique described in Section , “Designing Your Class to Use Isomorphic Persistence,” on page 155.

Note that in this implementation of *Tools.h++*, isomorphic persistence of types templatized on pointers is not supported. For example, saving and restoring *RWTValDlist*<`int*`> is not supported.[1] In this case, we suggest that you use *RWTPtrDlist*<`int`> instead..

Table 14-2 Isomorphic Persistence Classes

Category	Description
Rogue Wave Standard C++ Library-based collection classes	*RWTValDeque, RWTPtrMap,...*
RWCollectable (Smalltalk-like) classes	*RWCollectableDate, RWCollectableString...* Note that *RWCollectable* classes also provide *polymorphic persistence* (see “Polymorphic Persistence” on page 174)
RWCollection classes that derive from *RWCollectable*	*RWBinaryTree, RWBag,...*
Rogue Wave *Tools.h++ 6.x* templatized collections	*RWTPtrDlist, RWTValDlist, RWTPtrSlist, RWTValSlist, RWTPtrOrderedVector, RWTValOrderedVector, RWTPtrSortedVector, RWTValSortedVector, RWTPtrVector, RWTValVector*

Isomorphic versus Simple Persistence

Let's look at a couple of illustrations that show the difference between isomorphic and simple persistence. In Figure 14-2, a collection of multiple pointers to the same object is saved to and restored from a stream, using simple persistence. Notice that when the collection is stored and restored in Figure 14-2, each pointer points to a distinct object. Contrast this with the

1. C++ template mechanisms prevent us from being able to do this. However, this restriction is probably a good thing—pointers saved at one location are likely to be troublesome indeed when injected into another.

isomorphic persistence of the same collection, shown in Figure 14-3, in which all of the restored pointers point to the same object, just as they did in the original collection.

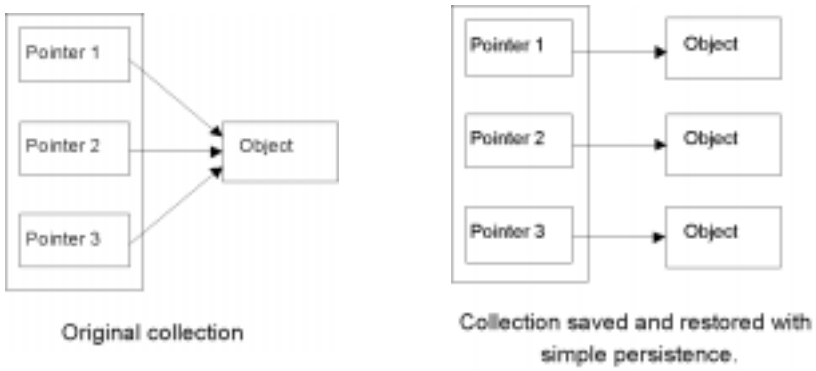


Figure 14-2 Saving and restoring with simple persistence.

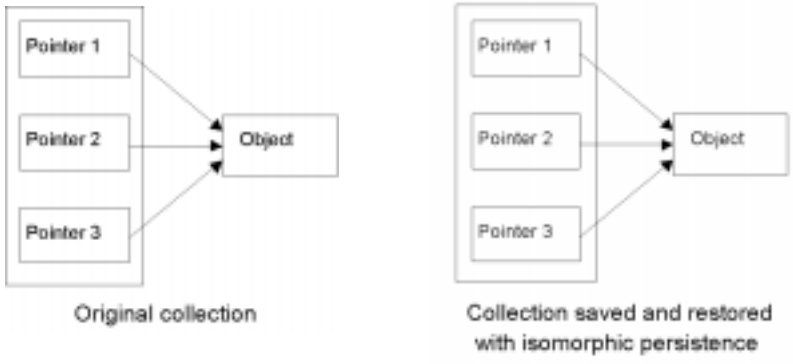


Figure 14-3 Saving and Restoring a Collection with Isomorphic Persistence

In Figure 14-4, we attempt to save and restore a circularly-linked list, using simple persistence. As shown in the figure, *any attempt to use simple persistence to save a circularly-linked list results in an infinite loop.*

The simple persistence mechanism creates a copy of each object that is pointed to and saves that object to a stream. But the simple persistence mechanism doesn't remember which objects it has already saved. When the simple persistence mechanism encounters a pointer, it has no way of knowing whether it has already saved that pointer's object. So in a circularly-linked list, the simple persistence mechanism saves the same objects over and over and over as the mechanism cycles through the list forever.

On the other hand, as shown in Figure 14-5, isomorphic persistence allows us to save the circularly-linked list. The isomorphic persistence mechanism uses a table to keep track of pointers it has saved. When the isomorphic persistence mechanism encounters a pointer to an unsaved object, it copies the object data, saves that object data—*not the pointer*—to the stream, then keeps track of the pointer in the *save table.* If the isomorphic persistence mechanism later encounters a pointer to the same object, instead of copying and saving the object data, the mechanism saves the save table's reference to the pointer.

When the isomorphic persistence mechanism restores pointers to objects from the stream, the mechanism uses a *restore table* to reverse the process. When the isomorphic persistence mechanism encounters a pointer to an unrestored object, it recreates the object with data from the stream, then changes the restored pointer to point to the recreated object. The mechanism keeps track of the pointer in the restore table. If the isomorphic persistence mechanism later

encounters a reference to an already-restored pointer, then the mechanism looks up the reference in the restore table, and updates the restored pointer to point to the object referred to in the table.

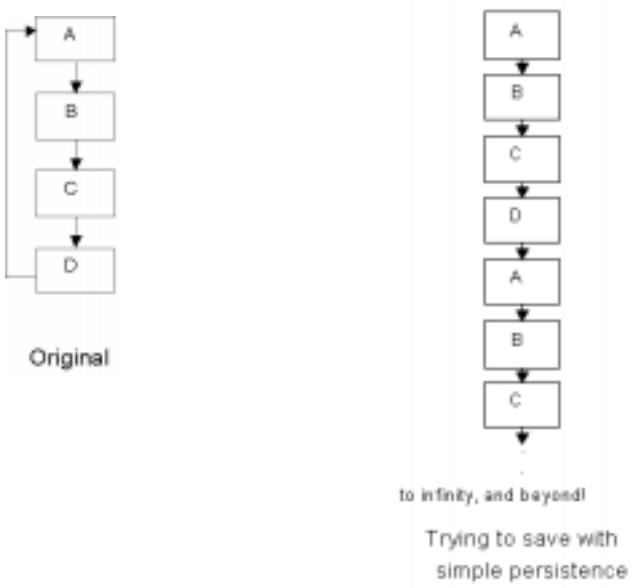


Figure 14-4 Attempt to Save and Restore a Circularly-linked List with Simple Persistence

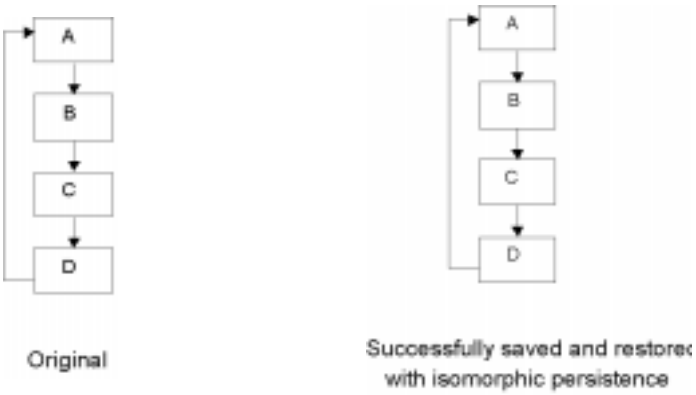


Figure 14-5 Saving and Restoring a Circularly-linked List with Isomorphic Persistence

Isomorphic Persistence of a Tools.h++ Class

The following example shows the isomorphic persistence of a templatized collection of *RWCollectable* integers, *RWTPtrDlist<RWCollectableInt>*. *RWTPtrDlist* is a templatized, reference-based, doubly-linked list that uses isomorphic persistence to store pointer references to values in memory.

This example uses *RWCollectableInt* instead of `int` because `int`s use simple persistence. By using *RWCollectableInt*s, we can implement isomorphic persistence.

When *RWTPtrDlist* is saved and then restored, the pointer relationships of the restored list will have the same morphology as the original list.

```
#include <assert.h>
#include <rw/tpdlist.h>  // RWTPtrDlist
#include <rw/collint.h>  // RWCollectableInt
#include <rw/rwfile.h>   // RWFile

main (){
  RWTPtrDlist<RWCollectableInt> dlist1;
  RWCollectableInt    *one = new RWCollectableInt(1);
  dlist1.insert(one);
  dlist1.insert(one);
  {
    RWFile            f("dlist.dat");
    f << dlist1;  // Isomorphic persistence of dlist1.
  }

  assert(dlist1[0] == one && dlist1[0] == dlist1[1]);
    // dlist1[0], dlist[1] and "one" all point to the
    // same place.

  RWTPtrDlist<RWCollectableInt> dlist2;
  {
    RWFile            f("dlist.dat");
    f >> dlist2;
      // restore dlist2 from f
      // dlist2 now contains 2 pointers
      // to the same RWCollectableInt of value 1.
      // However, this RWCollectableInt isn't at
```

```
      // the same address as the value
      // that "one" points to.
  }

  // See Figure 14-6 below to see what dlist1 and dlist2
  // now look like in memory.
  assert(dlist2[0] == dlist2[1] && (*dlist2[0]) == *one);
    // dlist2[0] and dlist2[1] point to the same place
    // and that place has the same value as "one".
  delete dlist2[0];
  delete one;
    // The developer must allocate and delete objects.
    // The templatized collection member function
    // clearAndDestroy() doesn't check that a given
    // pointer is deleted only once.
    // So in this case, delete the shared
    // pointer manually.

  return 0;
}
```

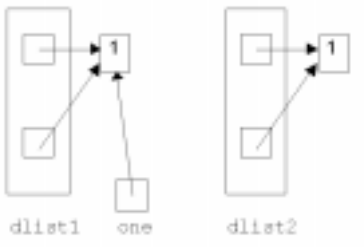


Figure 14-6 After Isomorphic Save and Restore of RWTPtrDlist<RWCollectableInt>

Designing Your Class to Use Isomorphic Persistence

Table 14-2 lists the *Tools.h++* classes that implement isomorphic persistence. You can also *add* isomorphic persistence to an existing class, even if you only have the header files for that class. Before you can add isomorphic persistence to a class, it must meet the following requirements:

- Class `T` must have appropriate default and copy constructors defined or generated by the compiler:

```
T();                    // default constructor
T(T& t);               // copy constructor
```

- Class `T` must have an assignment operator defined as a member or as a global function:

```
T& operator=(const T& t);             // member function
T& operator=(T& lhs, const T& rhs); // global function
```

- Class `T` *cannot* have any non-type template parameters. For example, in *RWTBitVec*<size>, "size" is placeholder for a value rather than a type. No present compiler accepts function templates with non-type template parameters, and the global functions used to implement isomorphic persistence (`rwRestoreGuts` and `RWSaveGuts`) are function templates when they are used to persist templatized classes.
- Class `T` must use the macros `RW_DECLARE_PERSISTABLE` and `RW_DEFINE_PERSISTABLE` or their equivalents. More about this in “Add RWDECLARE_PERSISTABLE to Your Header File” on page 158 and “Add RWDEFINE_PERSISTABLE to One Source File” on page 160.
- All the data necessary to recreate an instance of Class `T` must be globally available (have accessor functions). If you can't make this data available, you can't implement isomorphic persistence. More about this in “Make All Necessary Class Data Available” on page 157.

If your class `T` will be stored in a Standard C++ Library container or a Standard C++ Library-based collection, you may need to implement `operator<(const T&, const T&)` and `operator==(const T&, const T&)`. See “Migration Guide: For Users of Previous Versions of Tools.h++” on page 112 for more information.

To create an isomorphically persistent class or to add isomorphic persistence to an existing class, follow these steps:

1. Make all necessary class data available.
2. Add `RWDECLARE_PERSISTABLE` to your header file.
3. Add `RWDEFINE_PERSISTABLE` to one source file.
4. Check for possible problems.
5. Define `rwSaveGuts` and `rwRestoreGuts`.

Make All Necessary Class Data Available

All class data that will be isomorphically persisted must be accessible to the global functions, `rwSaveGuts` and `rwRestoreGuts`, used to implement persistence for the class.

Note that only the information necessary to recreate an object of that class must be accessible to `rwSaveGuts` and `rwRestoreGuts`. Other data can be kept protected or private.

There are several ways to make protected and private data members of classes accessible.

First, your class could make friends with `rwSaveGuts` and `rwRestoreGuts`:

```
class Friendly {
// These global functions access private members.
  friend void rwSaveGuts(RWvostream&, const Friendly&);
  friend void rwRestoreGuts(RWFile&, Friendly&);
  friend void rwSaveGuts(RWFile&, const Friendly&);
  friend void rwRestoreGuts(RWvistream&, Friendly&);
//...
};
```

Or your class could have accessor functions to the restricted but necessary members:

```
class Accessible {
public:
   int  secret(){return secret_}
   void secret(const int s){secret_ = s}
//...
private:
  int   secret_;
};
```

If you can't change the source code for the class to which you want to add isomorphic persistence, then you could consider deriving a new class that provides access via public methods or friendship:

```
class Unfriendly{
protected:
   int secret_;
// ...
};

class Friendlier : public Unfriendly {
```

```
public:
   int  secret(){return secret_}
   void secret(const int s){secret_ = s}
//...
};
```

If you can't change the source code for a class, you will be unable to isomorphically persist private members of that class. But remember: *you only need access to the data necessary to recreate the class object*, not to *all* the members of the class. For example, if your class has a private cache that is created at run time, you probably don't need to save and restore the cache. Thus, even though that cache is private, you don't need access to it in order to persist the class object.

Add RWDECLARE_PERSISTABLE to Your Header File

Once you have determined that all necessary class data is accessible, you must add declaration statements to your header files. These statements declare the global functions `operator<<` and `operator>>` for your class. The global functions permit storage to and retrieval from *RWvistream*, *RWvostream* and *RWFile*.

Tools.h++ provides several macros that make adding these declarations easy. The macro you choose depends upon whether your class is templatized or not, and if it is templatized, how many templatized parameters it has.

- *For non-templatized classes*, use `RWDECLARE_PERSISTABLE`.

`RWDECLARE_PERSISTABLE` is a macro found in `rw/edefs.h`. To use it, add the following lines to your header file (`*.h`):

```
#include <rw/edefs.h>
RWDECLARE_PERSISTABLE(YourClass)
```

`RWDECLARE_PERSISTABLE(YourClass)` will expand to declare the following global functions:

```
RWvostream& operator<<(RWvostream& strm, const YourClass& item);
RWvistream& operator>>(RWvistream& strm, YourClass& obj);
RWvistream& operator>>(RWvistream& strm, YourClass*& pObj);

RWFile& operator<<(RWFile& strm, const YourClass& item);
RWFile& operator>>(RWFile& strm, YourClass& obj);
RWFile& operator>>(RWFile& strm, YourClass*& pObj);
```

- *For templatized classes with a single template parameter* `T`, use the macro `RWDECLARE_PERSISTABLE_TEMPLATE`.

`RWDECLARE_PERSISTABLE_TEMPLATE` is also found in `rw/edefs.h`. To use it, add the following lines to your header file (`*.h`):

```
#include <rw/edefs.h>
RWDECLARE_PERSISTABLE_TEMPLATE(YourClass)
```

`RWDECLARE_PERSISTABLE_TEMPLATE(YourClass)` will expand to declare the following global functions:

```
template<class T>
RWvostream& operator<<
(RWvostream& strm, const YourClass<T>& item);

template<class T>
RWvistream& operator>>
(RWvistream& strm, YourClass<T>& obj);

template<class T>
RWvistream& operator>>
(RWvistream& strm, YourClass<T>*& pObj);

template<class T>
RWFile& operator<<(RWFile& strm, const YourClass<T>& item);

template<class T>
RWFile& operator>>(RWFile& strm, YourClass<T>& obj);

template<class T>
RWFile& operator>>(RWFile& strm, YourClass<T>*& pObj);
```

- *For templatized classes with more than one and less than five template parameters*, use one of the following macros from `rw/edefs.h`.:

```
// For YourClass<T1,T2>:
RWDECLARE_PERSISTABLE_TEMPLATE_2(YourClass)

// For YourClass<T1,T2,T3>:
RWDECLARE_PERSISTABLE_TEMPLATE_3(YourClass)

// For YourClass<T1,T2,T3,T4>:
RWDECLARE_PERSISTABLE_TEMPLATE_4(YourClass)
```

Note – Remember, if your templatized class has any non-type template parameters, it cannot be isomorphically persisted.

- *If you need to persist templatized classes with five or more template parameters,* you can write additional macros for `RWDECLARE_PERSISTABLE_TEMPLATE_n`. The macros are found in the header file `rw/edefs.h`.

Add RWDEFINE_PERSISTABLE to One Source File

After you have declared the global storage and retrieval operators, you must define them. *Tools.h++* provides macros that add code to your source file[1] to define the global functions `operator<<` and `operator>>` *f*or storage to and retrieval from *RWvistream*, *RWvostream*, and *RWFile*. `RWDEFINE_PERSISTABLE` macros will automatically create global `operator<<` and `operator>>` functions that perform isomorphic persistence duties and call the global persistence functions `rwSaveGuts` and `rwRestoreGuts` for your class. More about `rwSaveGuts` and `rwRestoreGuts` later.

Again, your choice of which macro to use is determined by whether your class is templatized, and if so, how many parameters it requires.

- *For non-templatized classes*, use `RWDEFINE_PERSISTABLE`.

`RWDEFINE_PERSISTABLE` is a macro found in `rw/epersist.h`. To use it, add the following lines to one *and only one* source file (`*.cpp or *.C`):

```
#include <rw/epersist.h>
RWDEFINE_PERSISTABLE(YourClass)
```

`RWDEFINE_PERSISTABLE(YourClass)` will expand to generate the source code for (that is, to define) the following global functions:

```
RWvostream& operator<<(RWvostream& strm, const YourClass& item)
RWvistream& operator>>(RWvistream& strm, YourClass& obj)
RWvistream& operator>>(RWvistream& strm, YourClass*& pObj)

RWFile& operator<<(RWFile& strm, const YourClass& item)
RWFile& operator>>(RWFile& strm, YourClass& obj)
RWFile& operator>>(RWFile& strm, YourClass*& pObj)
```

- *For templatized classes with a single template parameter* `T`, use `RWDEFINE_PERSISTABLE_TEMPLATE`.

`RWDEFINE_PERSISTABLE_TEMPLATE` is also found in `rw/epersist.h`. To use it, add the following lines to one *and only one* source file (`*.cpp or *.C`):

1. You may find for template classes that with some compilers the source file must have the same base name as the header file where `RWDECLARE_PERSISTABLE` was used.

```
#include <rw/epersist.h>
RWDEFINE_PERSISTABLE_TEMPLATE(YourClass)
```

`RWDEFINE_PERSISTABLE_TEMPLATE(YourClass)` will expand to generate the source code for the following global functions:

```
template<class T>
RWvostream& operator<<
(RWvostream& strm, const YourClass<T>& item)

template<class T>
RWvistream& operator>>
(RWvistream& strm, YourClass<T>& obj)

template<class T>
RWvistream& operator>>
(RWvistream& strm, YourClass<T>*& pObj)

template<class T>
RWFile& operator<<(RWFile& strm, const YourClass<T>& item)

template<class T>
RWFile& operator>>(RWFile& strm, YourClass<T>& obj)

template<class T>
RWFile& operator>>(RWFile& strm, YourClass<T>*& pObj)
```

- *For templatized classes with more than one and less than five template parameters,* use one of the following macros from `rw/epersist.h`:

```
// For YourClass<T1,T2>:
RWDEFINE_PERSISTABLE_TEMPLATE_2(YourClass)

// For YourClass<T1,T2,T3>:
RWDEFINE_PERSISTABLE_TEMPLATE_3(YourClass)

// For YourClass<T1,T2,T3,T4>:
RWDEFINE_PERSISTABLE_TEMPLATE_4(YourClass)
```

Note – Remember, if your templatized class has any non-type template parameters, it cannot be isomorphically persisted.

- *If you need to persist templatized classes with five or more template parameters,* you can write additional macros for `RWDEFINE_PERSISTABLE_TEMPLATE_n`. The macros are found in the header file `rw/epersist.h`.

Check for Possible Problems

You've made the necessary data accessible, and declared and defined the global functions required for isomorphic persistence. Before you go any further, you need to review your work for two possible problems.

1. You can't use the `RWDECLARE_PERSISTABLE_TEMPLATE` and `RWDEFINE_PERSISTABLE_TEMPLATE` macros to persist any templatized class that has *non-type template parameters*. Templates with non-type template parameters, such as *RWTBitVec<size>*, cannot be isomorphically persisted.

2. If you have defined any of the following global operators and you use the `RWDEFINE_PERSISTABLE` macro, you will get compiler ambiguity errors.

   ```
   RWvostream& operator<<(RWvostream& s, const YourClass& t);
   RWvistream& operator>>(RWvistream& s, YourClass& t);
   RWvistream& operator>>(RWvistream& s, YourClass*& pT);

   RWFile& operator<<(RWFile& s, const YourClass& t);
   RWFile& operator>>(RWFile& s, YourClass& t);
   RWFile& operator>>(RWFile& s, YourClass*& pT);
   ```

   The compiler errors occur because using `RWDEFINE_PERSISTABLE` along with a different definition of the operators defines the operators twice. This means that the compiler does not know which operator definition to use. In this case, you have two choices:

- Remove the `operator<<` and `operator>>` global functions that you previously defined for `YourClass` and replace them with the operators generated by the `RWDEFINE_PERSISTABLE(YourClass).`
- Modify your `operator<<` and `operator>>` global functions for `YourClass` using the contents of the `RWDEFINE_PERSISTABLE` macro in `rw/epersist.h` as a guide.

Define rwSaveGuts and rwRestoreGuts

Now you must add to one and only one source file the global functions `rwSaveGutsr` and `rwRestoreGuts,` which will be used to save and restore the internal state of your class. These functions are called by the `operator<<` and `operator>>` that were declared and defined as discussed in “Add RWDECLARE_PERSISTABLE to Your Header File” on page 158 and “Add RWDEFINE_PERSISTABLE to One Source File” on page 160.

Note – "Writing rwSaveGuts and rwRestoreGuts Functions" on page 164 provides guidelines about how to write `rwSaveGuts` and `rwRestoreGuts` global functions.

- *For non-templatized classes* , define the following functions:

```
void rwSaveGuts(RWFile& f, const YourClass& t){/*...*/}
void rwSaveGuts(RWvostream& s, const YourClass& t) {/*...*/}
void rwRestoreGuts(RWFile& f, YourClass& t) {/*...*/}
void rwRestoreGuts(RWvistream& s, YourClass& t) {/*...*/}
```

- *For templatized classes with a single template parameter* `T`, define the following functions:

```
template<class T> void
rwSaveGuts(RWFile& f, const YourClass<T>& t){/*...*/}
template<class T> void
rwSaveGuts(RWvostream& s, const YourClass<T>& t) {/*...*/}
template<class T> void
rwRestoreGuts(RWFile& f, YourClass<T>& t) {/*...*/}
template<class T>void
rwRestoreGuts(RWvistream& s, YourClass<T>& t) {/*...*/}
```

- *For templatized classes with more than one template parameter*, define `rwRestoreGuts` and `rwSaveGuts` with the appropriate number of template parameters.

Function `rwSaveGuts` saves the state of each class member necessary for persistence to an *RWvostream* or an *RWFile*. If the members of your class can be persisted (see Table 14-2 above), and if the necessary class members are accessible to `rwSaveGuts`, you can use `operator<<` to save the class members.

Function `rwRestoreGuts` restores the state of each class member necessary for persistence from an *RWvistream* or an *RWFile*. Provided that the members of your class are types that can be persisted, and provided that the members of your class are accessible to `rwRestoreGuts`, you can use `operator>>` to restore the class members.

Writing rwSaveGuts and rwRestoreGuts Functions

The next two sections discuss guidelines for writing `rwSaveGuts` and `rwRestoreGuts` global functions. To illustrate these guidelines, the following class will be used:

```
class Gut {
  public:
    int                    fundamentalType_;
    RWCString              aRogueWaveObject_;
    RWTValDlist            anotherRogueWaveObject_;
    RWCollectableString    anRWCollectable_
    RWCollectableString*   pointerToAnRWCollectable_;
    Gut*                   pointerToAnObject_;
};
```

The discussion in the next two sections describes how to write `rwSaveGuts` and `rwRestoreGuts` functions for non-templatized classes. However, the descriptions also apply to the templatized `rwSaveGuts` and `rwRestoreGuts` that are written for templatized classes.

Guidelines for Writing rwSaveGuts

The global overloaded functions:

```
rwSaveGuts(RWFile& f, const YourClass& t)
rwSaveGuts (RWvostream& s, const YourClass& t)
```

are responsible for saving the internal state of a `YourClass` object to either a binary file (using class *RWFile*) or to a virtual output stream (an *RWvostream*). This allows the object to be restored at some later time.

The `rwSaveGuts` functions that you write must save the state of each member in `YourClass`, *including the members of the class that you inherited from*.

How you write the functions depends upon the type of the member data:

- To save member data that are either C++ fundamental types (`int`, `char`, `float`,...), or most Rogue Wave classes, including *RWCollectable*, use the overloaded insertion operators (`operator<<`).
- Saving members that are pointers to non-*RWCollectable* objects can be a bit tricky. This is because it is possible that a pointer does not point to any object at all. One way of dealing with the possibility of nil pointers is to

check whether a pointer points to a valid object. If the pointer is valid, save a Boolean `true`, then save the dereferenced pointer. If the pointer is invalid, save a Boolean `false` but don't save the pointer.

When you restore the pointer, `rwRestoreGuts` first restores the Boolean. If the Boolean is `true`, then `rwRestoreGuts` restores the valid pointer. If the Boolean is `false`, then `rwRestoreGuts` sets the pointer to nil.

- Saving pointers to objects derived from *RWCollectable* is easier. It is still possible that a pointer is nil. But if you use:

  ```
  RWvostream& operator<<(RWvostream&, const RWCollectable*);
  ```

  to save the pointer, the nil pointer will be detected automatically.

Using these guidelines, you can write `rwSaveGuts` functions for the example class `Gut` as follows:

```
void rwSaveGuts(RWvostream& stream, const Gut& gut) {

  // Use insertion operators to save fundamental objects,
  // Rogue Wave objects and pointers to
  // RWCollectable-derived objects.

  stream
    << gut.fundamentalType_
    << gut.aRogueWaveObject_
    << gut.anotherRogueWaveObject_
    << gut.pointerToAnRWCollectable_;

  // The tricky saving of a pointer
  // to a non-RWCollectable object.

  if (gut.pointerToAnObject_ == 0)  // Is it a nil pointer?
    stream << false;                    // Yes, don't save.
  else {
    stream << true;                       // No, it's valid
    stream << *(gut.pointerToAnObject_);  // so save it.
  }
}

void rwSaveGuts(RWFile& stream, const Gut& gut) {
  // The body of this function is identical to
  // rwSaveGuts(RWvostream& stream, const Gut& gut).
}
```

Guidelines for Writing rwRestoreGuts

The global overloaded functions:

```
rwRestoreGuts(RWFile& f, YourClass& t)
rwRestoreGuts(RWvostream& s, YourClass& t)
```

are responsible for restoring the internal state of a `YourClass` object from either a binary file (using class *RWFile*) or from a virtual output stream (an *RWvostream*).

The `rwRestoreGuts` functions that you write must restore the state of each member in `YourClass`, including *the members of the class that you inherited from*. The functions must restore member data in the order that it was saved.

How you write the functions depends upon the type of the member data:

- To restore member data that are either C++ fundamental types (`int`, `char`, `float`,...) or most Rogue Wave classes, including *RWCollectable*, use the overloaded extraction operators (`operator>>`).
- Restoring members that are pointers to non-*RWCollectable* objects can be a bit tricky. This is because it is possible that a saved pointer did not point to any object at all. But if `rwSaveGuts` saved a Boolean flag before saving the pointer, as we described in the previous section, then it is a relatively simple matter for the `rwRestoreGuts` to restore valid and nil pointers.

  Assuming that the members were saved with a compatible `rwSaveGuts`, when you restore the pointer, `rwRestoreGuts` first restores the Boolean. If the Boolean is `true`, then `rwRestoreGuts` restores the valid pointer. If the Boolean is `false`, then `rwRestoreGuts` sets the pointer to nil.
- Restoring pointers to objects derived from *RWCollectable* is easier. It is still possible that the pointer is nil. But if you use:

  ```
  RWvostream& operator>>(RWvostream&, const RWCollectable*&);
  ```

  to restore the pointer, the nil pointer will be detected automatically.

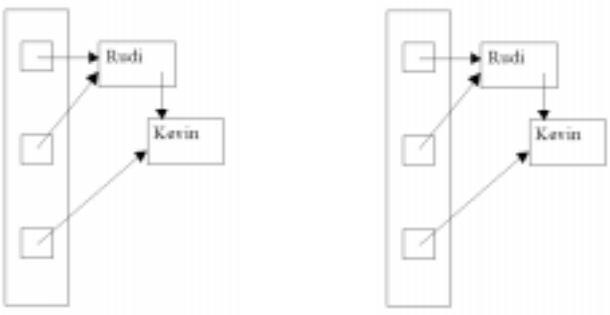
Using these guidelines, you can write the `rwRestoreGuts` functions for the example class `Gut` as follows:

```
void rwRestoreGuts(RWvistream& stream, const Gut& gut) {

  // Use extraction operators to restore fundamental objects,
  // Rogue Wave objects and pointers to
  // RWCollectable-derived objects.

  stream
    >> gut.fundamentalType_
    >> gut.aRogueWaveObject_
    >> gut.anotherRogueWaveObject_
    >> gut.pointerToAnRWCollectable_;

  // The tricky restoring of a pointer
  // to a non-RWCollectable object.

  bool isValid;
  stream >> isValid;                  // Is it a nil pointer?

  if (isValid)                        // No,
    stream >> gut.pointerToAnObject_; // restore the pointer.
  else                                // Yes,
    gut.pointerToAnObject_ = rwnil;   // set pointer to nil.
}

void rwRestoreGuts(RWFile& stream, Gut& gut) {
  // The body of this function is identical to
  // rwRestoreGuts(RWvostream& stream, Gut& gut).
}
```

Isomorphic Persistence of a User-designed Class

“Example Two: Simple Persistence and Pointers” on page 147 described some example code that implements simple persistence on a collection that includes pointers. That example illustrated how simple persistence does not maintain the original collection's morphology.

This example implements isomorphic persistence on the collection we set up in “Example Two: Simple Persistence and Pointers” on page 147: `Team`, which contains three `Developer`s. Figure 14-7 shows the morphology of the original `Team` collection and of the `Team` collection after we saved and restored it with isomorphic persistence.



Collection to be saved (`team1`). *Collection restored* (`team2`).

Figure 14-7 Isomorphic Persistence

As you read the code, notice how the `Developer::alias_` member, which points to other `Developer`s, is saved and restored. You'll find that after saving `Developer::name_` , the `rwSaveGuts` function for `Developer` checks to see if `alias_` is pointing to a `Developer` in memory. If not, `rwSaveGuts` stores a Boolean `false` to signify that `alias_` is a nil pointer. If `alias_` *is* pointing to a `Developer`, `rwSaveGuts` stores a Boolean `true`. It is only afterwards that `rwSaveGuts` finally stores the *value* of the `Developer` that `alias_` is pointing to.

This code can distinguish between new `Developer`s and existing `Developer`s because the insertion operators generated by `RWDEFINE_PERSISTABLE(Developer)` keep track of `Developer`s that have been stored previously. The insertion operator, `operator<<`, calls the `rwSaveGuts` if and only if a `Developer` has not yet been stored in the stream by `operator<<`.

When a `Developer` object is restored, the extraction operator, `operator>>`, for `Developer` is called. Like the insertion operators, the extraction operators are generated by `RWDEFINE_PERSISTABLE(Developer)`. If a `Developer` object has already been restored, then the extraction operator will adjust the

`Developer::alias_` pointer so that it points to the already existing `Developer`. If the `Developer` has not yet been restored, then `rwRestoreGuts` for `Developer` will be called.

After restoring `Developer::name_`, `rwRestoreGuts` for `Developer` restores a Boolean value to determine whether `Developer::alias_` should point to a `Developer` in memory or not. If the Boolean is `true`, then `alias_` should point to a `Developer`, so `rwRestoreGuts` restores the `Developer` object. Then `rwRestoreGuts` updates `alias_` to point to the restored `Developer`.

The isomorphic persistence storage and retrieval process described above for `Developer.alias_` can also be applied to the `Developer` pointers in `Team`.

Here is the code:

```
#include <iostream.h>   // For user output.
#include <assert.h>
#include <rw/cstring.h>
#include <rw/rwfile.h>
#include <rw/epersist.h>
//------------------ Declarations --------------------
//------------------- Developer ----------------------
class Developer {
  public:
    Developer
      (const char* name = "", Developer* anAlias = rwnil)
      : name_(name), alias_(anAlias) {}
  RWCString         name_;
  Developer*        alias_;
};
#include <rw/edefs.h>
RWDECLARE_PERSISTABLE(Developer)
//--------------------- Team -------------------------
class Team {
  public:
    Developer*  member_[3];
};
RWDECLARE_PERSISTABLE(Team);
```

```
//---------- rwSaveGuts and rwRestoreGuts -------------
//------------------- Developer ----------------------
RWDEFINE_PERSISTABLE(Developer)
// This macro generates the following insertion and extraction
// operators:
//   RWvostream& operator<<
//     (RWvostream& strm, const Developer& item)
//   RWvistream& operator>>(RWvistream& strm, Developer& obj)
//   RWvistream& operator>>(RWvistream& strm, Developer*& pObj)
//   RWFile& operator<<(RWFile& strm, const Developer& item)
//   RWFile& operator>>(RWFile& strm, Developer& obj)
//   RWFile& operator>>(RWFile& strm, Developer*& pObj)
void rwSaveGuts(RWFile& file, const Developer& developer){
// Called by:
//    RWFile& operator<<(RWFile& strm, const Developer& item)
  file << developer.name_;        // Save name.
  // See if alias_ is pointing to a Developer in memory.
  // If not, then rwSaveGuts stores a boolean false to signify
  // that alias_ is a nil pointer.
  // If alias_ is pointing to a Developer,
  // then rwSaveGuts stores a boolean true
  // and stores the value of the Developer
  // that alias_ is pointing to.
  if (developer.alias_ == rwnil)
    file << false;               // No alias.
  else {
    file << true;
    file << *(developer.alias_);  // Save alias.
  }
}

void rwSaveGuts(RWvostream& stream, const Developer& developer) {
// Called by:
//    RWvostream& operator<<
//      (RWvostream& strm, const Developer& item)
  stream << developer.name_;        // Save name.
  // See if alias_ is pointing to a Developer in memory.
  if (developer.alias_ == rwnil)
    stream << false;               // No alias.
  else {
    stream << true;
    stream << *(developer.alias_);  // Save alias.
  }
}
```

```
void rwRestoreGuts(RWFile& file, Developer& developer) {
//  Called by:
//    RWFile& operator>>(RWFile& strm, Developer& obj)
  file >> developer.name_;        // Restore name.
  // Should developer.alias_ point to a Developer?
  RWBoolean alias;
  file >> alias;
  // If alias_ should point to a Developer,
  // then rwRestoreGuts restores the Developer object
  // and then updates alias_ to point to the new Developer.
  if (alias)                      // Yes.
     file >> developer.alias_;
        // Call:
        //   RWFile& operator>>(RWFile& strm, Developer*& pObj)
}
void rwRestoreGuts(RWvistream& stream, Developer& developer) {
// Called by:
//   RWvistream& operator>>(RWvistream& strm, Developer& obj)

  stream >> developer.name_;     // Restore name.

  // Should developer.alias_ point to a Developer?
  RWBoolean alias;
  stream >> alias;
  if (alias)                    // Yes.
    stream >> developer.alias_;
       // Restore alias and update pointer.
       // Calls:
       //    RWvistream& operator>>
       //      (RWvistream& strm, Developer*& pObj)

}
// For user output only:
ostream& operator<<(ostream& stream, const Developer& d) {
  stream << d.name_
    << " at memory address: " << (void*)&d;
  if (d.alias_)
    stream << " has an alias at memory address: "
      << (void*)d.alias_ << " ";
  else
    stream << " has no alias.";
  return stream;
}
```

```
//---------------------- Team -----------------------------
RWDEFINE_PERSISTABLE(Team);
// This macro generates the following insertion and extraction
// operators:
//    RWvostream& operator<<
//      (RWvostream& strm, const Team& item)
//    RWvistream& operator>>(RWvistream& strm, Team& obj)
//    RWvistream& operator>>(RWvistream& strm, Team*& pObj)
//    RWFile& operator<<(RWFile& strm, const Team& item)
//    RWFile& operator>>(RWFile& strm, Team& obj)
//    RWFile& operator>>(RWFile& strm, Team*& pObj)
void rwSaveGuts(RWFile& file, const Team& team){
// Called by RWFile& operator<<(RWFile& strm, const Team& item)

  for (int i = 0; i < 3; i++)
    file << *(team.member_[i]);
      // Save Developer value.
      // Call:
      //   RWFile& operator<<
      //     (RWFile& strm, const Developer& item)
}
void rwSaveGuts(RWvostream& stream, const Team& team) {
// Called by:
//    RWvostream& operator<<(RWvostream& strm, const Team& item)
  for (int i = 0; i < 3; i++)
    stream << *(team.member_[i]);
      // Save Developer value.
      // Call:
      //   RWvostream& operator<<
      //     (RWvostream& strm, const Developer& item)
}
void rwRestoreGuts(RWFile& file, Team& team) {
// Called by RWFile& operator>>(RWFile& strm, Team& obj)

  for (int i = 0; i < 3; i++)
    file >> team.member_[i];
      // Restore Developer and update pointer.
      // Call:
      //    RWFile& operator>>(RWFile& strm, Developer*& pObj)
}
void rwRestoreGuts(RWvistream& stream, Team& team) {
// Called by:
```

```
//    RWvistream& operator>>(RWvistream& strm, Team& obj)
  for (int i = 0; i < 3; i++)
    stream >> team.member_[i];
       // Restore Developer and update pointer.
       // Call:
       //    RWvistream& operator>>
       //      (RWvistream& strm, Developer*& pObj)
}
// For user output only:

ostream& operator<<(ostream& stream, const Team& t) {
  for (int i = 0; i < 3; i++)
    stream << "[" << i << "]:" << *(t.member_[i]) << endl;
  return stream;
}

//-------------------- main -------------------------
main (){
  Developer*  kevin   = new Developer("Kevin");
  Developer*  rudi    = new Developer("Rudi", kevin);
  Team        team1;

  team1.member_[0] = rudi;
  team1.member_[1] = rudi;
  team1.member_[2] = kevin;
  cout << "team1 (before save):" << endl
    << team1 << endl << endl;   // Output to user.
  {
    RWFile    f("team.dat");
    f << team1;  // Isomorphic persistence of team.
  }

  Team        team2;
  {
    RWFile            f("team.dat");
    f >> team2;
  }
  cout << "team2 (after restore):" << endl
    << team2 << endl << endl;   // Output to user.

  delete kevin;
  delete rudi;
  return 0;
}
```

Output:

```
team1 (before save):
[0]:Rudi at memory address: 0x10002be0
    has an alias at memory address: 0x10002bd0
[1]:Rudi at memory address: 0x10002be0
    has an alias at memory address: 0x10002bd0
[2]:Kevin at memory address: 0x10002bd0 has no alias.


team2 (after restore):
[0]:Rudi at memory address: 0x10002c00
    has an alias at memory address: 0x10002c10
[1]:Rudi at memory address: 0x10002c00
    has an alias at memory address: 0x10002c10
[2]:Kevin at memory address: 0x10002c10 has no alias.
```

Polymorphic Persistence

Polymorphic persistence preserves pointer relationships (or morphology) among persisted objects, and also allows the restoring process to restore an object without prior knowledge of that object's type.

Tools.h++ uses classes derived from *RWCollectable* to do polymorphic persistence. The objects created from those classes may be any of the different types derived from *RWCollectable*. A group of such objects, where the objects may have different types, is called a *heterogeneous collection*.

Table 14-3 lists the classes that use polymorphic persistence.

Table 14-3 Polymorphic Persistence Classes

Category	Description
RWCollectable (Smalltalk-like) classes	*RWCollectableDate*, *RWCollectableString*...
RWCollection classes (which derive from *RWCollectable*)	*RWBinaryTree*, *RWBag*...

Operators

The storage and retrieval of polymorphic objects that inherit from *RWCollectable* is a powerful and adaptable feature of the *Tools.h++* class library. Like other persistence mechanisms, polymorphic persistence uses the overloaded extraction and insertion operators (`operator<<` and `operator>>`polymorphic persistence). When these operators are used in polymorphic persistence, not only are objects isomorphically saved and restored, but objects of unknown type can be restored.

Polymorphic persistence uses the operators listed below.

- *Operators that save references to RWCollectable objects*:

```
Rwvostream&  operator<<(RWvostream&, const RWCollectable&);
RWFile&      operator<<(RWFile&,     const RWCollectable&);
```

  Each *RWCollectable*-derived object is saved isomorphically with a class ID that uniquely identifies the object's class.

- *Operators that save RWCollectable pointers:*

```
Rwvostream&  operator<<(RWvostream&, const RWCollectable*);
RWFile&      operator<<(RWFile&,     const RWCollectable*);
```

  Each pointer to an object is saved isomorphically with a class ID that uniquely identifies the object's class. Even nil pointers can be saved.

- *Operators that restore already-existing RWCollectable objects:*

```
Rwvistream&  operator>>(RWvistream&, RWCollectable&);
RWFile&      operator>>(RWFile&,     RWCollectable&);
```

  Each *RWCollectable*-derived object is restored isomorphically. The persistence mechanism determines the object type at run time by examining the class ID that was stored with the object.

- *Operators that restore pointers to RWCollectable objects:*

```
Rwvistream&  operator>>(RWvistream&, RWCollectable*&);
RWFile&      operator>>(RWFile&,     RWCollectable*&);
```

  Each object derived from *RWCollectable* is restored isomorphically and the pointer reference is updated to point to the restored object. The persistence mechanism determines the object type at run time by examining the class ID that was stored with the object. *Since the restored objects are allocated from the heap, you are responsible for deleting them when you are done with them.*

Designing your Class to Use Polymorphic Persistence

Note that the ability to restore the pointer relationships of a polymorphic object is a property of the base class, *RWCollectable*. Polymorphic persistence can be used by *any* object that inherits from *RWCollectable* —including your own classes. Chapter 15 describes how to implement polymorphic persistence in the classes that you create by inheriting from *RWCollectable*.

Polymorphic Persistence Example

This example of polymorphic persistence contains two distinct programs. The first example polymorphically *saves* the contents of a collection to standard output (`stdout`). The second example polymorphically *restores* the contents of the saved collection from standard input (`stdin`). We divided the example to demonstrate that you can use persistence to share objects between two different processes.

If you compile and run the first example, the output is an object as it would be stored to a file. However, you can pipe the output of the first example into the second example:

```
firstExample | secondExample
```

Example One: Saving Polymorphically

This example constructs an empty collection, inserts objects into that collection, then saves the collection polymorphically to standard output.

Notice that example one creates and saves a collection that includes two copies of the same object and two other objects. The four objects have three different types. When example one saves the collection and when example two restores the collection, we see that:

- The morphology of the collection is maintained;
- The process that restores the collection does not know the object's type before it restores that object.

Here's the first example:

```
#include <rw/ordcltn.h>
#include <rw/collstr.h>
#include <rw/collint.h>
#include <rw/colldate.h>
#include <rw/pstream.h>

main(){
   // Construct an empty collection
   RWOrdered collection;

   // Insert objects into the collection.

   RWCollectableString* george;
   george = new RWCollectableString("George");

   collection.insert(george);     // Add the string once
   collection.insert(george);     // Add the string twice
   collection.insert(new RWCollectableInt(100));
   collection.insert(new RWCollectableDate(3, "May", 1959));

   // "Store" to cout using portable stream:
   RWpostream ostr(cout);
   ostr << collection;
      // The above statement calls the insertion operator:
      //    Rwvistream&
      //      operator<<(RWvistream&, const RWCollectable&);

   // Now delete all the members in collection.
   // clearAndDestroy() has been written so that it deletes
   // each object only once, so that you do not have to
   // worry about deleting the same object too many times.

   collection.clearAndDestroy();

   return 0;
}
```

Note that there are three types of objects stored in `collection`, an `RWCollectableDate`, and `RWCollectableInt`, and two `RWCollectableString`s. The same `RWCollectableString`, `george`, is inserted into `collection` twice.

Example Two: Restoring Polymorphically

The second example shows how the polymorphically saved collection of the first example can be read back in and faithfully restored using the overloaded extraction operator:

```
Rwvistream&  operator>>(RWvistream&, RWCollectable&);
```

In this example, persistence happens when the program executes the statement:

```
istr >> collection2;
```

This statement uses the overloaded extraction operator to isomorphically restore the collection saved by the first example into `collection2`.

How does persistence happen? For each pointer to an *RWCollectable*-derived object restored into `collection2` from the input stream `istr`, the extraction operator `operator>>` calls a variety of overloaded extraction operators and persistence functions. For each *RWCollectable*-derived object pointer, `collection2`'s extraction operators:

- Read the stream `istr` to discover the type of the *RWCollectable*-derived object.
- Read the stream `istr` to see if the *RWCollectable*-derived object that is pointed to has already been restored and referenced in the restore table.
  - *If the RWCollectable-derived object has not yet been restored*, the extraction operators create a pointer, create an object of the correct type from the heap, and initialize the created object with data read from the stream. Then the operators update the pointer with the address of the new object, and finally save a reference to the object in the restore table.
  - *If the RWCollectable-derived object has already been restored*, the extraction operators create a pointer and read the reference to the object from the stream. Then the operators use the reference to get the object's address from the restore table, and update the pointer with this address.
- Finally, the restored pointer is inserted into the collection.

We'll look at the implementation details for the persistence mechanism again in “Example Two Revisited”. You should note, however, that when a heterogeneous collection (which must be based on *RWCollection*) is restored, the restoring process does not know the types of objects it will be restoring. Hence, it must *always* allocate the objects off the heap. This means that *you are responsible for deleting the restored contents.* This happens at the end of the example, in the expression `collection2.clearAndDestroy`.

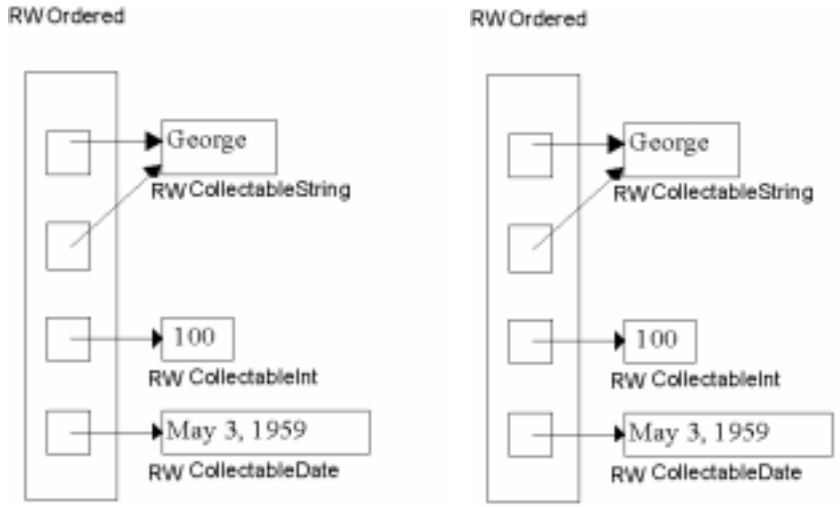
Here is the listing of the example:

```
#define RW_STD_TYPEDEFS
#include <rw/ordcltn.h>
#include <rw/collstr.h>
#include <rw/collint.h>
#include <rw/colldate.h>
#include <rw/pstream.h>

main(){
   RWpistream istr(cin);
   RWOrdered collection2;

   // Even though this program does not need to have prior
   // knowledge of exactly what it is restoring, the linker
   // needs to know what the possibilities are so that the
   // necessary code is linked in for use by RWFactory.
   // RWFactory creates RWCollectable objects based on
   // class ID's.

       RWCollectableInt   exemplarInt;
       RWCollectableDate  exemplarDate;

   // Read the collection back in:
       istr >> collection2;

   // Note: The above statement is the code that restores
   // the collection.  The rest of this example shows us
   // what is in the collection.

   // Create a temporary string with value "George"
   // in order to search for a string with the same value:
       RWCollectableString temp("George");

   // Find a "George":
   //   collection2 is searched for an occurrence of a
   //   string with value "George".
   //   The pointer "g" will point to such a string:
         RWCollectableString* g;
         g = (RWCollectableString*)collection2.find(&temp);

   // "g" now points to a string with the value "George"
```

```
// How many occurrences of g are there in the collection?

size_t georgeCount   = 0;
size_t stringCount   = 0;
size_t integerCount  = 0;
size_t dateCount     = 0;
size_t unknownCount  = 0;

// Create an iterator:
    RWOrderedIterator sci(collection2);
    RWCollectable* item;

// Iterate through the collection, item by item,
// returning a pointer for each item:

    while ( item = sci() ) {

      // Test whether this pointer equals g.
      // That is, test for identity, not just equality:
         if (item->isA() == __RWCOLLECTABLESTRING && item==g)
            georgeCount++;

      // Count the strings, dates and integers:
          switch (item->isA()) {
            case __RWCOLLECTABLESTRING: stringCount++; break;
           case __RWCOLLECTABLEINT:    integerCount++; break;
            case __RWCOLLECTABLEDATE:   dateCount++; break;
           default:                    unknownCount++; break;
          }
}

// Output results:
    cout << "There are:\n\t"
      << stringCount   << " RWCollectableString(s)\n\t"
      << integerCount  << " RWCollectableInt(s)\n\t"
      << dateCount     << " RWCollectableDate(s)\n\t"
      << unknownCount  << " other RWCollectable(s)\n\n"
      << "There are "
      << georgeCount
      << " pointers to the same object \"George\"" << endl;

// Delete all objects created and return:
```

```
       collection2.clearAndDestroy();
       return 0;
}

Program Output:
There are:
        2 RWCollectableString(s)
        1 RWCollectableInt(s)
        1 RWCollectableDate(s)
        0 other RWCollectable(s)

There are 2 pointers to the same object "George"
```

Figure 14-8 illustrates the collection created in the first example and restored in the second. Notice that both the memory map and the datatypes are identical in the saved and restored collection.



Collection to be saved (`collection1`) *Collection restored* (`collection2`)

Figure 14-8 Polymorphic Persistence

Example Two Revisited

It is worth looking at the second example again so that you can see the mechanisms used to implement polymorphic persistence. The expression:

```
istr >> collection2;
```

calls the overloaded extraction operator:

```
RWvistream& operator>>(RWvistream& str, RWCollectable& obj);
```

This extraction operator has been written to call the object's `restoreGuts()` virtual function. In this case the object, `obj`, is an ordered collection and its version of `restoreGuts()` has been written to repeatedly call:

```
RWvistream& operator>>(RWvistream&, RWCollectable*&);
```

once for each member of the collection[1]. Notice that its second argument is a *reference to a pointer*, rather than just a reference. This version of the overloaded `operator>>` looks at the stream, figures out the kind of object on the stream, allocates an object of that type off the heap, restores it from the stream, and finally returns a pointer to it. If this `operator>>` encounters a reference to a previous object, it just returns the old address. These pointers are inserted into the collection by the ordered collection's `restoreGuts()`.

These details about the polymorphic persistence mechanism are particularly important when you design your own polymorphically persistable class, as described in Chapter 15, Designing an RWCollectable Class. And when working with such classes, note that when Smalltalk-like collection classes are restored, the type of the restored objects is never known. Hence, the restoring processes must *always* allocate those objects off the heap. This means that *you are responsible for deleting the restored contents*. An example of this occurs at the end of both polymorphic persistence examples.

Choosing Which Persistence Operator to Use

In the second example, the persistence operator restored our collection to a reference to an *RWCollectable*:

```
Rwvistream&  operator>>(RWvistream&, RWCollectable&);
```

instead of to a pointer to a reference to an *RWCollectable*:

```
Rwvistream&  operator>>(RWvistream&, RWCollectable*&);
```

The collection was allocated on the stack:

1. Actually, the Smalltalk collection classes are so similar that they all share the same version of `restoreGuts()`, inherited from *RWCollection*.

```
RWpistream istr(cin);
RWOrdered collection2;
istr >> collection2;
...
collection2.clearAndDestroy();
```

instead of having `operator>>(RWvistream&,RWCollectable*&)` allocate the memory for the collection:

```
RWpistream istr(cin);
RWOrdered* pCollection2;
istr >> pCollection2;
...
collection->clearAndDestroy();
delete pCollection2;
```

Why make this choice? If you know the type of the collection you are restoring, then you are usually better off allocating it yourself, then restoring via:

```
Rwvistream&  operator>>(RWvistream&, RWCollectable&);
```

By using the reference operator, you eliminate the time required for the persistence machinery to figure out the type of object and have the *RWFactory* allocate one (see "A Note on the RWFactory" on page 208). Furthermore, by allocating the collection yourself, you can tailor the allocation to suit your needs. For example, you can decide to set an initial capacity for a collection class.

A Few Friendly Warnings

Persistence is a useful quality, but requires care in some areas. Here are a few things to look out for when you use persistence on objects.

Always Save an Object by Value before Saving the Identical Object by Pointer

In the case of both isomorphic and polymorphic persistence of objects, you should never stream out an object by pointer before streaming out the identical object by value. Whenever you design a class that contains a value and a pointer to that value, the `saveGuts` and `restoreGuts` member functions for that class should always save or restore the value *then* the pointer.

Here is an example that creates a class that uses isomorphic persistence, then instantiates objects of the class and attempts to persist those objects isomorphically. *This example will fail.* See the explanation that follows the code.

```
   class Elmer {
   public:
    /* ... */
    Elroy  elroy_;
    Elroy* elroyPtr_;  // elroyPtr_ will point to elroy_.
};

RWDEFINE_PERSISTABLE(Elmer)

void rwSaveGuts(RWFile& file, const Elmer& elmer) {

  // Create a value for elroyPtr_ and stream it:
      file << *elroyPtr_;

  // If elroyPtr_ == &elroy_, store a reference
  // to elroy_ that points to the created value for
  // elroyPtr_:
      file << elroy_;
}

void rwRestoreGuts(RWFile& file, Elmer& elmer){

  // Create a value for elroyPtr_ in memory
  // and change elroyPtr_ to point to that
  // value in memory:
      file >> elroyPtr_;

  // Assign reference to value.
  // If elroyPtr_ == &elroy then the value of
  // elroy_ will already be created but now
  // elroyPtr_ != &elroy so an
  // RWTOOL_REF exception will be thrown:
      file >> elroy_;
}

/* ... */
RWFile file("elmer.dat");
Elmer  elmer;
Elmer  elmer2;
elmer.elroyPtr_ = &(elmer.elroy_);
```

```
/* ... */
file << elmer;  // Trouble is coming...
/* ... */
file >> elmer2;  // Trouble has arrived. RWTOOL_REF exception!
/* ... */
```

In the above code, the following statement isomorphically saves `elmer` to `file`:

```
file << elmer;
```

First, the statement calls the insertion operator:

```
operator<<(RWFile&, const Elmer&)
```

Since `elmer` hasn't been saved yet, the value of `elmer` will be saved to `file` and that value will be added to the isomorphic save table. However, `elmer` has the members `elroy_` and `elroyPtr_`, which must be saved as part of saving `elmer`. The members of `elmer` are saved in `rwSaveGuts(RWFile&, const Elmer&)`.

The function `rwSaveGuts(RWFile&, const Elmer&)` saves `elroyPtr_` first, then `elroy_`. When `rwSaveGuts` saves the value `*elroyPtr_`, it calls the insertion operator `operator<<(RWFile&, const Elroy&)`.

The insertion operator `operator<<(RWFile&, const Elroy&)` sees that `elmer.elroyPtr_` hasn't been stored yet, so it saves the value `*(elmer.elroyPtr_)` to `file`, and makes a note in the isomorphic save table that this value has been stored. Then `operator<<(RWFile&, const Elroy&)` returns to `rwSaveGuts(RWFile&, const Elmer&)`.

Back in `rwSaveGuts(RWFile&, const Elmer&)`, it's time for `elmer.elroy_` to be saved. In this example, `elmer.elroyPtr_` has the same address as `elmer.elroy_`. Once again the insertion operator `operator<<(RWFile&, const Elroy&)` is called, but this time the insertion operator notices from the isomorphic save table that `*(elmer.elroyPtr_)` has already been stored, so a *reference* to `*(elmer.elroyPtr_)` is stored to `file` instead of the *value*.

Everything seems okay, but trouble is looming. Trouble arrives with the statement:

```
file >> elmer2;
```

This statement calls the extraction operator `operator>>(RWFile&, const Elmer&)`. Since `elmer2` hasn't been restored yet, the value of `elmer2` will be extracted from `file` and added to the isomorphic restore table. In order to extract the value of `elmer2`, the members `elroy_` and `elroyPtr_` must be extracted. The members of `elmer2` are extracted in `rwRestoreGuts(RWFile&, Elmer&)`.

The function `rwRestoreGuts(RWFile&, Elmer&)` restores `elroyPtr_` first, then `elroy_`. When `rwRestoreGuts` restores the value `*elroyPtr_`, it calls the extraction operator `operator>>(RWFile&, Elroy*&)`.

The extraction operator `operator>>(RWFile&, Elroy*&)` sees that `elmer2.elroyPtr_` hasn't yet been extracted from `file`. So the extraction operator extracts the value `*(elmer2.elroyPtr_)` from `file`, allocates memory to create this value, and updates `elmer2.elroyPtr_` to point to this value. Then the extraction operator makes a note in the isomorphic restore table that this value has been created. After making the note, `operator>>(RWFile&, Elroy&)` returns to `rwRestoreGuts(RWFile&, Elmer&)`.

Back in `rwRestoreGuts(RWFile&, Elmer&)`, it's time to restore `elmer2.elroy_`. Remember that `elmer.elroyPtr_` has the same address as `elmer.elroy_`. The `rwRestoreGuts` function calls the extraction operator `operator>>(RWFile&, Elroy&)`, which notices from the isomorphic save table that `*(elmer2.elroyPtr_)` has already been extracted from `file`. Because a value has already been stored in the restore table, the extraction operator extracts a *reference* to `*(elmer.elroyPtr_)` from `file` instead of extracting the *value*. But the extraction operator notices that the address of the value that `elmer2.elroyPtr_` put into the restore table is different from the address of `elmer2.elroy_`. So `operator>>(RWFile&, Elroy&)` throws an `RWTOOL_REF` exception, and the restoration is aborted.

The solution to the problem in this particular case is easy: reverse the order of saving and restoring `Elmer`'s members! Here is the problem:

```
// WRONG!


void rwSaveGuts(RWFile& file, const Elmer& elmer) {
  file << *elroyPtr_;
  file << elroy_;
}
```

```
void rwRestoreGuts(RWFile& file, Elmer& elmer){
  file >> elroyPtr_;
  file >> elroy_;
}
```

Instead, you should write your functions the following way:

```
// RIGHT!

void rwSaveGuts(RWFile& file, const Elmer& elmer) {
  file << elroy_;
  file << *elroyPtr_;
}
void rwRestoreGuts(RWFile& file, Elmer& elmer){
  file >> elroy_;
  file >> elroyPtr_;
}
```

If you correct `rwRestoreGuts` and `rwSaveGuts` as suggested above, then the isomorphic save and restore tables can use the address of `Elmer::elroy_` to update `Elmer::elroyPtr_` if necessary.

Summary: Because of the possibility of having both a value and a pointer that points to that value in the same class, it's a good idea to define your `rwSaveGuts` and `rwRestoreGuts` member functions so that they always operate on values before pointers.

Don't Save Distinct Objects with the Same Address

You must be careful not to isomorphically save distinct objects that may have the same address. The internal tables that are used in isomorphic and polymorphic persistence use the address of an object to determine whether or not an object has already been saved.

The following example assumes that all the work has been done to make `Godzilla` and `Mothra` isomorphically persistable:

```
class Mothra {/* ... */};
RWDEFINE_PERSISTABLE(Mothra)

struct Godzilla {
  Mothra  mothra_;
  int     wins_;
};
RWDEFINE_PERSISTABLE(Godzilla)
```

```
/*... */
Godzilla  godzilla;
/* ... */
stream << godzilla;
/* ... */
stream >> godzilla;       // The restore may be garbled!
```

When `godzilla` is saved, the address of `godzilla` will be saved in an isomorphic save table. The next item to be saved is `godzilla.mothra_`. Its address is saved in the same internal save table.

The problem is that on some compilers `godzilla` and `godzilla.mothra_` have the same address! Upon restoration of `godzilla`, `godzilla.mothra_` is streamed out as a value, and `godzilla` is streamed out as a reference to `godzilla.mothra_`. If `godzilla` and `godzilla.mothra` have the same address, the restore of `godzilla` fails because the extraction operator attempts to initialize `godzilla` with the contents of `godzilla.mothra_`.

There are two ways to overcome this difficulty. The first is to structure your class so that simple data members, such as `int`, precede data members that are isomorphically persistent. Using this method, class `Godzilla` looks like this:

```
struct Godzilla {
  int    wins_;
  Mothra mothra_;   // mothra_ now has a different address.
};
```

If `Godzilla` is structured as shown here, `mothra_` is displaced from the front of `godzilla` and can't be confused with `godzilla`. The variable `wins_`, of type `int`, is saved with simple persistence and is not stored in the isomorphic save table.

The second approach to solving the problem of identical addresses between a class and its members is to insert an isomorphically persistable member as a pointer rather than a value. For `Godzilla` this would look like:

```
struct Godzilla {
  Mothra* mothraPtr_;// mothraPtr_ points to a different address.
  int     wins_;
};
```

In this second approach, `mothraPtr_` points to a different address than `godzilla`, so confusion is once again avoided.

Don't Use Sorted RWCollections to Store Heterogeneous RWCollectables

When you have more than one different type of *RWCollectable* stored in an *RWCollection*, you can't use a sorted *RWCollection*. For example, this means that if you plan to store *RWCollectableString*s and *RWCollectableDate*s in the same *RWCollection*, you can't store them in a sorted *RWCollection* such as *RWBtree*. The sorted *RWCollections* are *RWBinaryTree*, *RWBtree*, *RWBTreeDictionary*, and *RWSortedVector*.

The reason for this restriction is that the comparison functions for sorted *RWCollection*s expect that the objects to be compared will have the same type.

Define All RWCollectables That Will Be Restored

Make certain that your program declares variables of all possible *RWCollectable* objects that you might restore. For an example of this practice, see “Example Two: Restoring Polymorphically” on page 178.

These declarations are of particular concern when you save an *RWCollectable* in a collection, then attempt to take advantage of polymorphic persistence by restoring the collection in a different program, without using the *RWCollectable* that you saved. If you don't declare the appropriate variables, during the restore attempt the *RWFactory* will throw an `RW_NOCREATE` exception for some *RWCollectable* class ID that you know exists. The *RWFactory* won't throw an `RW_NOCREATE` exception when you declare variables of all the *RWCollectable*s that could be polymorphically restored.

The problem occurs because your compiler's linker only links the code that *RWFactory* needs to create the missing *RWCollectable* when that *RWCollectable* is specifically mentioned in your code. Declaring the missing *RWCollectable*s gives the linker the information it needs to link the appropriate code needed by *RWFactory*.

14

Designing an RWCollectable Class 15

Classes that derive from *RWCollectable* carry two major advantages: they can be used by the Smalltalk-like collections, which also derive from *RWCollectable*, and they are the only set of collection classes able to use the powerful polymorphic persistence machinery. In this section, we will provide some examples of *RWCollectable* classes, then describe how to create your own *RWCollectable* classes.

What we don't do in this section is describe the mechanism of polymorphic persistence itself. Polymorphic, isomorphic, and simple persistence are all covered in detail in the previous chapter called *Persistence*. To summarize, polymorphic persistence is the storage and retrieval of objects to and from a stream or file in such a way that pointer relationships are preserved among persisted objects, which can be restored without the restoring process knowing the object's type.

Why Design an RWCollectable Class?

Before we get to the nuts and bolts of *how* to design an *RWCollectable* class, let's discuss a concrete example of why you might choose to design *RWCollectable* classes.

Suppose you run a bus company. To automate part of your ridership tracking system, you want to write classes that represent a bus, its set of customers, and its set of actual passengers. In order to be a passenger, a person must be a customer. Hence, the set of customers is a superset of the set of passengers. Also, a person can physically be on the bus only once, and there is no point in

putting the same person on the customer list more than once. As the developer of this system, you must make sure there are no duplicates on either list.

These duplicates can be a problem. Suppose that the program needs to be able to save and restore information about the bus and its customers. When it comes time to polymorphically save the bus, if your program naïvely iterates over the set of customers, then over the set of passengers, saving each one, any person who is both a customer and a passenger is saved twice. When the program polymorphically restores the bus, the list of passengers will not simply *refer to* people already on the customer list. Instead, each passenger will have a separate instantiation on both lists.

You need some way of recognizing when a person has already been polymorphically saved to the stream and, instead of saving him or her again, merely saving a reference to the previous instance.

This is the job of class *RWCollectable*. Objects that inherit from *RWCollectable* have the ability to save not only their contents, but also their *relationships* with other objects that inherit from *RWCollectable*. We call this feature *isomorphic persistence*. Class *RWCollectable* has *isomorphic persistence*, but more than that, it can determine at run time the type of the object to be saved or restored. We call the type of persistence provided by *RWCollectable polymorphic persistence*, and recognize it as a superset of isomorphic persistence.

An Example of RWCollectable Classes

The code below shows how we might declare the classes described in the previous section. Later we'll use the macro `RWDECLARE_COLLECTABLE` and discuss our function choices. You'll find the complete code from which this example is taken at the end of this chapter; it is also given as the bus example in the `toolexam` directory.

```
class Bus : public RWCollectable  {
  RWDECLARE_COLLECTABLE(Bus)

public:

   Bus();
```

```
   Bus(int busno, const RWCString& driver);
   ~Bus();

   // Inherited from class "RWCollectable":
   Rwspace     binaryStoreSize() const;
   int         compareTo(const RWCollectable*) const;
   RWBoolean   isEqual(const RWCollectable*) const;
   unsigned    hash() const;
   void        restoreGuts(RWFile&);
   void        restoreGuts(RWvistream&);
   void        saveGuts(RWFile&) const;
   void        saveGuts(RWvostream&) const;

   void        addPassenger(const char* name);
   void        addCustomer(const char* name);
   size_t      customers() const;
   size_t      passengers() const;
   RWCString   driver() const    {return driver_;}
   int         number() const    {return busNumber_;}

private:

   RWSet       customers_;
   RWSet*      passengers_;
   int         busNumber_;
   RWCString   driver_;
};

class Client : public RWCollectable  {
  RWDECLARE_COLLECTABLE(Client)
  Client(const char* name) : name_(name) {}
private:
  RWCString    name_;
//ignore other client information for this example
};
```

Note how both classes inherit from *RWCollectable*. We have chosen to implement the set of customers by using class *RWSet*, which does not allow duplicate entries. This will guarantee that the same person is not entered into the customer list more than once. For the same reason, we have also chosen to implement the set of passengers using class *RWSet*. However, we have chosen to have this set live on the heap. This will help illustrate some points in the coming discussion.

15

How to Create an RWCollectable Object

Here's an outline of how to make your object inherit from *RWCollectable*. Additional information about how to do each step appears in the indicated section.

1. Define a default constructor. See “Define a Default Constructor” on page 195.

2. Add the macro `RWDECLARE_COLLECTABLE` to your class declaration. See “Add RWDECLARE_COLLECTABLE() to your Class Declaration” on page 195.

3. Provide a class identifier for your class by adding one of two definition macros, `RWDEFINE_COLLECTABLE` or `RWDEFINE_NAMED_COLLECTABLE`, to one and only one source file (`.cpp`), to be compiled. See “Provide a Class Identifier for Your Class” on page 195.

4. Add definitions for inherited virtual functions as necessary. You may be able to use inherited definitions. “Add Definitions for Virtual Functions” on page 198 discusses the following virtual functions:

```
Int          compareTo(const RWCollectable*) const;
RWBoolean    isEqual(const RWCollectable*) const;
unsigned     hash() const;
```

5. Consider whether you need to define a destructor. See “Object Destruction” on page 202.

6. Add persistence to the class. You may be able to use inherited definitions, or you may have to add definitions for the following functions. See “How to Add Polymorphic Persistence” on page 202.

```
RWspace       binaryStoreSize() const;
void          restoreGuts(RWFile&);
void          restoreGuts(RWvistream&);
void          saveGuts(RWFile&) const;
void          saveGuts(RWvostream&) const;
```

A note on *RWFactory* follows these steps. See “A Note on the RWFactory” on page 208.

Define a Default Constructor

All *RWCollectable* classes must have a default constructor . The default constructor takes no arguments. The persistence mechanism uses this constructor to create an empty object, then restore that object with appropriate contents.

Default constructors are necessary in order to create vectors of objects in C++, so providing a default constructor is a good habit to get into anyway. Here's a possible definition of a default constructor for our *Bus* class.

```
Bus::Bus() :
  busNumber_  (0),
  driver_     ("Unknown"),
  passengers_ (rwnil)
{
}
```

Add RWDECLARE_COLLECTABLE() to your Class Declaration

The example in "An Example of RWCollectable Classes" on page 192 includes the macro invocation `RWDECLARE_COLLECTABLE(Bus)` in the declaration for *Bus*. You must put this macro in your class declaration, using the class name as the argument. Using the macro guarantees that all necessary member functions are declared correctly.

Provide a Class Identifier for Your Class

Polymorphic persistence lets you save a class in one executable, and restore it in a different executable or in a different run of the original executable. The restoring executable can use the class, without prior knowledge of its type. In order to provide polymorphic persistence, a class must have a unique,[1] unchanging identifier. Because classes derived from *RWCollectable* are polymorphically persistent, they must have such an identifier.

Identifiers can be either numbers or strings. A numeric identifier is an `unsigned short` with a typedef of `RWClassID`. A string identifier has a typedef of `RWStringID`. If you choose to specify a numeric identifier, your class will have an automatically generated string identifier, which will be the

1. Strictly, it only needs to be different from every other identifier in any given executable.

same sequence of characters as the name of the class. Similarly, if you choose to specify a string identifier, your class will have an automatically generated numeric ID when used in an executable.

Tools.h++ includes two definition macros to provide an identifier for the class you design. If you want to specify a numeric ID, use:

```
RWDEFINE_COLLECTABLE (className, numericID)
```

If you want to specify a string ID, use:

```
RWDEFINE_NAMED_COLLECTABLE (className, stringID)
```

Note that you do not include the definition macros in the header file for the class. Rather, the macros are part of a `.cpp` file that uses the class. You must include exactly one define macro for each *RWCollectable* class that you're creating, in one and only source file (`.cpp`). Use the class name as the first argument, and a numeric class ID or string class ID as the second argument. For the bus example, you can include the following definition macros:

```
RWDEFINE_COLLECTABLE(Bus, 200)
```

or:

```
RWDEFINE_NAMED_COLLECTABLE(Client, "a client")
```

The first use provides a numeric ID `200` for class *Bus*, and the second provides a string ID, `"a client"`, for class *Client*.

In the remainder of this manual, we use *RWDEFINITION_MACRO* to indicate that you can choose either of these macros. In example code, we will pick one or the other macro.

Either macro will automatically supply the definitions for the virtual functions `isA()` and `newSpecies()`.[1] In “Virtual Function isA( )” on page 197 through “A Note on the RWFactory” on page 208, we describe these virtual functions, discuss the `stringID()` method which is new in Version 7 of *Tools.h++*, and provide a brief introduction to the *RWFactory* class, which helps implement polymorphic persistence.

1. The *RWDEFINITION_MACRO*s do more than merely implement the two mentioned methods. Before you choose not to use one of the provided macros, review them in detail to be sure you understand all that they do.

Virtual Function isA( )

The virtual function `isA()` returns a *class identifier*: a unique number that identifies an object's class. It can be used to determine the class to which an object belongs. Here's the function declaration provided by macro `RWDECLARE_COLLECTABLE`:

```
virtual RWClassID isA() const;
```

`RWClassID` is actually a typedef to an `unsigned short`. *Numbers from 0x8000 (hex) and up are reserved for use by Rogue Wave.* You may choose a numeric class ID from 9x0001 to 0x7fff. There is a set of class symbols defined in `<rw/tooldefs.h>` for the *Tools.h++* Class Library. Generally, these follow the pattern of a double underscore followed by the class name with all letters in upper case. For example:

```
RWCollectableString yogi;
yogi.isA() == __RWCOLLECTABLESTRING;          // Evaluates TRUE
```

The macro `RWDECLARE_COLLECTABLE(className)` will automatically provide a declaration for `isA()`. Either *RWDEFINITION_MACRO* will supply the definition.

Virtual Function newSpecies()

The job of this function is to return a pointer to a brand new object of the same type as self. Here is the function declaration provided by macro `RWDECLARE_COLLECTABLE`:

```
virtual RWCollectable*  newSpecies() const;
```

The definition is automatically provided by either version of *RWDEFINITION_MACRO*.

Function stringID()

The `stringID()` function *acts* like a virtual function, but it is not.[1] It returns an instance of *RWStringID*, a unique string that identifies an object's class. *RWStringID* is derived from class *RWCString*. By default, the string identifier for a class is the same as the name of the class. *RWStringID* can be used instead of, or as a suppplement to, *RWClassID*s.

1. See "Implementing Virtuals Via Statics" on page 258 for a discussion of *RWStringID* and how to mimic a virtual function. We wrote the code this way to maintain link compatibility with object code compiled from the previous version of *Tools.h++*.

Add Definitions for Virtual Functions

Class *RWCollectable* declares the following virtual functions:

```
virtual                 ~RWCollectable();
virtual Rwspace         binaryStoreSize() const;
virtual int             compareTo(const RWCollectable*) const;
virtual unsigned        hash() const;
virtual RWClassID       isA() const;
virtual RWBoolean       isEqual(const RWCollectable*) const;
virtual RWCollectable*  newSpecies() const;
virtual void            restoreGuts(RWvistream&);
virtual void            restoreGuts(RWFile&);
virtual void            saveGuts(RWvostream&) const;
virtual void            saveGuts(RWFile&) const;
```

In these functions `RWBoolean` is a typedef for an `int`, `RWspace` is a typedef for `unsigned long`, and `RWClassID` is a typedef for an `unsigned short`. Any class that derives from class *RWCollectable* should be able to understand any of these methods. Although default definitions are given for all of them in the base class *RWCollectable*, it is best for you as the class designer to provide definitions tailored to the class at hand.

We've split our discussion of these virtual functions. We discuss the destructor in “Object Destruction” on page 202, and the `binaryStoreSize()`, `saveGuts()`, and `restoreGuts()` functions in “How to Add Polymorphic Persistence” on page 202, where we describe how to add persistence to a class. Virtual functions `isA()` and `newSpecies()` are declared and defined by macros, so they were discussed above, in “Virtual Function isA( )” and “Virtual Function newSpecies()”. This section presents discussion on the remaining functions: `compareTo()`, `isEqual()`, and `hash()`. A very brief example, showing how all three functions deal with the same data, appears in “An Example of compareTo(), isEqual(), and hash()” on page 201.

Virtual Function compareTo()

The virtual function `compareTo()` is used to order objects relative to each other. This function is required in collection classes that depend on such ordering, such as *RWBinaryTree* or *RWBTree*. Here is its declaration:

```
virtual int  compareTo(const RWCollectable*) const;
```

The function `int compareTo(const RWCollectable*) const` should return a number greater than zero if self is greater than the argument, a number less than zero if self is less than the argument, and zero if self is equal to the argument.

The definition and meaning of whether one object is greater than, less than, or equal to another object is left to the class designer. The default definition, found in class *RWCollectable*, is to compare the two addresses of the objects. This default definition should be considered a placeholder; in practice, it is not very useful and could vary from run to run of a program.

Here is a possible definition of `compareTo()`:

```
int Bus::compareTo(const RWCollectable* c) const
{  const Bus* b = (const Bus*)c;
   if (busNumber_ == b->busNumber_) return 0;
   return busNumber_ > b->busNumber_ ? 1 : -1;
}
```

Here we are using the bus number as a measure of the ordering of buses. If we need to insert a group of buses into an *RWBinaryTree*, they would be sorted by their bus number. Note that there are many other possible choices—we could have used the driver name, in which case they would have been sorted by the driver name. Which choice you use will depend on your particular problem.

There is a hazard here. We have been glib in assuming that the actual type of the *RWCollectable* which `c` points to is always a *Bus*. If a careless user inserted, say, an *RWCollectableString* into the collection, then the results of the cast `(const Bus*)c` would be invalid, and dereferencing it could bring disaster[1]. The necessity for all overloaded virtual functions to share the same signatures requires that they return the lowest common denominator, in this case, class *RWCollectable*. The result is that much compile-time type checking breaks down.

Note – You must be careful that the members of a collection are either homogeneous (i.e., all of the same type), or that there is some way of telling them apart. The member functions `isA()` or `stringID()` can be used for this.

---

1. This is a glaring deficiency in C++ that the user must constantly be aware of, especially if the user plans to have heterogeneous collections. See the section in *Persistence* called *Don't Use Sorted RWCollections to Store Heterogeneous Collections* for a description of the problem.

Virtual Function isEqual()

The virtual function `isEqual()` plays a similar role to the tester function of the generic collection classes described in "Tester Functions" on page 118.

```
RWBoolean  isEqual(const RWCollectable* c) const;
```

The function `RWBoolean isEqual(const RWCollectable*)` should return `TRUE` if the object and its argument are considered equal, and `FALSE` otherwise. The definition of equality is left to the class designer. The default definition, as defined in class *RWCollectable*, is to test the two addresses for equality, that is, to test for *identity*.

Note that `isEqual` does not have to be defined as being identical. Rather `isEqual` can mean that two objects are equivalent in some sense. In fact, the two objects need not even be of the same type. The only requirement is that the object passed as an argument must inherit type *RWCollectable*. You are responsible for making sure that any typecasts you do are appropriate.

Also note that there is no formal requirement that two objects that compare equal (i.e., `compareTo()` returns zero) must also return `TRUE` from `isEqual()`, although it is hard to imagine a situation where this wouldn't be the case. It is also possible to design a class for which the `isEqual` test returns true for objects that have different hash values. This would make it impossible to search for such objects in a hash-based collection.

For the *Bus* class, an appropriate definition of `isEqual` might be:

```
RWBoolean Bus::isEqual(const RWCollectable* c) const
{  const Bus* b = (const Bus*)c;
   return busNumber_ == b->busNumber_;
}
```

Here we are considering buses to be equal if their bus numbers are the same. Again, other choices are possible.

Virtual Function hash()

The function `hash()` should return an appropriate hashing value for the object. Here is the function's declaration:

```
unsigned  hash() const;
```

A possible definition of `hash()` for our class *Bus* might be:

```
unsigned Bus::hash() const{
  return (unsigned)busNumber_;
}
```

The example above simply returns the bus number as a hash value. Alternatively, we could choose the driver's name as a hash value:

```
unsigned Bus::hash() const{
  return driver_.hash();
}
```

In the above example, `driver_` is an *RWCString* that already has a hash function defined.

Note – we expect that two objects that test `TRUE` for `isEqual` will hash to the same value.

An Example of compareTo(), isEqual(), and hash()

We've described three inherited virtual functions: `compareTo()`, `isEqual()`, and `hash()`. Here is an example that defines a set of objects, and applies the functions. The results of the functions appear as comments in the code.

```
RWCollectableString a("a");
RWCollectableString b("b");
RWCollectableString a2("a");

a.compareTo(&b);      // Returns -1
a.compareTo(&a2);     // Returns 0 ("compares equal")
b.compareTo(&a);      // Returns 1

a.isEqual(&a2);       // Returns TRUE
a.isEqual(&b);        // Returns FALSE

a.hash()              // Returns 96 (operating system dependent)
```

Note that the `compareTo()` function for *RWCollectableString*s has been defined to compare strings lexicographically in a case sensitive manner. See class *RWCString* in the *Class Reference* for details.

Object Destruction

All objects inheriting from class *RWCollectable* inherit a virtual destructor. Hence, the actual type of the object need not be known until run time in order to delete the object. This allows all items in a collection to be deleted without knowing their actual type.

As with any C++ class, objects inheriting from *RWCollectable* may need a destructor to release the resources they hold. In the case of *Bus*, the names of passengers and customers are *RWCollectableString*s that were allocated off the heap. Hence, they must be reclaimed. Because these strings never appear outside the scope of the class, we do not have to worry about the user having access to them. Hence, we can confidentially delete them in the destructor, knowing that no dangling pointers will be left.

Furthermore, because the set pointed to by `customers_` is a superset of the set pointed to by `passengers_`, it is essential that we delete only the contents of `customers_`.

Here's a possible definition:

```
Bus::~Bus()
{  customers_.clearAndDestroy();
   delete passengers_;
}
```

Note that the language guarantees that it is okay to call `delete` on the pointer `passengers_` even if it is nil.

How to Add Polymorphic Persistence

The `saveGuts()` and `restoreGuts()` virtual functions are responsible for saving and restoring the internal state of *RWCollectable* objects. To add persistence to your *RWCollectable* class, you must override the `saveGuts()` and `restoreGuts()` virtual member functions so that they write out all of your object's member data. "Virtual Functions saveGuts(RWFile&) and saveGuts(RWvostream&)" on page 203 and "Virtual Functions restoreGuts(RWFile&) and restoreGuts(RWvistream&)" on page 205 describe approaches you can use to correctly define these functions. "Multiply-referenced Objects" on page 205 describes how these functions handle multiply-referenced objects.

Polymorphically saving an object to a file may require some knowledge of the number of bytes that need to be allocated for storage of an object. The `binaryStoreSize()` function calculates this value. "Virtual Function binaryStoreSize()" on page 206 describes how to use `binaryStoreSize()`.

RWCollection has its own versions of the `saveGuts()` and `restoreGuts()` functions that are used to polymorphically save collections that inherit from that class. "Polymorphically Persisting Custom Collections" on page 207 briefly describes how these functions work.

Virtual Functions saveGuts(RWFile&) and saveGuts(RWvostream&)

The `saveGuts(RWFile&)` and `saveGuts(RWvostream&)` virtual functions are responsible for polymorphically saving the internal state of an *RWCollectable* object on either a binary file, using class *RWFile*, or on a virtual output stream, using class *RWvostream*.[1] This allows the object to be restored at some later time, or in a different location. Here are some rules for defining a `saveGuts()` function:

1. Save the state of your base class by calling its version of `saveGuts()`.
2. For each type of member data, save its state. How to do this depends upon the type of the member data:
   - *Primitives.* For primitives, save the data directly. When saving to *RWFile*s, use `RWFile::Write()`; when saving to virtual streams, use the insertion operator `RWvostream::operator<<()`.
   - *Rogue Wave classes.* Most Rogue Wave classes offer an overloaded version of the insertion operator. For example, *RWCString* offers:

     ```
     RWvostream& operator<<(RWvostream&, const RWCString&
     str);
     ```

     Hence, many Rogue Wave classes can simply be shifted onto the stream.
   - *Objects inheriting from RWCollectable.* For most of these objects, use the global function:

     ```
     RWvostream& operator<<(RWvostream&,
                            const RWCollectable& obj);
     ```

     This function will call `saveGuts()` recursively for the object.

1. For a description of the persistence mechanism, see the section called *Persistence*.

With these rules in mind, let's look at a possible definition of the `saveGuts()` functions for the *Bus* example:

```
void Bus::saveGuts(RWFile& f) const
{  RWCollectable::saveGuts(f);    // Save base class
   f.Write(busNumber_);           // Write primitive directly
   f << driver_ << customers_;    // Use Rogue Wave
                                  //provided versions
   f << passengers_;              // Will detect nil pointer
                                  // automatically
}

void Bus::saveGuts(RWvostream& strm) const
{  RWCollectable::saveGuts(strm); // Save base class
   strm << busNumber_;            // Write primitives directly
   strm << driver_ << customers_; // Use Rogue Wave
                                  // provided versions
   strm << passengers_;           // Will detect nil pointer
                                  // automatically
}
```

Member data `busNumber_` is an `int`, a C++ primitive. It is stored directly using either `RWFile::Write(int)`, or `RWvostream::operator<<(int)`.

Member data `driver_` is an *RWCString*. It does *not* inherit from *RWCollectable*. It is stored using:

```
RWvostream& operator<<(RWvostream&, const RWCString&);
```

Member data `customers_` is an *RWSet*. It *does* inherit from *RWCollectable*. It is stored using:

```
RWvostream& operator<<(RWvostream&, const RWCollectable&);
```

Finally, member data `passengers_` is a little tricky. This data is a pointer to an *RWSet*, which inherits from *RWCollectable*. However, there is the possibility that the pointer is nil. If it is nil, then passing it to:

```
RWvostream& operator<<(RWvostream&, const RWCollectable&);
```

would be disastrous, as we would have to dereference `passengers_`:

```
strm << *passengers_;
```

Instead, since our class has declared `passenger_` as an *RWSet**, we pass it to:

```
RWvostream& operator<<(RWvostream&, const RWCollectable*);
```

which automatically detects the nil pointer and stores a record of it.

Virtual Functions restoreGuts(RWFile&) and restoreGuts(RWvistream&)

In a manner similar to `saveGuts()`, these virtual functions are used to restore the internal state of an *RWCollectable* from a file or stream. Here is a definition of these functions for the *Bus* class:

```
void Bus::restoreGuts(RWFile& f)
{  RWCollectable::restoreGuts(f);   // Restore base class
   f.Read(busNumber_);              // Restore primitive
   f >> driver_ >> customers_;      // Uses Rogue Wave provided
                                    // versions

   delete passengers_;              // Delete old RWSet
   f >> passengers_;                // Replace with a new one
}

void Bus::restoreGuts(RWvistream& strm)
{  RWCollectable::restoreGuts(strm);   // Restore base class
   strm >> busNumber_ >> driver_ >> customers_;

   delete passengers_;                 // Delete old RWSet
   strm >> passengers_;                // Replace with a new one
}
```

Note that the pointer `passengers_` is restored using:

```
RWvistream& operator>>(RWvistream&, RWCollectable*&);
```

If the original `passengers_` is non-nil, then this function restores a new *RWSet* off the heap and returns a pointer to it. Otherwise, it returns a nil pointer. Either way, the old contents of `passengers_` are replaced. Hence, we must call `delete passengers_` first.

Multiply-referenced Objects

A passenger name can exist in the set pointed to by `customers_` *and* in the set pointed to by `passengers_`; that is, both collections contain the same string. When the *Bus* is restored, we want to make sure that the pointer relationship is maintained, and that our restoration does not create another copy of the string.

Fortunately, we don't have to do anything special to insure that the pointer relationship stays as it should be. Consider the call:

```
Bus aBus;
RWFile aFile("busdata.dat");

aBus.addPassenger("John");
aFile << aBus;
```

Because `passenger_`is a subset of `customer_`, the function `addPassenger` puts the name on both the customer list and the passenger list. When we save `aBus` to `aFile`, *both* lists are saved in a single call: first the customer list, then the passenger list. The polymorphic persistence machinery saves the first reference to John, but for the second reference it merely stores a reference to the first copy. During the restore, both references will resolve to the same object, replicating the original morphology of the collection.

Virtual Function binaryStoreSize()

The `binaryStoreSize()` virtual function calculates the number of bytes necessary to store an object using *RWFile*. The function is:

```
virtual Rwspace  binaryStoreSize() const;
```

This function is useful for classes *RWFileManager* and *RWBTreeOnDisk*, which require allocation of space for an object before it can be stored. The non-virtual function `recursiveStoreSize()` returns the number of bytes that is actually stored. Recursive store size uses `binaryStoreSize()` to do its work.

Writing a version of `binaryStoreSize()` is usually straightforward. You just follow the pattern set by `saveGuts(RWFile&)`, except that instead of saving member data, you add up their sizes. The only real difference is a syntactic one: instead of insertion operators, you use `sizeof()` and the member functions indicated below:

- For primitives, use `sizeof()`;
- For objects that inherit from *RWCollectable*, if the pointer is non-nil, use member function:

  ```
  RWspace RWCollectable::recursiveStoreSize();
  ```

- For objects that inherit from *RWCollectable*, if the pointer is nil, use the static member function:

  ```
  RWspace RWCollectable::nilStoreSize();
  ```

- For other objects, use member function `binaryStoreSize()`.

Here's a sample definition of a `binaryStoreSize()` function for class *Bus*:

```
RWspace Bus::binaryStoreSize() const{
   RWspace count = RWCollectable::binaryStoreSize() +
     customers_.recursiveStoreSize() +
     sizeof(busNumber_) +
     driver_.binaryStoreSize();

   if (passengers_)
     count += passengers_->recursiveStoreSize();
   else
     count += RWCollectable::nilStoreSize();

   return count;
}
```

Polymorphically Persisting Custom Collections

The versions of `saveGuts()` and `restoreGuts()` that *Tools.h++* built into class *RWCollection* are sufficient for most collection classes. The function `RWCollection::saveGuts()` works by repeatedly calling:

```
RWvostream& operator<<(RWvostream&, const RWCollectable&);
```

for each item in the collection. Similarly, `RWCollection::restoreGuts()` works by repeatedly calling:

```
RWvistream& operator>>(RWvistream&, RWCollectable*&);
```

This operator allocates a new object of the proper type off the heap, then calls `insert()`. Because all of the Rogue Wave Smalltalk-like collection classes inherit from *RWCollection*, they all use this mechanism.

If you decide to write your own collection classes that inherit from class *RWCollection*, you will rarely have to define your own `saveGuts()` or `restoreGuts()`.

There are exceptions. For example, class *RWBinaryTree* has its own version of `saveGuts()`. This is necessary because the default version of `saveGuts()` stores items in order. For a binary tree, this would result in a severely

unbalanced tree when the tree was read back in—essentially, the degenerate case of a linked list. Hence, *RWBinaryTree*'s version of `saveGuts()` stores the tree level-by-level.

When you design your class, you must determine whether it has similar special requirements which may need a custom version of `saveGuts()` and `restoreGuts()`.

A Note on the RWFactory

Let's review what the *RWDEFINITION_MACRO*s look like:

```
RWDEFINE_COLLECTABLE(className, numericID)
```

or, using a string ID:

```
RWDEFINE_NAMED_COLLECTABLE(className, stringID)
```

In the `.cpp` file for the bus example, the macros appear like this:

```
RWDEFINE_COLLECTABLE(Bus, 200)
```

and:

```
RWDEFINE_NAMED_COLLECTABLE(Client, "a client")
```

Because you use these macros, a program can allow a new instance of your class to be created given only its `RWClassID`:

```
Bus* newBus = (Bus*)theFactory->create(200);
```

or its `RWStringID`:

```
Client* aClient = (Client*)theFactory->create("a client");
```

The pointer `theFactory` is a global pointer that points to a one-of-a-kind global instance of class *RWFactory*, used to hold information about all *RWCollectable* classes that have instances in the executable. The `create()` method of *RWFactory* is used internally by the polymorphic persistence machinery to create a new instance of a persisted object whose type is not known at run time. You will not normally use this capability in your own source code, because the use of *RWFactory* is generally transparent to the user. See the *Class Reference* for more details on *RWFactory*.

Summary

In general, you may not have to supply definitions for all of these virtual functions when designing your own class. For example, if you know that your class will never be used in sorted collections, then you do not need a definition for `compareTo()`. Nevertheless, it is a good idea to supply definitions for all virtual functions anyway: that's the best way to encourage code reuse!

Here then, is the complete listing for our class *Bus*:

```
BUS.H:


#ifndef __BUS_H__
#define __BUS_H__

#include <rw/rwset.h>
#include <rw/collstr.h>

class Bus : public RWCollectable  {
  RWDECLARE_COLLECTABLE(Bus)

public:

  Bus();
  Bus(int busno, const RWCString& driver);
  ~Bus();

  // Inherited from class "RWCollectable":
  Rwspace      binaryStoreSize() const;
  int          compareTo(const RWCollectable*) const;
  RWBoolean    isEqual(const RWCollectable*) const;
  unsigned     hash() const;
  void         restoreGuts(RWFile&);
  void         restoreGuts(RWvistream&);
  void         saveGuts(RWFile&) const;
  void         saveGuts(RWvostream&) const;

  void         addPassenger(const char* name);
  void         addCustomer(const char* name);
  size_t       customers() const;
```

```
  size_t       passengers() const;
  RWCString    driver() const     {return driver_;}
  int          number() const     {return busNumber_;}

private:

  RWSet        customers_;
  RWSet*       passengers_;
  int          busNumber_;
  RWCString    driver_;
};

class Client : public RWCollectable  {
  RWDECLARE_COLLECTABLE(Client)
  Client();
  Client(const char* name);
  Rwspace      binaryStoreSize() const;
  int          compareTo(const RWCollectable*) const;
  RWBoolean    isEqual(const RWCollectable*) const;
  unsigned     hash() const;
  void         restoreGuts(RWFile&);
  void         restoreGuts(RWvistream&);
  void         saveGuts(RWFile&) const;
  void         saveGuts(RWvostream&) const;
private:
  RWCString    name_;
//ignore other client information for this example
};
#endif
```

BUS.CPP:

```
#include "bus.h"
#include <rw/pstream.h>
#include <rw/rwfile.h>
#ifdef __GLOCK__
#  include <fstream.hxx>
#else
#  include <fstream.h>
```

```
#endif

RWDEFINE_COLLECTABLE(Bus, 200)

Bus::Bus() :
  busNumber_  (0),
  driver_     ("Unknown"),
  passengers_ (rwnil)
{}

Bus::Bus(int busno, const RWCString& driver) :
  busNumber_  (busno),
  driver_     (driver),
  passengers_ (rwnil)
{}

Bus::~Bus()  {
  customers_.clearAndDestroy();
  delete passengers_;
}

RWspace
Bus::binaryStoreSize() const  {
  RWspace count = RWCollectable::binaryStoreSize() +
    customers_.recursiveStoreSize() +
    sizeof(busNumber_) +
    driver_.binaryStoreSize();

  if (passengers_)
    count += passengers_->recursiveStoreSize();

  return count;
}

int
Bus::compareTo(const RWCollectable* c) const  {
  const Bus* b = (const Bus*)c;
  if (busNumber_ == b->busNumber_) return 0;
  return busNumber_ > b->busNumber_ ? 1 : -1;
}

RWBoolean
Bus::isEqual(const RWCollectable* c) const  {
```

```
  const Bus* b = (const Bus*)c;
  return busNumber_ == b->busNumber_;
}

unsigned
Bus::hash() const  {
  return (unsigned)busNumber_;
}

size_t
Bus::customers() const  {
  return customers_.entries();
}

size_t
Bus::passengers() const    return passengers_ ? passengers_-
>entries() : 0;
}

void
Bus::saveGuts(RWFile& f) const  {
  RWCollectable::saveGuts(f);    // Save base class
  f.Write(busNumber_);           // Write primitive directly
  f << driver_ << customers_;    // Use Rogue Wave provided
versions
  f << passengers_;       // Will detect nil pointer automatically
}

void
Bus::saveGuts(RWvostream& strm) const  {
  RWCollectable::saveGuts(strm);   // Save base class
  strm << busNumber_;              // Write primitives directly
  strm << driver_ << customers_;   // Use Rogue Wave
                                   // provided versions
  strm << passengers_;    // Will detect nil pointer automatically
}

void Bus::restoreGuts(RWFile& f)  {
  RWCollectable::restoreGuts(f);  // Restore base class
  f.Read(busNumber_);             // Restore primitive
  f >> driver_ >> customers_;     // Uses Rogue Wave
                                  // provided versions
```

```
  delete passengers_;             // Delete old RWSet
  f >> passengers_;               // Replace with a new one
}

void Bus::restoreGuts(RWvistream& strm)  {
  RWCollectable::restoreGuts(strm); // Restore base class
  strm >> busNumber_ >> driver_ >> customers_;

  delete passengers_;             // Delete old RWSet
  strm >> passengers_;            // Replace with a new one
}

void
Bus::addPassenger(const char* name)  {
  Client* s = new Client(name);
  customers_.insert( s );

  if (!passengers_)
    passengers_ = new RWSet;

  passengers_->insert(s);
}

void
Bus::addCustomer(const char* name)  {
  customers_.insert( new Client(name) );
}

///////////////    Here are Client methods //////////////
RWDEFINE_NAMED_COLLECTABLE(Client,"client")

Client::Client() {} // Uses RWCString default constructor

Client::Client(const char* name) : name_(name) {}

RWspace
Client::binaryStoreSize() const {
  return name_->binaryStoreSize();
}

int
Client::compareTo(const RWCollectable* c) const  {
  return name_.compareTo(((Client*)c)->name_);
}
```

```
RWBoolean
Client::isEqual(const RWCollectable* c) const  {
  return name_ == *(Client*)c;
}

unsigned
Client::hash() const  {
  return name_.hash();
}

void
Client::restoreGuts(RWFile& f)  {
  f >> name_;
}

void
Client::restoreGuts(RWvistream& vis)  {
  vis >> name_;
}

void
Client::saveGuts(RWFile& f) const  {
  f << name_;
}

void
Client::saveGuts(RWvostream& vos) const  {
  vos << name_;
}


main()  {
  Bus theBus(1, "Kesey");
  theBus.addPassenger("Frank");
  theBus.addPassenger("Paula");
  theBus.addCustomer("Dan");
  theBus.addCustomer("Chris");

  { // block controls lifetime of stream
    ofstream f("bus.str");
    RWpostream stream(f);
    stream << theBus;      // Persist theBus to an ASCII stream
  }
```

```
  {
    ifstream f("bus.str");
    RWpistream stream(f);
    Bus* newBus;
    stream >> newBus;      // Restore it from an ASCII stream

    cout << "Bus number " << newBus->number()
         << " has been restored; its driver is "
         << newBus->driver() << ".\n";
    cout << "It has " << newBus->customers()
         << " customers and "
        << newBus->passengers() << " passengers.\n\n";

    delete newBus;
  }

  return 0;
}
```

Program Output:

```
Bus number 1 has been restored; its driver is KeseyKesey.
It has 4 customers and 2 passengers.
```

15

Internationalization 16

As a developer, you belong to an international community. Whatever your nation, you share with other developers the problems of adapting software and applications for your own culture and others. Accommodating the needs of users in different cultures is called *localization*; making software easily localized is called *internationalization*.

Tools.h++ is made in the United States. It is *internationalized* in the sense that it provides the framework you need to *localize* fundamental aspects of different cultures, such as alphabets, languages, currencies, numbers, and date- and time-keeping notations. With *Tools.h++*, you write a single application you can ship to any country. When your application is executed, it will be able to process times, dates, strings, and currency in the native format.

While some aspects of internationalization are limited, a useful feature of *Tools.h++* is that it imposes no policy. *Tools.h++* gives you the freedom and flexibility to design your application to meet the needs of your clients' cultures and your own.

Localizing Alphabets with RWCString and RWWString

Localizing alphabets begins with allowing them to be represented. As mentioned in "Eight-bit Clean" on page 18, *Tools.h++* code is "8-bit clean" to accommodate the extended character set. All of the English alphabet is described in 7 bits, leaving the eighth free for umlauts, cedillas, and other

diacritical marks and special characters. And because even 8 bits often isn't enough to represent all the character glyphs of various languages, *Tools.h++* also allows two kinds of extensions: multibyte and wide-character encodings.

Multibyte encodings use a sequence of one or more bytes to represent a single character. (Typically the ASCII characters are still one byte long.) These encodings are compact, but may be inconvenient for indexing and substring operations. Wide character encodings, in contrast, place each character in a 16- or 32-bit integral type called a `wchar_t`, and represent a string as an array of `wchar_t`. Usually it is possible to translate a string encoded in one form into the other.

Tools.h++ two efficient string types, *RWCString* and *RWWString*, were discussed in Chapter 3. *RWCString* represents strings of 8-bit chars, with some support for multibyte strings. *RWWString* represents strings of `wchar_t`. Both provide access to Standard C Library support for local collation conventions with the member function `collate()` and the global function `strXForm()`. In addition, the library provides conversions between wide and multibyte representations. The wide- and multibyte-character encodings used are those of the host system.

But representation of alphabets can be even more complex. For example, is a character upper case, lower case, or neither? In a sorted list, where do you put the names that begin with accented letters? What about Cyrillic names? How are wide-character strings represented on byte streams? Standards bodies and corporate labs are addressing these issues, but the results are not yet portable. For the time being, *Tools.h++* strives to make best use of what they provide.

Localizing Messages

To accommodate a user's language, a program must display titles, menu choices, and status messages in that language. Usually such text is stored in a message catalog or resource file, separate from program code, so it may be easily edited or replaced. *Tools.h++* does not display titles or menus directly, but does return status messages when errors occur. By default, *Tools.h++* makes no attempt to localize these messages. Instead, it provides an optional facility that allows error messages to be retrieved from your own catalog.

The facility can be used in one of four modes:

Mode	**Define**
No messaging	`RW_NOMSG`

Use `catgets()`	`RW_CATGETS`
Use `gettext()`	`RW_GETTEXT`
Use `dgettext()`	`RW_DGETTEXT`

These localization techniques and their documentation are specific to your platform. Once you discover what your system provides, you specify that mode for *Tools.h++* by setting the appropriate switch in `<rw/compiler.h>` before compiling the library. If you have object code, this choice has already been made for you.

Function `catgets()` uses both a message set number and a message number within that set to look up a localized version of a message. The number for the message set to use is defined in the macro `RW_MESSAGE_SET_NUMBER` found in `<rw/compiler.h>`. Function `gettext()` uses the message itself. The messages and their respective message numbers are given in Appendix C.

You will find information on using `catgets()`, `gettext()`, and `dgettext()`in the documentation that comes with your compiler.

Challenges of Localizing Currencies, Numbers, Dates, and Times

If you write applications for cultures other than your own, you will soon confront the challenges of representing currencies, numbers, dates, and times. Currencies vary in both unit value and notation. Numbers are written differently; for example, Europe and the United States use periods and commas in opposite ways. Often a program must display values in notations customary to both vendor and customer.

Scheduling, a common software function, involves time and calendar calculations. Local versions of the Gregorian calendar use different names for days of the week and months, and different ordering for the components of a date. Time may be represented according to a 12- or 24-hour clock, and further complicated by time zone conventions, like daylight-saving time (DST), that vary from place to place, or even year to year.

The Standard C Library provides `<locale.h>` to accommodate some of these different formats, but it is incomplete. It offers no help for conversion from strings to these types, and almost no help for conversions involving two or more locales. Common time zone facilities, such as those defined in POSIX.1 (see the *Appendix*), are similarly limited, usually offering no way to compute wall clock time for other locations, or even for the following year in the same location.

16

RWLocale and RWZone

Tools.h++ addresses these problems with the abstract classes *RWLocale* and *RWZone*. If you have used *RWDate*, you have already used *RWLocale*, perhaps unknowingly. Every time you convert a date or time to or from a string, a default argument carries along an *RWLocale* reference. Unless you change it, this is a reference to a global instance of a class derived from *RWLocale* at program startup to provide the effect of a C locale. To use *RWLocale* explicitly, you can construct your own instance and pass it in place of the default. Similarly, when you manipulate times, you can substitute your own instance for the default *RWZone* reference.

You can also install your own instance of *RWLocale* or *RWZone* as the global default. Many streams even allow you to install your *RWLocale* instance in the stream so that dates and times transferred on and off that stream are formatted or parsed accordingly, without any special arguments. This is called *imbuing the stream*, a process described in more detail in the next section.

In the following sections, let us look at some examples of how to localize various data using *RWLocale* and *RWZone*. Let us begin by constructing a date, today's date:

```
RWDate today = RWDate::now();
```

We can display it the usual way using ordinary C-locale conventions:

```
cout << today << endl;
```

But what if you're outside your home locale? Or perhaps you have set your environment variable `LANG` to `fr`[1], because you want French formatting. To display the date in your preferred format, you construct an *RWLocale* object:

```
RWLocale& here = *new RWLocaleSnapshot("");
```

Class *RWLocaleSnapshot* is the main implementation of the interface defined by *RWLocale*. It extracts the information it needs from the global environment during construction with the help of such Standard C Library functions as `strftime()` and `localeconv()`. The most straightforward way to use *RWLocaleSnapshot* is to pass it directly to the *RWDate* member function `asString()`[2]:

```
cout << today.asString('x', here) << endl;
```

1. Despite the existing standard for locale names, many vendors provide variant naming schemes. Check your vendor's documentation for details.
2. The first argument of the function `asString()` is a character, which may be any of the format options supported by the Standard C Library function `strftime()`.

There is, however, a more convenient way. You can install `here` as the global default locale so the insertion operator will use it:

```
RWLocale::global(&here);
cout << today << endl;
```

Dates

Now suppose you are American and want to format a date in German, but don't want German to be the default. Construct a German locale:

```
RWLocale& german = *new RWLocaleSnapshot("de");  //See footnote
1
```

You can format the same date for both local and German readers as follows:

```
cout << today << endl
     << today.asString('x', german) << endl;
```

See the definition of `x` in the entry for *RWLocale* in the *Class Reference*.

Would you like to read in a German date string? Again, the straightforward way is to call everything explicitly:

```
RWCString str;
cout << "enter a date in German: " << flush;
str.readLine(cin);
today = RWDate(str, german);
if (today.isValid())
  cout << today << endl;
```

Sometimes, however, you would prefer to use the extraction operator `>>`. Since the operator must expect a German-formatted date, and know how to parse it, you pass this information along by imbuing a stream with the German locale.

The following code snippet imbues the stream `cin` with the German locale, reads in and converts a date string from German, and displays it in the local format:

```
german.imbue(cin);
cout << "enter a date in German: " << flush;
cin >> today;  // read a German date!
if (today.isValid())
  cout << today << endl;
```

Imbuing is useful when many values must be inserted or extracted according to a particular locale, or when there is no way to pass a locale argument to the point where it will be needed. By using the static member function `RWLocale::of(ios&)`, your code can discover the locale imbued in a stream. If the stream has not yet been imbued, `of()` returns the current global locale.[1]

The interface defined by *RWLocale* handles more than dates. It can also convert times, numbers, and monetary values to and from strings. Each has its complications. Time conversions are complicated by the need to identify the time zone of the person who entered the time string, or the person who will read it. The mishmash of daylight-saving time jurisdictions can magnify the difficulty. Numbers are somewhat messy to format because their insertion and extraction operators (`<<` and `>>`) are already defined by `<iostream.h>`. For money, the main problem is that there is no standard internal representation for monetary values. Fortunately, none of these problems is overwhelming with *Tools.h++*

Time

Let us first consider the time zone problem. We can easily see that there is no simple relationship between time zones and locales. All of Switzerland shares a single time zone, including daylight-saving time (DST) rules, but has four official languages: French, German, Italian, and Romansch. On the other hand, Hawaii and New York share a common language, but occupy time zones five hours apart—sometimes six hours apart, because Hawaii does not observe DST. Furthermore, time zone formulas have little to do with cultural formatting preferences. For these reasons, *Tools.h++* uses a separate time zone object, rather than letting *RWLocale* subsume time zone responsibilities.

In *Tools.h++*, the class *RWZone* encapsulates knowledge about time zones. It is an abstract class, with an interface implemented in the class *RWZoneSimple*. Three instances of *RWZoneSimple* are constructed at startup to represent local wall clock time, local Standard time, and Universal time (GMT). Local wall clock time includes any DST in use. Whenever you convert an absolute time to or from a string, as in the class *RWTime*, an instance of *RWZone* is involved. By default, the local time is assumed, but you can pass a reference to any *RWZone* instance.

1. You can restore a stream to its unimbued condition with the static member function `RWLocale::unimbue(ios&)`; note that this is not the same as imbuing it with the current global locale.

It's time for some examples! Imagine you had scheduled a trip from New York to Paris. You were to leave New York on December 20, 1993, at 11:00 p.m., and return on March 30, 1994, leaving Paris at 5:00 a.m., Paris time. What will the clocks show at your destination when you arrive?

First, let's construct the time zones and the departure times:

```
RWZoneSimple newYorkZone(RWZone::USEastern, RWZone::NoAm);
RWZoneSimple parisZone  (RWZone::Europe,    RWZone::WeEu);
RWTime leaveNewYork(RWDate(20, 12, 1993), 23,00,00, newYorkZone);
RWTime leaveParis  (RWDate(30,  3, 1994), 05,00,00, parisZone);
```

The flight is about seven hours long each way, so:

```
RWTime arriveParis  (leaveNewYork + long(7 * 3600));
RWTime arriveNewYork(leaveParis   + long(7 * 3600));
```

Now let's display the arrival times and dates according to their respective local conventions, French in Paris and American English in New York:

```
RWLocaleSnapshot french("fr");            // or vendor specific
cout << "Arrive' au Paris a` "
     << arriveParis.asString('c', parisZone, french)
     << ", heure local." << endl;
cout << "Arrive in New York at "
     << arriveNewYork.asString('c', newYorkZone)
     << ", local time." << endl;
```

The code works even though your flight crosses several time zones and arrives on a different day than it departed; even though, on the day of the return trip in the following year, France has already begun observing DST, but the U.S. has not. None of these details is visible in the example code above–they are handled silently and invisibly by *RWTime* and *RWZone*.

All this is easy for places that follow *Tools.h++* built-in DST rules for North America, Western Europe, and "no DST". But what about places that follow other rules, such as Argentina, where spring begins in September and summer ends in March? *RWZoneSimple* is table-driven; if the rule is simple enough, you can construct your own table of type *RWDaylightRule*, and specify it as you construct an *RWZoneSimple*. For example, imagine that DST begins at 2 a.m. on the last Sunday in September, and ends the first Sunday in March. Simply create a static instance of *RWDaylightRule*:

```
static RWDaylightRule sudAmerica =
   { 0, 0, TRUE, {8, 4, 0, 120}, {2, 0, 0, 120}};
```

(See the documentation for *RWZoneSimple* for details on what the numbers mean.) Then construct an *RWZone* object:

```
RWZoneSimple  ciudadSud( RWZone::Atlantic, &sudAmerica );
```

Now you can use `ciudadSud` just like you used `paris` or `newYork` above.

But what about places where the DST rules are too complicated to describe with a simple table, such as Great Britain? There, DST begins on the morning after the third Saturday in April, unless that is Easter, in which case it begins the week prior! For such jurisdictions, you might best use standard time, properly labeled. If that just won't do, you can derive from *RWZone* and implement its interface for Britain alone. This strategy is much easier than trying to generalize a case to handle all possibilities including Britain, and it's smaller and faster besides.

The last time problem we will discuss here is that there is no standard way to discover what DST rules are in force for any particular place. In this the Standard C Library is no help; you must get the information you need from the local environment your application is running on, perhaps by asking the user.

One example of this problem is that the local wall clock time *RWZone* instance is constructed to use North American DST rules, if DST is observed at all. If the user is not in North America, the default local time zone probably performs DST conversions wrong, and you must replace it. If you are a user in Paris, for example, you could solve this problem as follows:

```
RWZone::local(new RWZoneSimple(RWZone::Europe, RWZone::WeEu));
```

If you look closely into `<rw/locale.h>`, you will find that *RWDate* and *RWTime* are never mentioned. Instead, *RWLocale* uses the Standard C Library type `struct tm`. *RWDate* and *RWTime* both provide conversions to this type, and you may prefer using it directly rather than using `RWTime::asString()`. For example, suppose you must write out a time string containing only hours and minutes; e.g.,`12:33`. The standard formats defined for `strftime()`and implemented by *RWLocale* don't include that option, but you can fake it. Here's one way:

```
RWTime now = RWTime::now();
cout << now.hour() << ":" << now.minute() << endl;
```

Without using various manipulators, this code might produce a string like `9:5`. Here's another option:

```
RWTime now = RWTime::now();
cout << now.asString('H') << ":" << now.asString('M') << endl;
```

This produces `09:05`.

In each of the previous examples, `now` is disassembled into component parts twice, once to extract the hour and once to extract the minute. This is an expensive operation. If you expect to work often with the components of a time or date, you may be better off disassembling the time only once:

```
RWTime now = RWTime::now();
struct tm tmbuf;
now.extract(&tmbuf);
const RWLocale& here = RWLocale::global();        // the default
                                                 // global locale
cout << here.asString(&tmbuf, 'H') << ":"
     << here.asString(&tmbuf, 'M'); << endl;
```

Please note that if you work with years before 1901 or after 2037, you can't use *RWTime* because it does not have the required range.[1] You can use *RWLocale* to perform conversions for any time or date because `struct tm` operations are not so restricted.

Numbers

RWLocale also provides you with an interface for conversions between strings and numbers, both integers and floating point values. *RWLocaleSnapshot* implements this interface, providing the full range of capabilities defined by the Standard C Library type `struct lconv`. The capabilities include using appropriate digit group separators, decimal "point", and currency notation. When converting from strings, *RWLocaleSnapshot* allows and checks the same digit group separators.

Unfortunately, stream operations of this class are clumsier than we might like, since the standard iostream library provides definitions for number insertion and extraction operators which cannot be overridden. Instead, we can use *RWCString* functions directly:

```
RWLocaleSnapshot french("fr");
double f = 1234567.89;
long i = 987654;
RWCString fs = french.asString(f, 2);
RWCString is = french.asString(i);
if (french.stringToNum(fs, &f) &&
```

1. Of course, if you are working on a 64-bit system, there is no practical upper limit to the dates you can use.

```
      french.stringToNum(is, &i))           // verify conversion
    cout << "C:\t" << f << "\t" << i << endl
         << "French:\t" << fs << "\t" << is << endl;
```

Since the French use periods for digit group separators, and commas to separate the integer from the fraction part of a number, this code might display as:

```
C:      1.234567e+07     987654
French: 1.234.567,89     987.654
```

You will notice that numbers with digit group separators are easier to read.

Currency

Currency conversions are trickier than number conversions, mainly because there is no standard way to represent monetary values in a computer. We have adopted the convention that such values represent an integral number of the smallest unit of currency in use. For example, to represent a balance of $10.00 in the United States, you could say:

```
double sawbuck = 1000.;
```

This representation has the advantages of wide range, exactness, and portability. Wide range means you can *exactly* represent values from $0.00 up to and beyond $10,000,000,000,000.00—larger than any likely budget. Exactness means that, representing monetary values without fractional parts, you can perform arithmetic on them and compare the results for equality:

```
double price = 999.;                                    // $9.99
double penny = 1.;                                      //  $.01
assert(price + penny == sawbuck);
```

This would not be possible if the values were naively represented, as in `price = 9.99;`.

Portability means simply that `double` is a standard type, unlike common 64-bit integer or BCD representations. Of course, you can perform financial calculations on such other representations, but because you can always convert between them and `double`, they are all supported. In the future, *RWLocale* may directly support some other common representations as well.

Consider the following examples of currency conversions:

```
const RWLocale& here = RWLocale::global();
double sawbuck = 1000.;
RWCString tenNone  = here.moneyAsString(sawbuck, RWLocale::NONE);
RWCString tenLocal = here.moneyAsString(sawbuck,RWLocale::LOCAL);
RWCString tenIntl  = here.moneyAsString(sawbuck, RWLocale::INTL);
if (here.stringToMoney(tenNone,  &sawbuck) &&
    here.stringToMoney(tenLocal, &sawbuck) &&
    here.stringToMoney(tenIntl,  &sawbuck))  // verify conversion
  cout << sawbuck  << "  " << tenNone << "  "
       << tenLocal << "  " << tenIntl << "  " << endl;
```

In a United States locale, the code would display as:

```
1000.00000  10.00  $10.00  USD 10.00
```

A Note on Setting Environment Variables

As mentioned in the section on class *RWTime*, some compilers and operating systems, including the Windows operating systems, require you to set certain environment variables in order for a locale feature to work. Failure to do this can lead to great difficulties.

If you use Borland, MetaWare, Microsoft, Symantec, or Watcom, you must set your environment variable `TZ` to the appropriate time zone:

```
set TZ=PST8PDT
```

Check the documentation for your compiler and operating system for information on setting environment variables.

16

Error Handling 17

Thinking about error handling is like anticipating root canal work—it's a messy, often unpredictable, and painful topic, one we prefer to avoid. Yet think about error handling we must, in order to write robust code.

The Rogue Wave class libraries all use the same extensive and complete error handling facility. In this model, errors are divided into two broad categories: internal and external. *Internal errors*, which are further classified as either recoverable or non-recoverable, are due to errors in the internal logic of the program. As you might expect, they can be difficult to recover from and, indeed, the common default response is to abort the program. *External errors* are due to events beyond the scope of the program. Any non-trivial program should be prepared to recover from an external error.

The next section presents a table that summarizes the Rogue Wave error model. The sections following the table discuss the types of errors in more detail.

17

The Tools.h++ Error Model

The following table categorizes and describes errors in the *Tools.h++* error model. You can use the table as a quick overview, and return to it as a reference.

Table 17-1 Comparison of Error Types in Tools.h++

Error Type:	**Internal**		**External**
	Non-recoverable	**Recoverable**	
Cause:	Faulty logic or coding in the program	Faulty logic or coding in the program	Events beyond the scope of the program
Examples:	Bounds error; inserting a null pointer into a collection	Bounds error in a linked list; attempt to use an invalid date	Attempt to write a bad date, or to invert a singular matrix; stream write error; out of memory
Predictable?	Yes	Yes	No
Cost to detect:	High	Low	Low
Level of abstraction:	Low	Low	High
Where detected:	Debug version of library	Debug and production version of library	Debug and production version of library
Response:	No recovery mechanism	Throw an exception inheriting from *RWInternalErr*	Throw an exception inheriting from *RWExternalErr*, or provide test for object validity

Internal Errors

Internal errors are due to faulty logic or coding in the program. Common types of internal errors include:

- Bounds errors;
- Inserting a null pointer into a collection;
- Attempting to use a bad date.

All of these errors should be preventable. For example, you always know the permissible range of indices for an array, so you can probably avoid a bounds error. You would correct your program's use of a bad date as an obvious logic error.

Internal errors can be further classified according to the cost of error detection, and whether or not the error will be detected at run time. The two categories are:

- Non-recoverable internal errors;
- Recoverable internal errors.

Non-recoverable Internal Errors

Non-recoverable internal errors share the following distinguishing characteristics. They are:

- Easily predicted in advance;
- Encountered at relatively low levels;
- Costly to detect;
- Detected only in the debug version of the library.

Non-recoverable internal errors by definition have no recovery mechanism. Examples of these errors include bounds errors, and inserting a null pointer into a collection.

Why does a library define some errors as unrecoverable? Because detecting errors takes time. For performance reasons, a library demands some minimal level of correctness on the part of your program, and pitches anything that falls short. Errors are non-recoverable in the sense that the production version of the library has no mechanism for detecting such errors and, hence, no opportunity for recovering from them.

Bounds errors are non-recoverable because the cost of checking to make sure an index is in range can well exceed the cost of the array access itself. If a program does a lot of array accesses, checking every one may result in a slow program. To avoid this, the library may require you to always use a valid index. Because a minimum level of correctness is demanded, non-recoverable errors are simple in concept and relatively easy to avoid.

You can best discover and eliminate non-recoverable errors by compiling and linking your application with the debug version of the library. See "The Debug Version of Tools.h++" on page 236 for details. The debug version includes lots

of extra checks designed to uncover coding errors. Some of these checks may take extra time, or even cause debug messages to be printed out, so you will want to compile and link with the production version for an efficient final product.

If the debug version of the library discovers an error, it typically aborts the program.

Recoverable Internal Errors

Recoverable internal errors are similar to their non-recoverable relatives in that they are easy to predict and occur at low levels. They differ in that they are:

- Not costly to detect;
- Detected in both the debug *and* the production versions of the library.

A bounds error in a linked list or an attempt to use an invalid date are both examples of recoverable internal errors. The library's response to these errors is to throw an exception inheriting from *RWInternalErr*.

The production version of the library can check for recoverable internal errors because the cost is relatively low. For example, to find a bounds error in a linked list, the cost of walking the list will far exceed the cost of detecting whether the index is in bounds. Hence, you can afford to check for a bounds error on every access.

If an error is discovered, the library will throw an exception inheriting from *RWInternalErr*, as we have mentioned. Here's what it looks like when *Tools.h++* throws an exception:

```
// Find link "i"; the index must be in range:
RWIsvSlink* RWIsvSlist::at(size_t i) const
{
   if (i >= entries()){
     if(RW_NPOS == i)
       RWTHROW( RWBoundsErr( RWMessage( RWTOOL_NPOSINDEX)));
     else
       RWTHROW( RWBoundsErr( RWMessage( RWTOOL_INDEXERR,
               (unsigned)i,
               (unsigned)entries()) ));
   }

   register RWIsvSlink* link = head_.next_;
```

```
   while (i--) link = link->next_;
   return link;
}
```

In this code, note how the function *always* attempts to detect a bounds error. If it finds one, it throws an instance of *RWBoundsErr*, a class that inherits from *RWInternalErr*. This instance contains an internationalized message, discussed in “Localizing Messages” on page 218. The `RWTHROW` macro is discussed in Section 19.3.1.

Throwing an exception gives you the opportunity to catch and possibly recover the exception. However, because the internal logic of the program has been compromised, most likely you will want to attempt to save the document you are working on, and abort the program.

External Errors

External errors are due to events beyond the scope of the program. As we mentioned in the introduction, any non-trivial program should be prepared to recover from an external error. In general, external errors are:

- Not easily predicted in advance;
- Encountered at more abstract levels;
- Not costly to detect;
- Detected in both the production and the debug versions of the library.

Examples of external errors would include: an attempt to set a bad date, such as 31 June 1992; an attempt to invert a singular matrix; a stream write error; being out of memory. *Tools.h++* would respond to these by throwing an exception inheriting from *RWExternalErr*, or providing a test for object validity.

External errors may be run time errors. In an object-oriented environment, run time errors frequently show up as an attempt to set an object into an invalid state, perhaps as a result of invalid user input. The example given above, initializing a date object with the nonexistent date 31 June 1992, is an external run time error.

External errors often take the form of exceptions thrown by the operating system. *Tools.h++* takes responsibility for detecting these exceptions and recovering the resources it has acquired; it will close files and restore heap memory. As the user, however, you are responsible for all resources acquired by code external to the *Tools.h++* library during these kinds of exceptions. Generally, *Tools.h++* assumes that these exceptions will not be thrown during

memory-related C library calls such as `memcpy`. *Tools.h++* make every effort to detect throws which occur during operations on *Tools.h++* objects or user-defined objects.

In theory, the *Tools.h++* response to an external error is to throw an exception, or to provide a test for object validity. It should never abort the program. In practice, however, some compilers do not handle exceptions, so *Tools.h++* provides an opportunity to recover in an error handler, or to test for a status value. Here is an example of using the `isValid` function to validate user input:

```
RWDate date;

while (1) {
   cout << "Give a date: ";
   cin >> date;
   if (date.isValid()) break;
   cout << "You entered a bad date; try again\n";
}
```

Exception Architecture

When an exception is thrown a *throw operand* is passed. The type of the throw operand determines which handlers can catch it. *Tools.h++* uses the following hierarchy for throw operands:

xmsg

- *RWxmsg*
  - *RWInternalErr*
    - *RWBoundsErr*
  - *RWExternalErr*
    - *RWFileErr*
    - *RWStreamErr*
- *xalloc*
  - *RWxalloc*

As you can see, the hierarchy parallels the error model outlined in “The Tools.h++ Error Model” on page 230. This hierarchy assumes the presence of class `xmsg`, nominally provided by your compiler vendor. This is the class now being considered for standardization by the Library Working Group of the

C++ Standardization Committee X3J16 (Document 92-0116). If your compiler does not come with versions of `xmsg` and `xalloc`, the Rogue Wave classes *RWxmsg* and *RWxalloc* will emulate them for you.

Class `xmsg` carries a string that can be printed out at the catch site to give the user some idea of what went wrong. This string is formatted and internationalized as described in "Localizing Messages" on page 218.

Error Handlers

Tools.h++ uses the macro `RWTHROW` to throw an exception. If your compiler supports exceptions, this macro resolves by calling a function which throws the exception. If your compiler does not support exceptions, the macro resolves to call an error handler with prototype:

```
void errHandler(const RWxmsg&);
```

The default error handler aborts the program. You can change the default handler with the function:

```
typedef void (*rwErrHandler)(const RWxmsg&);
rwErrHandler rwSetErrHandler(rwErrHandler);
```

The next example demonstrates how a user-defined error handler works in a compiler that doesn't support exceptions:

```
#include <rw/rwerr.h>
#include <rw/coreerr.h>
#include <iostream.h>

#ifdef RW_NO_EXCEPTIONS

void myOwnErrorHandler(const RWxmsg& error){
  cout << "myOwnErrorHandler(" << error.why() << ")" << endl;
}

int main(){
  rwSetErrHandler(myOwnErrorHandler);  // Comment out this line
                                       // to get the default error handler.
  RWTHROW( RWExternalErr(RWMessage( RWCORE_GENERIC, 12345, "Howdy!") ));
  cout << "Done." << endl;
  return 0;
}

#else  //RW_NO_EXCEPTIONS

#error This example only for compilers without exception handling

#endifKJKevin "Rudi" Johnsrude
```

The Debug Version of Tools.h++

You can build the *Tools.h++* library in a debug mode, and gain a very powerful tool for uncovering and correcting internal errors in your code.

To build a debug version of the library, you must compile the *entire* library with the preprocessor flag `RWDEBUG` defined. *You must compile the entire library and application with a single setting of the flag—either defined or not defined.* The resultant library and program will be slightly larger and slightly slower. See the appropriate `makefile` for additional directions.

The flag RWDEBUG activates a set of PRECONDITION and POSTCONDITION clauses at the beginning and end of critical functions. These pre- and postconditions are implemented with asserts. A failure will cause the program to halt after first printing out the offending condition, along with the file and line number where it occurred.

RWPRECONDITION and RWPOSTCONDITION

In his landmark book *Object-oriented Software Construction*, Bertrand Meyer suggests regarding functions as a contract between a caller and a callee. If the caller agrees to abide by a set of preconditions, the callee guarantees to return results that satisfy a set of postconditions. The following comparison shows the usefulness of Meyer's paradigm in *Tools.h++*. Let's look first at a bounds error in C:

```
char buff[20];
char j = buff[20];                                  // Bounds error!
```

Such a bounds error is extremely tough to detect in C, but easy in C++, as shown below:

```
RWCString buff(20);
char j = buff[20];                        // Detectable bounds error
```

The bounds error is easy to detect in C++ because the operator[] can be overloaded to perform an explicit bounds check. In other words, when the flag RWDEBUG is set, operator[] also executes the PRECONDITION clause, as below:

```
char& RWCString::operator[](size_t I){
   RWPRECONDITION(i < length() );
   return rep[i];
}
```

The case just described would trigger a failure because operator[]would find that the PRECONDITION is not met.

Here's a slightly more complicated example:

```
template <class T> void List::insert(T* obj){
   RWPRECONDITION( obj!= 0 );
   head = new Link(head, obj);
   RWPOSTCONDITION( this->contains(obj) );
}
```

In this example, the job of the function `insert()` is to insert the object pointed to by the argument into a linked list of pointers to objects of type T. The only precondition for the function to work is that the pointer `obj` not be null. If this condition is satisfied, then the function guarantees to successfully insert the object. The condition is checked by the postcondition clause.

The macros `RWPRECONDITION` and `RWPOSTCONDITION` are defined in `<rw/defs.h>` and compile out to no-ops, so long as the preprocessor macro `RWDEBUG` is not defined. Here's what appears in the `makefile`:

```
#ifdef RWDEBUG
#  define RWPRECONDITION(a)      assert(a)
#  define RWPOSTCONDITION(a)     assert(a)
#else
#  define RWPRECONDITION(a)      ((void*)0)
#  define RWPOSTCONDITION(a)     ((void*)0)
#endif
```

Advanced Topics 18

You'll probably come to this section on Advanced Topics after you've had some experience with *Tools.h++*. You don't need all the information contained here to start using the library effectively, but you should read it to expand your knowledge of specific topics, to see how we've implemented various product features, and to check for new information.

Dynamic Link Library

The *Tools.h++*Class Library can be linked as a Microsoft Windows 3.X Dynamic Link Library (DLL). In a DLL, linking occurs at *run time* when the routine is actually used. This results in much smaller executables because routines are pulled in only as needed. Another advantage is that many applications can share the same code, rather than duplicating it in each application.

Because Windows and C++ technology is evolving so rapidly, be sure to check the file `TOOLDLL.DOC` on your distribution disk for more updates.

The DLL Example

This section discusses a sample Windows application, `DEMO.CPP`, that uses the *Tools.h++* DLL. The code for this program can be found in the subdirectory `DLLDEMO`. This program falls into the familiar category of "Hello World" examples.

18

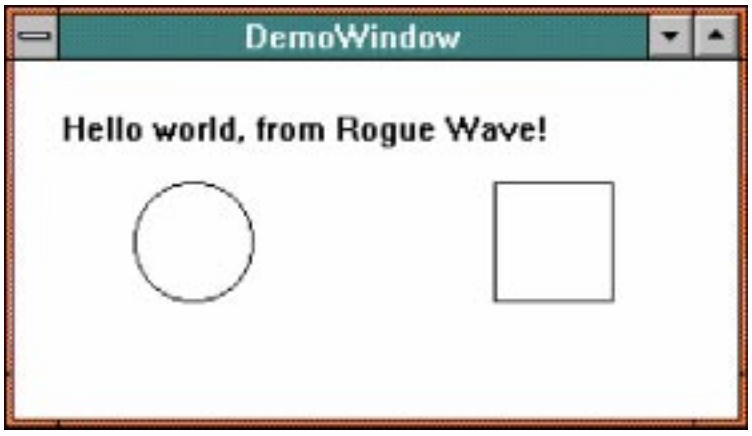


Figure 18-1 The demo program window.

The program is somewhat unusual, however, in that it maintains a linked list of *Drawable* objects you can insert into the window at run time. You can find the list, which is implemented using class *RWSlistCollectables*, in the subdirectory `DLLDEMO`. The discussion in this section assumes that you are somewhat familiar with Windows 3.X programming, but not with its relationship to C++.

The DEMO.CPP Code

Here's the main program of `DEMO.CPP`, the sample Windows application that uses the *Tools.h++* DLL.

```
/*
 * Sample Windows 3.X program, using the Tools.h++ DLL.
 */

#include "demowind.h"

int PASCAL
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR , int nCmdShow)                                  // 1
{  // Create an instance of the "DemoWindow" class:
   DemoWindow ww(hInstance, hPrevInstance, nCmdShow);          // 2

   // Add some items to it:
   ww.insert( new RWRectangle(200, 50, 250, 100));             // 3
   ww.insert( new RWEllipse(50, 50, 100, 100));
   ww.insert( new RWText(20, 20, "Hello world, from Rogue
              Wave!"));

   ww.show();      // Show the window                          // 4
   ww.update();    // Update the window

   // Enter the message loop:
   MSG msg;
   while( GetMessage( &msg, NULL, 0, 0))                       // 5
   {
     TranslateMessage( &msg );
     DispatchMessage( &msg );
   }

   return msg.wParam;                                          // 6
}
```

Here's the line-by-line description of the program.

`//1` This is the Windows program entry point that every Windows program must have. It is equivalent to C's `main` function.

`//2` This creates an instance of the class *DemoWindow*, which we will see later. It represents an abstract window into which objects can be inserted.

`//3` Here we insert three different objects into the *DemoWindow*: a rectangle, an ellipse, and a text string.

`//4` This tells the *DemoWindow* to show itself and to update itself.

`//5` Finally, the windows main event loop is entered.

`//6` The *DemoWindow* destructor frees all memory allocated for its window objects.

DEMOWIND.H

This header file declares the class *DemoWindow*. A key feature is the singly linked list called `myList`, which holds the list of items that have been inserted into the window. The member function `DemoWindow::insert(RWDrawable*)` allows new items to be inserted. Only objects that inherit from class *RWDrawable*, to be defined in "An Excerpt from SHAPES.H" on page 248, may be inserted.

The member function `paint()` may be called when it is time to repaint the window. The list `myList` will be traversed and each item in it will be repainted onto the screen.

Here's a listing of the `DEMOWIND.H` file

```
#ifndef __DEMOWIND_H__
#define __DEMOWIND_H__

/*
 * A Demonstration Window class --- allows items that
 * inherit from the base class RWDrawable to be
 * "inserted" into it.
 */

#include <windows.h>
#include <rw/slistcol.h>
#include "shapes.h"

class DemoWindow {
  HWND                hWnd;               // My window handle
```

```
  HINSTANCE            myInstance;      // My instance's handle
  int                   nCmdShow;
  RWSlistCollectables   myList;  // A list of items in the window

public:

  DemoWindow(HINSTANCE mom, HINSTANCE prev, int);
  ~DemoWindow();

    void   insert(RWDrawable*);   // Add a new item to the window

  HWND     handle() {return hWnd;}
  int      registerClass();           // First time registration
  void     paint();               // Called by Windows procedure
  int      show() {return ShowWindow(hWnd,nCmdShow);}
  void     update() {UpdateWindow(hWnd);}

  friend long FAR PASCAL _export
       DemoWindow_Callback(HWND, UINT, WPARAM, LPARAM);
};

DemoWindow*  RWGetWindowPtr(HWND h);
void         RWSetWindowPtr(HWND h, DemoWindow* p);

#endif
```

The DEMOWIND.CPP File

Now let's look at the definitions of the public functions of class *DemoWindow*: the *DemoWindow* constructor, the *DemoWindow* destructor, and the member functions `insert()`, `registerClass()`, and `paint()`. Detailed comments follow this listing.

```
#include <windows.h>
#include <rw/vstream.h>
#include "demowind.h"
#include "shapes.h"
#include <stdlib.h>
#include <string.h>
```

```
/*
 * Construct a new window.
 */
DemoWindow::DemoWindow( HINSTANCE mom, HINSTANCE prev,
                        int cmdShow)                          //1
{
  myInstance = mom;
  nCmdShow = cmdShow;

  // Register the class if there was no previous instance:
  if(!prev) registerClass();                                  //2

  hWnd = CreateWindow("DemoWindow",                           //3
                      "DemoWindow",
                      WS_OVERLAPPEDWINDOW,
                      CW_USEDEFAULT, 0,
                      CW_USEDEFAULT, 0,
                      NULL,
                      NULL,
                      myInstance,
                     (LPSTR)this );    // Stash away 'this'   //4

  if(!hWnd) exit( FALSE );
}

/*
 * Register self. Called once only.
 */
int
DemoWindow::registerClass(){                                  //5
   WNDCLASS wndclass;
   wndclass.style = CS_HREDRAW | CS_VREDRAW;
   wndclass.lpfnWndProc = ::DemoWindow_Callback;              //6
   wndclass.cbClsExtra = 0;
   // Request extra space to store the 'this' pointer
   wndclass.cbWndExtra = sizeof(this);                        //7
   wndclass.hInstance = myInstance;
   wndclass.hIcon = 0;
   wndclass.hCursor = LoadCursor( NULL, IDC_ARROW );
   wndclass.hbrBackground = GetStockObject( WHITE_BRUSH );
   wndclass.lpszMenuName = NULL;
   wndclass.lpszClassName = "DemoWindow";
```

```
   if( !RegisterClass(&wndclass) ) exit(FALSE);
   return TRUE;
}

DemoWindow::~DemoWindow(){
   // Delete all items in my list:
   myList.clearAndDestroy();                                    //8
}

void
DemoWindow::insert(RWDrawable* d){
   // Add a new item to the window:
   myList.insert(d);                                            //9
}

void
DemoWindow::paint(){                                           //10
   RWDrawable* shape;
   PAINTSTRUCT ps;
   BeginPaint( handle(), &ps);                                 //11

   // Draw all items in the list.  Start by making an iterator:
   RWSlistCollectablesIterator next(myList);                   //12

   // Now iterate through the collection, drawing each item:
   while( shape = (RWDrawable*)next() )                        //13
     shape->drawWith(ps.hdc);                                  //14

   EndPaint( handle(), &ps );                                  //15
}

/*
 * The callback routine for this window class.
 */
long FAR PASCAL _export
DemoWindow_Callback(HWND hWnd, unsigned iMessage,              //16
                    WPARAM wParam, LPARAM lParam)
{
   DemoWindow* pDemoWindow;

   if( iMessage==WM_CREATE ){                                  //17
```

```
     // Get the "this" pointer out of the create structure
     // and put it in the windows instance:
     LPCREATESTRUCT lpcs = (LPCREATESTRUCT)lParam;
     pDemoWindow = (DemoWindow*) lpcs->lpCreateParams;
     RWSetWindowPtr(hWnd, pDemoWindow);
     return NULL;
  }

   // Get the appropriate "this" pointer
   pDemoWindow = RWGetWindowPtr(hWnd);                          //18

   switch( iMessage ){
     case WM_PAINT:
       pDemoWindow->paint();                                    //19
       break;
     case WM_DESTROY:
       PostQuitMessage( 0 );
       break;
     default:
       return DefWindowProc(hWnd, iMessage, wParam, lParam);
  };
   return NULL;
}

void RWSetWindowPtr(HWND h, DemoWindow* p){                     //20
   SetWindowLong(h, 0, (LONG)p);
}

DemoWindow* RWGetWindowPtr(HWND h){                             //21
   return (DemoWindow*)GetWindowLong(h, 0);
}
```

`//1` This is the constructor for *DemoWindow*. It requires the handle of the application instance creating it, `mom`, the handle of any previously existing instance, `prev`, and whether to show the window in iconic form. These variables are as received from `WinMain`, the main windows procedure that we have already seen.

`//2` The constructor checks to see if any previous application instance has been run and, if not, registers the class.

`//3` The new window is created.

`//4` A key feature is the use of the Windows extra data feature to store the `this` pointer for this *DemoWindow*. The procedure to do this is somewhat cumbersome, but very useful. The value placed here will appear in the `CREATESTRUCT` structure, which we can retrieve when processing the `WM_CREATE` message generated by the `CreateWindow` function.

`//5` This member function is only called for the first instance of an application.

`//6` The global function `DemoWindow_Callback` is registered as the callback procedure for this window.

`//7` We ask Windows to set aside space for the `this` pointer in this Windows class. This will be used to associate a Windows handle with a particular *DemoWindow*.

`//8` The destructor calls `clearAndDestroy()` to delete all items that have been inserted into `myList`.

`//9` The member function `insert(RWDrawable*)` inserts a new item into the window. Because *RWDrawable* inherits from *RWCollectable*, as we shall see in “An Excerpt from SHAPES.H” on page 248, there is no need to do a cast when calling `RWSlistCollectables::insert(RWCollectable*)`. By making `myList` private and offering a restricted insert that takes arguments of type `RWDrawable*` only, we ensure that only drawables will be inserted into the window.

`//10` Here’s the paint procedure, called when it is time to repaint the window.

`//11` We start by getting the handle for this window, then calling `BeginPaint` to fill a `PAINTSTRUCT` with information about the painting.

`//12` An iterator is constructed in preparation for traversing the list of items to be painted.

`//13` Get the next item to be painted. If nil is returned, then there are no more items and the while loop will finish.

`//14` For each item, call the virtual function `drawWith`, causing the item to paint itself onto the given device context.

`//15` The `PAINTSTRUCT` is returned.

`//16` Here's the callback routine to be called when it is necessary to process a message for this window. It uses the very convenient `_export` keyword of Borland C++ to indicate that this function should be exported. If you use this procedure, you don't have to list the function in an Exports section of the module definition file.

`//17` If the message is a `WM_CREATE` message, it was generated by the `CreateWindow` function and this is the first time through for this procedure. Use the rather baroque procedure to fetch the `this` pointer from the `CREATESTRUCT` (recall it had been inserted at line 4), and put it into the Windows extra data. The function `RWSetWindowPtr` will be defined later.

`//18` The function `RWGetWindowPtr(HWND)` is used to retrieve the pointer to the appropriate *DemoWindow*, given a Windows `HANDLE`.

`//19` If a `WM_PAINT` has been retrieved, then call the `paint()` member function, which we have already seen.

`//20` This function is used to put a `this` pointer into the Windows extra data. The idea is to have a one-to-one mapping of Windows handles to instances of *DemoWindow*.

`//21` This function is used to fetch the `this` pointer back out.

An Excerpt from SHAPES.H

This section deals with the subclasses of *RWDrawable*. Class *RWDrawable* is an abstract base class that inherits from the *Tools.h++* class *RWCollectable*, and adds a new member function `drawWith(HDC)`. Subclasses specializing *RWDrawable* should implement this function to draw itself onto the supplied device context handle.

We have reprinted only one subclass here, *RWRectangle*. Class *RWRectangle* inherits from *RWDrawable*. Note that it uses the struct RECT provided by Windows as member data to hold the corners of the rectangle.

```
class RWDrawable : public RWCollectable{
public:
  virtual void  drawWith(HDC) const = 0;
};

class RWRectangle : public RWDrawable{
  RWDECLARE_COLLECTABLE(RWRectangle)

public:
  RWRectangle() { }
  RWRectangle(int, int, int, int);

  // Inherited from RWDrawable:
  virtual void  drawWith(HDC) const;

  // Inherited from RWCollectable:
  virtual Rwspace    binaryStoreSize() const
                        {return 4*sizeof(int);}
  virtual unsigned   hash() const;
  virtual RWBoolean  isEqual(const RWCollectable*) const;
  virtual void       restoreGuts(RWvistream& s);
  virtual void       restoreGuts(RWFile&);
  virtual void       saveGuts(RWvostream& s) const;
  virtual void       saveGuts(RWFile&) const;

private:

  RECT  bounds;                  // The bounds of the rectangle
};
```

An Excerpt from SHAPES.CPP

For the purposes of this DLL demo, it really isn't necessary to provide definitions for any of the member functions inherited from *RWCollectable*, but let's do it anyway, for the sake of completeness.

```
#include "shapes.h"
#include <rw/vstream.h>
#include <rw/rwfile.h>

RWDEFINE_COLLECTABLE(RWRectangle, 0x1000)                    //1

// Constructor
RWRectangle::RWRectangle(int l, int t, int r, int b)         //2
{
   bounds.left   = l;
   bounds.top    = t;
   bounds.right  = r;
   bounds.bottom = b;
}

// Inherited from Drawable:
void
RWRectangle::drawWith(HDC hdc) const                         //3
{
   // Make the Windows call:
   Rectangle(hdc, bounds.left, bounds.top,
             bounds.right, bounds.bottom);
}

// Inherited from RWCollectable:
unsigned
RWRectangle::hash() const                                    //4
{
  return bounds.left ^ bounds.top ^ bounds.bottom ^ bounds.right;
}

RWBoolean
RWRectangle::isEqual(const RWCollectable* c) const           //5
{
   if(c->isA() != isA() ) return FALSE;

   const RWRectangle* r = (const RWRectangle*)c;
```

```
    return bounds.left   == r->bounds.left  &&
           bounds.top    == r->bounds.top   &&
           bounds.right  == r->bounds.right &&
           bounds.bottom == r->bounds.bottom;
}
// Restore the RWRectangle from a virtual stream:
void
RWRectangle::restoreGuts(RWvistream& s)                    //6
{
   s >> bounds.left  >> bounds.top;
   s >> bounds.right >> bounds.bottom;
}

// Restore from an RWFile:
void
RWRectangle::restoreGuts(RWFile& f)                        //7
{
   f.Read(bounds.left);
   f.Read(bounds.top);
   f.Read(bounds.right);
   f.Read(bounds.bottom);
}

void
RWRectangle::saveGuts(RWvostream& s) const                 //8
{
   s << bounds.left  << bounds.top;
   s << bounds.right << bounds.bottom;
}

void
RWRectangle::saveGuts(RWFile& f) const                     //9
{
   f.Write(bounds.left);
   f.Write(bounds.top);
   f.Write(bounds.right);
   f.Write(bounds.bottom);
}
```

`//1` This is a macro that all subclasses of *RWCollectable* are required to compile once, and only once. See “How to Create an RWCollectable Object” on page 194 for more details.

`//2` This is the constructor for *RWRectangle* that fills in the `RECT` structure.

`//3` This is the definition of the virtual function `drawWith(HDC)`. It simply makes a call to the Windows function `Rectangle()` with appropriate arguments.

`//4` Supplies an appropriate hashing value in case we ever want to retrieve this *RWRectangle* from a hash table.

`//5` Supplies an appropriate `isEqual()` implementation.

`//6` This function retrieves the *RWRectangle* from a virtual stream.

`//7` This function retrieves the *RWRectangle* from an *RWFile*.

`//8` This function stores the *RWRectangle* on a virtual stream. Note how there is no need to separate elements by white space–this will be done by the virtual stream, if necessary.

`//9` This function stores the *RWRectangle* on an *RWFile*.

The other shapes, *RWEllipse* and *RWText*, are implemented in a similar manner.

Copy on Write

Classes *RWCString*, *RWWString*, and *RWTValVirtualArray<T>* use a technique called *copy on write* to minimize copying. This technique offers the advantage of easy-to-understand value semantics with the speed of reference counted pointer implementation.

Here is how the technique works. When an *RWCString* is initialized with another *RWCString* via the copy constructor:

```
RWCString(const RWCString&);
```

the two strings share the same data until one of them tries to write to it. At that point, a copy of the data is made and the two strings go their separate ways. Copying only at “write” time makes copies of strings, particularly read-

only copies, very inexpensive. In the following example, you can see how four objects share one copy of a string until one of the objects attempts to change the string:

```
#include <rw/cstring.h>

RWCString g;                                        // Global object

void setGlobal(RWCString x) { g = x; }

main(){
  RWCString a("kernel");                                          // 1
  RWCString b(a);                                                 // 2
  RWCString c(a);                                                 // 3

  setGlobal(a);       // Still only one copy of "kernel"!         // 4

  b += "s";           // Now b has its own data: "kernels"        // 5
}
```

`//1` The actual allocation and initialization of the memory to hold the string `kernel` occurs at the *RWCString* object `a`.

`//2 -//3` When objects `b` and `c` are created from `a`, they merely increment a *reference count* in the original data and return. At this point, there are *three* objects looking at the same piece of data.

`//4` The function `setGlobal()` sets the value of `g`, the global *RWCString*, to the same value. Now the reference count is up to four, and there is still only one copy of the string `kernel`.

`//5` Finally, object `b` tries to change the value of the string. It looks at the reference count and sees that it is greater than one, implying that the string is being shared by more than one object. At this point, a clone of the string is made and modified. The reference count of the original string drops back down to three, while the reference count of the newly cloned string is one.

A More Comprehensive Example

Because copies of *RWCString*s are so inexpensive, you are encouraged to store them by value inside your objects, rather than storing a pointer. This will greatly simplify their management, as the following comparison demonstrates. Suppose you have a window whose background and foreground colors can be set. A simple-minded approach to setting the colors would be to use pointers as follows:

```
class SimpleMinded {
  const RWCString*  foreground;
  const RWCString*  background;
public:
  setForeground(const RWCString* c) {foreground=c;}
  setBackground(const RWCString* c) {background=c;}
};
```

On the surface, this approach is appealing because only one copy of the string need be made. In this sense, calling `setForeground()` seems efficient. However, a closer look indicates that the resulting semantics can be muddled: what if the string pointed to by `foreground` changes? Should the foreground color change? If so, how will class *Simple* know of the change? There is also a maintenance problem: before you can delete a color string, you must know if anything is still pointing to it.

Here's a much easier approach:

```
class Smart {
  RWCString  foreground;
  RWCString  background;
public:
  setForeground(const RWCString& c) {foreground=c;}
  setBackground(const RWCString& c) {background=c;}
```

Now the assignment `foreground=c` will use *value* semantics. The color that class *Smart* should use is completely unambiguous. Copy on write makes the process efficient, too, since a copy of the data will not be made unless the string should change. The next example maintains a single copy of `white` until white is changed:

```
Smart window;
RWCString color("white");

window.setForeground(color);// Two references to white

color = "Blue";// One reference to white, one to blue
```

RWStringID

Many Rogue Wave clients have asked for a larger range of possible class identifiers for *RWCollectable* classes than is available using *RWClassID*. We did not change the meaning of *RWClassID*, in order to preserve backward compatibility for existing polymorphically persisted files, but we did add a new kind of class identifier, *RWStringID*.

An *RWStringID* is an identifier for *RWCollectable*s in *Tools.h++*Version 7 and later. It is derived from *RWCString*, and may be manipulated by any of the const *RWCString* methods. The non-const methods have been hidden to prevent the disaster that could occur if the *RWStringID* of a class changed at run time.

You can associate an *RWStringID* with an *RWCollectable* class in one of two ways: pick the *RWStringID* for the class, or allow the library to automatically generate an *RWStringID* that is the same sequence of characters as the name of the class; for example, `class MyColl : public RWCollectable` would get the automatic *RWStringID* `"MyColl"`.

You specify a class with a fixed *RWClassID* and generated *RWStringID* by using the macro `RWDEFINE_COLLECTABLE` as follows:

```
RWDEFINE_COLLECTABLE(ClassName, ClassID)
RWDEFINE_COLLECTABLE(MyCollectable1,0x1000)         //for example
```

You specify a class with a fixed *RWStringID* and a generated *RWClassID* by using the new macro `RWDEFINE_NAMED_COLLECTABLE` as follows:

```
RWDEFINE_NAMED_COLLECTABLE(ClassName, StringID)
RWDEFINE_NAMED_COLLECTABLE(MyCollectable2, "Second Collectable")
// for example
```

Using the examples above, you could write:

```
// First set up the experiment
MyCollectable1 one; MyCollectable2 two;
// All running RWClassIDs are guaranteed distinct
one.isA() != two.isA();
// Every RWCollectable has an RWStringID
one.stringID() == "MyCollectable1";
// There are several ways to find ids
RWCollectable::stringID(0x1000) == "MyCollectable1";
two.isA() == RWCollectable::classID("Second Collectable");
```

Duration of Identifiers

Providing polymorphic persistence between different executions of the same or different programs requires a permanent identifier for each class being persisted. Until now, the permanent identifier for any *RWCollectable* has been its *RWClassID*. For each class that derives from *RWCollectable*, the macro `RWDEFINE_COLLECTABLE` caused code to be generated that forever associated the class and its *RWClassID*. This identification has been retained, but in the current version of *Tools.h++* you may choose the `RWDEFINE_NAMED_COLLECTABLE` macro, which will permanently link the chosen *RWStringID* with the class.

The addition of *RWStringID* identifiers will result in more identifiers, and more self-documenting *RWCollectable* identifiers, than were possible under the old restriction. To accommodate the new identifiers, a temporary *RWClassID* is now generated for each *RWCollectable* class that has an *RWStringID* specified by the developer. These *RWClassID*s are built as needed during the run of an executable, and remain constant throughout that run. However, they may be generated in a different order on a different executable or during a different run, so they are not suitable for permanent storage.

Programming with RWStringIDs

RWCollectable now has a new regular member function, and two new static member functions. Since one of the major goals of Version 7 of *Tools.h++* is to maintain link compatibility with objects compiled against Version 6, none of these functions is virtual. The functions are therefore slightly less efficient than they would be if we broke link-compatibility.

The new regular member function is:

```
RWStringID  stringID() const;
```

The new static member functions are:

```
RWStringID  stringID(RWClassID);          // looks up RWStringID
RWClassID  classID(RWStringID);             // looks up classID
```

RWFactory also includes the following new functions:

```
void           addFunction(RWuserCreator, RWClassID, RWStringID);
RWCollectable* create(RWStringID) const;
RWuserCreator  getFunction(RWStringID) const;
```

```
void           removeFunction(RWStringID);
RWStringID     stringID(RWClassID) const;
RWClassID      classID(RWStringID) const;
```

You can use *RWCollectable*s that ship with *Tools.h++* and *RWCollectable*s that have been defined with fixed *RWClassID*s exactly as in previous versions of *Tools.h++*. For instance, you could still use this common programming idiom:

```
RWCollectable *ctp;                          // assign the pointer
if (ctp->isA() == SOME_CONST_CLASSID)      // do a specific thing
```

However, when you use *RWCollectable*s that have user-provided *RWStringID*s, which implies any non-permanent *ClassID*s, you must anticipate that the *RWClassID* may have different values during different runs of the executable. For these classes, there are two possible idioms to replace the one above:

```
RWCollectable *ctp;
// assign the pointer somehow
// use with existing RWCollectable for comparison:
// comparison will be faster than comparing RWStringIDs
if(ctp->isA() == someRWCollectablePtr->isA())
  // you may code to that class interface
// ...
// idiom to hard code the identification. Slightly
// slower because string comparisons are slower than int
// comparisons; also stringID() uses a dictionary lookup.
if (ctp->stringID() == "Some ID String")  {
  // you may code to that class interface
}
```

Implementation Details of RWStringID

The next few sections cover implementation details of *RWStringID*. If you are curious about how we manage to provide virtual functionality without adding virtual methods, or if you are interested in issues of design, efficiency, and other specifics, these sections are for you.

Automatic RWClassIDs

Automatic *RWClassID*s are created in a systematic way from unused *RWClassID*s in the range 0x9200 to 0xDAFF. There are 18,687 possible such *RWClassID*s, so only extraordinary programs can possibly run out. However,

we are used to dealing with extraordinary customers, so we feel we must warn you: you will not be able to build and use more than 18,687 different *classes* with automatically generated *RWClassID*s in any one program[1].

Note that this implies nothing about the total number of objects of each class that you may have. That number is limited only by the requirements of your operating system and compiler. Of course, you also have access to the full set of *RWClassID*s below 0x8000–that is, 32767 more possible *RWCollectable*s—but they will not be automatically generated. You must specify them manually.

Implementing Virtuals Via Statics

Since the virtual method `isA()` returns a "run-time unique" *RWClassID*, we can use this one virtual method to provide an index into a lookup table where various data or function pointers are stored. (This may remind you of C++ built-in `vtables`!) Since *RWCollectable*s already depend on the existence of a single *RWFactory*, we chose to use that *RWFactory* instance to hold the lookup information.

The static method:

```
RWStringID  RWCollectable::stringID(RWClassID id);
```

will attempt to look up `id` in the *RWFactory* instance. If it succeeds in finding an associated *RWStringID*, it will return it. Otherwise, it will return `RWStringID("NoID")`.

The static method:

```
RWClassID  RWCollectable::classID(RWStringID sid)
```

works in an analogous manner, looking in the *RWFactory* instance to see if there is an *RWClassID* associated with `sid`. If the method finds one, it returns it; otherwise, it returns *RWClassID* `__RWUNKNOWN`.

Polymorphic Persistence

Polymorphic persistence of *RWCollectable*s is not affected by the addition of the new class *RWStringID*. Existing files can still be read using newly compiled and linked executables, as long as the old *RWClassID*s are unchanged. New classes that have *RWStringID*s may be freely intermixed with old classes. The

1. 16-bit DLLS will also accumulate automatic *RWClassIDs* while they are loaded in memory.

storage size of collectables that do not have permanent *RWClassID*s will reflect their larger space requirements, but the store size of other *RWCollectable*s will be unaffected.

Note that collections containing *RWCollectable*s with the same *RWStringID* have that *RWStringID* stored into a stream or file only once, just as multiple references to the same *RWCollectable* are only stored the first time they are seen.

Efficiency

Since *RWClassID* is more efficient in both time and space than *RWStringID*, you may wish to continue using it wherever possible. *RWStringID*s are useful:

- For organizations that need to generate unique identifiers for many programming groups;
- For third party libraries that need to avoid clashes with other libraries or users;
- Anywhere the self-documenting feature of *RWStringID* adds enough value to compensate for its slight inefficiencies.

*RWStringID*s are generated for all *RWCollectable* classes that are compiled under the current version of *Tools.h++*. This additional code generation has only minor impact on programs that do not use the *RWStringID*s. The *RWFactory* will be larger, to hold lookups from *RWClassID* and *RWStringID*; and startup time will be very slightly longer, to accommodate the addition of the extra data to the *RWFactory*.

Identification Collisions

While *RWStringID* can help alleviate identification collisions, the possibility of collisions between *RWStringID*s of different classes still exists. Collisions can occur:

- When an automatically generated *RWStringID* conflicts with a user-chosen one;
- When one or more classes are accidentally assigned the same *RWStringID*;
- When two classes in different namespaces have the same name and thus the same automatically generated *RWStringID*. This assumes your compiler supports namespaces.

In some cases, collisions like these will be unimportant. Automatically generated *RWClassID*s are guaranteed to be distinct from one another and from any legal user-provided *RWClassID*. The virtual `isA()` method, the `stringID()` method, and constructor lookup based on the *RWClassID* will all continue to work correctly.

There will be some situations, however, where collisions will cause difficulty. Polymorphic persistence of classes with user-chosen *RWStringID*s that collide will not work correctly. In these cases, the data will not be recoverable, even though it is stored correctly. Similarly, user code that depends on distinguishing between classes based only on their *RWStringID*s will fail.

As a developer, you can work to avoid such collisions. First of all, you should use an *RWStringID* which is unlikely to collide with any other. For instance, you might choose *RWStringID*s that mimic the inheritance hierarchy of your class, or that imbed your name, your company's name, a creation time, or a file path such as found in revision control systems. And of course, you should always test your program to insure that the class actually associated with your *RWStringID* is the one you expected.

More on Storing and Retrieving RWCollectables

In "Operators" on page 175, we saw how to save and restore the morphology or pointer relationships of a class using the following global functions:

```
Rwvostream&  operator<<(RWvostream&, const RWCollectable&);
RWFile&      operator<<(RWFile&,     const RWCollectable&);
Rwvostream&  operator<<(RWvostream&, const RWCollectable*);
RWFile&      operator<<(RWFile&,     const RWCollectable*);
Rwvistream&  operator>>(RWvistream&, RWCollectable&);
RWFile&      operator>>(RWFile&,     RWCollectable&);
Rwvistream&  operator>>(RWvistream&, RWCollectable*&);
RWFile&      operator>>(RWFile&,     RWCollectable*&);
```

When working with *RWCollectable*s, it is useful to understand how these functions work. Here is a brief description.

When you call one of the left-shift `<<` operators for any collectable object for the first time, an identity dictionary is created internally. The object's address is put into the dictionary, along with its ordinal position in the output file—for example, first, second, sixth, etc.

Once this is done, a call is made to the object's virtual function `saveGuts()`. Because this is a *virtual* function, the call will go to the definition of `saveGuts()` used by the derived class. As we have seen, the job of `saveGuts()` is to store the internal components of the object. If the object contains other objects inheriting from *RWCollectable*, the object's `saveGuts()`calls `operator<<()` recursively for each of these objects.

Subsequent invocations of `operator<<()` do not create a new identity dictionary, but store the object's address in the already existing dictionary. If an address is encountered which is identical to a previously written object's address, then `saveGuts()` *is not called*. Instead, a reference is written that this object is identical to some previous object.

When the entire collection is traversed and the initial call to `saveGuts()` returns, the identity dictionary is deleted and the initial call to `operator<<()` returns.

The function `operator>>()` essentially reverses this whole process by calling `restoreGuts` to restore objects into memory from a stream or file. When encountering a reference to an object that has already been created, it merely returns the address of the old object rather than asking the *RWFactory* to create a new one.

Here is a more sophisticated example of a class that uses these features:

```
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <assert.h>

class Tangle : public RWCollectable
{

public:

  RWDECLARE_COLLECTABLE(Tangle)

  Tangle* nextTangle;
  int     someData;

  Tangle(Tangle* t = 0, int dat = 0){nextTangle=t; someData=dat;}
```

```
  virtual void saveGuts(RWFile&) const;
  virtual void restoreGuts(RWFile&);

};

void Tangle::saveGuts(RWFile& file) const{
   RWCollectable::saveGuts(file);         // Save the base class

   file.Write(someData);                       // Save internals

   file << nextTangle;                     // Save the next link
}

void Tangle::restoreGuts(RWFile& file){
   RWCollectable::restoreGuts(file);    // Restore the base class

  file.Read(someData);                      // Restore internals

  file >> nextTangle;                   // Restore the next link
}

// Checks the integrity of a null terminated list with head "p":
void checkList(Tangle* p){
   int i=0;
   while (p)
   {
     assert(p->someData==i);
     p = p->nextTangle;
     i++;
   }
}

RWDEFINE_COLLECTABLE(Tangle, 100)

main(){
   Tangle *head = 0, *head2 = 0;

   for (int i=9; i >= 0; i--)
     head = new Tangle(head,i);

   checkList(head);                   // Check the original list
```

```
   {
     RWFile file("junk.dat");
     file << head;
   }

   RWFile file2("junk.dat");
   file2 >> head2;

   checkList(head2);                  // Check the restored list
   return 0;
}
```

In the above example, the class Tangle implements a circularly linked list. What happens? When function `operator<<()` is called for the first time for an instance of Tangle, it sets up the identity dictionary as described above, then calls Tangle's `saveGuts()`, whose definition is shown above. This definition stores any member data of Tangle, then calls `operator<<()` for the next link. This recursion continues on around the chain.

If the chain ends with a nil object (that is, if `nextTangle` is zero), then `operator<<()` notes this internally and stops the recursion.

On the other hand, if the list is circular, then a call to `operator<<()` is eventually made again for the first instance of Tangle, the one that started this whole chain. When this happens, `operator<<()` will recognize that it has already seen this instance before and, rather than call `saveGuts()` again, will just make a reference to the previously written link. This stops the series of recursive calls and the stack unwinds.

Restoration of the chain is done in a similar manner. A call to:

```
RWFile& operator>>(RWFile&, RWCollectable*&);
```

can create a new object off the heap and return a pointer to it, return the address of a previously read object, or return the null pointer. In the last two cases, the recursion stops and the stack unwinds.

18

Multiple Inheritance

In Chapter 15, we built a *Bus* class by inheriting from *RWCollectable*. If we had an existing Bus class at hand, we might have saved ourselves some work by using multiple inheritance to create a new class with the functionality of both *Bus* and *RWCollectable* as follows:

```
class CollectableBus : public RWCollectable, public Bus {
  .
  .
  .
};
```

This is the approach taken by many of the Rogue Wave collectable classes; for example, class *RWCollectableString* inherits from both class *RWCollectable* and class *RWCString*. The general idea is to create your object first, then tack on the *RWCollectable* class to make the whole thing collectable. This way, you will be able to use your objects for other things or in other situations, where you might not want to inherit from class *RWCollectable*.

There is another good reason for using this approach: to avoid ambiguous base classes. Here's an example:

```
class A { };
class B : public A { };
class C : public A { };
class D : public B, public C { };
void fun(A&);
main () {
  D  d;
  fun(d);                                          // Which A ?
}
```

There are two approaches to disambiguating the call to `fun()`. We can either change it to:

```
fun((B)d);                          // We mean B's occurrence of A
```

or make `A` a virtual base class.

The first approach is error-prone because the user must know the details of the inheritance tree in order to make the proper cast.

The second approach, making A a virtual base class, solves this problem, but introduces another: it becomes nearly impossible to make a cast back to the derived class! This is because there are now two or more paths back through the inheritance hierarchy or, if you prefer a more physical reason, the compiler implements virtual base classes as pointers to the base class and you can't follow a pointer backwards.

We could exhaustively search all possible paths in the object's inheritance hierarchy, looking for a match. (This is the approach of the NIH Classes.) However, this search is slow, even if speeded up by "memorizing" the resulting addresses, since it must be done for every cast. Since it is also bulky and always complicated, we decided that it was unacceptable.

Hence, we went back to the first route. This can be made acceptable if we keep the inheritance trees simple by not making everything derive from the same base class. Hence, rather than using a large secular base class with lots of functionality, we have chosen to tease out the separate bits of functionality into separate, smaller base classes.

The idea is to first build your object, *then* tack on the base class that will supply the functionality you need, such as collectability. You thus avoid multiple base classes of the same type and the resulting ambiguous calls.

18

Common Mistakes 19

Rogue Wave libraries are built by developers like you who understand the frustration of programming errors. We try hard to fine tune our libraries to minimize these errors. Nevertheless, the complexity of C++ affords countless opportunities for making some very subtle mistakes.

In writing this section, we went though our technical support documents to uncover the most common mistakes our users were reporting. When we found mistakes that could be prevented, we tried to rewrite the library to make them impossible. This is always the best approach. We can't always follow it, however, if unacceptable performance degradations result, or the language itself prohibits the change.

This section summarizes the most common mistakes that are left over after we fixed the ones we could. If you're having a problem, take a look through this list and, of course, be sure to read the pertinent parts of the manual.

Redefinition of Virtual Functions

If you subclass off an existing class and override a virtual function, make sure that the overriding function has exactly the same signature as the overridden function. This includes any `const` modifiers!

This problem arises particularly when creating new *RWCollectable* classes. For example:

```
class MyObject : public RWCollectable {
public:
  RWBoolean isEqual();                              // No "const" !
};
```

The compiler will treat this definition of `isEqual()` as completely independent of the `isEqual()` in the base class *RWCollectable*, because it is missing a `const` modifier. Hence, if called through a pointer:

```
MyObject obj;
  RWCollectable* c = &obj;
  c->isEqual();// RWCollectable::isEqual() will get called!
```

Iterators

Since the drafting of the ANSI/ISO Standard C++ Library, there are now two kinds of iterators available for use in *Tools.h++*: the traditional iterators which we describe in detail throughout this manual, and the new "Standard Library" iterators. For more information about using the iterators now mandated by the standard, we refer you to the manual which came with your version of the Standard C++ Library. In this *Tools.h++* manual, unless we specifically say otherwise, an *iterator* refers to a *traditional* iterator.

Immediately after construction, the position of a *Tools.h++*iterator is formally undefined. You must advance it or position it before it has a well-defined position. The rule of thumb is "advance and then return." If the end of the collection has been reached, the return value after advancing will be special, either `FALSE` or nil, depending on the type of iterator.

Hence, the proper idiom is:

```
RWSlistCollectables ct;
RWSlistCollectablesIterator it(ct);


.
.
.


RWCollectable* c;
while (c=it()) {
  // Use c
```

```
}
```

Return Type of operator>>()

An extremely common mistake is to forget that the functions:

```
Rwvistream&  operator>>(RWvistream&, RWCollectable*&);
RWFile&      operator>>(RWFile&,     RWCollectable*&);
```

return their results *off the heap*. This can result in a memory leak like the following:

```
main(){
   RWCollectableString* string = new RWCollectableString;
   RWFile file("mydata.dat");

   // WRONG:
   file >> string;                          // Memory leak!

   // RIGHT:
   delete string;
   file >> string;
}
```

Avoid Persisting Value Collections of Pointers

It may sometimes be reasonable to collect pointers into a value-based collection in order to deal with *identies* instead of *values*. However, you should never attempt to persist them, since a collection with value semantics will simply store the *values* of the pointers into the stream, rather than storing the *information pointed to*. If you were to pull those old pointers out of the stream and back into memory, they would almost surely point to invalid locations.

Include Path

When you specify an include path to the Rogue Wave header files, make sure that it does *not* include a final `rw`:

```
#  Use this:
CC -I/usr/local/include -c myprog.C

#  not this:
CC -I/usr/local/include/rw -c myprog.C
```

19

Match Memory Models and Other Qualifiers

When it comes time to link your program to the Rogue Wave library, make sure that all the following match: qualifiers, compilation "mode" macros (such as `RWDEBUG` and `RW_MULTI_THREAD`), the choice to use (or not use) shared libraries, and memory models. For example, if you compiled using the flat memory model, make sure you are linking to a library compiled with the flat memory model. Similarly, if you are using a debug version of the library, be sure to compile your programs with the same debug settings (see "The Debug Version of Tools.h++" on page 236).

Failure to do so will result in the linker emitting mysterious "undefined external reference" or other errors, or even worse, you may find that programs run, but do not execute as expected.

Keep Related Methods Consistent

When you design classes that will be used with the *Tools.h++* library, you may be tempted to take short cuts, like providing a simplistic `hash` method, or `operator<()`, since you "know it will never be used anyhow." Decisions like this can have disastrous maintenance consequences later. Unless you have a very good reason, it makes sense to ensure, for example, that `operator<()`, `operator==()`, `compareTo()`, and `isEqual()` are based on the same information. You must also be sure that values which are `isEqual()` or `==` with each other have the same hash value, since otherwise they will never be found if placed into a collection that uses hashing techniques. A little extra effort at the beginning can pay big dividends in reduced debugging time later on!

DLL

Because the DLL version of *Tools.h++* uses the large memory model, any data segment that uses it must be fixed. For example, if you were to create an *RWCollectable* object in your data segment and insert it into a *Tools.h++* collection, that collection will be holding a four byte pointer. If your data segment were to move, the pointer would no longer be valid. Hence, be sure that your `.DEF` definition file has a line similar to the following:

```
DATA  PRELOAD FIXED
```

Note that with Microsoft's decision to abandon real mode Windows, working with fixed data and global memory is no longer the problem it used to be. The extra level of indirection offered by protected mode allows data to be moved around in physical memory without invalidating selectors. The entries in the descriptor table are changed instead.

Use the Capabilities of the Library!

By far the most common mistake is not to use the full power of the library. If you find yourself writing a little "helper" class, consider why you are doing it. Or, if what you are writing is looking a little clumsy, then maybe there's a more elegant approach. A bit of searching through the *Tools.h++* manual may uncover just the thing you're looking for!

Here's a surprisingly common example:

```
main(int argc, char* argv[]){
   char buffer[120];                    //uh oh: possible overflow
   ifstream fstr(argv[1]);
   RWCString line;

   while (fstr.readline(buf,sizeof(buf)) {
     line = buf;                                  //hmm: extra copy
     cout << line;
   }
}
```

This program reads lines from a file specified on the command line and prints them to standard output. By using the full abilities of the *RWCString* class it could be greatly simplified as follows:

```
main(int argc, char* argv[]){
   ifstream fstr(argv[1]);
   RWCString line;

   while (line.readLine(fstr)) {
     cout << line;
   }
}
```

There are countless other such examples. The point is, if it's looking awkward to you, most likely there's a better way!

19

Choosing A Collection A

Tools.h++ has an abundance of collection classes--when you're faced with choosing which one to use in your code, it may seem like an overabundance! This section provides suggestions and information that will help you select the most appropriate collection for a given programming task.

Choosing the most appropriate collection class to fit your needs is not a trivial task. First you need to consider the data in your collection. Does your collection need to store the data in order? Will there be duplicate data? And, how do you find or insert data in your collection? The first part of this appendix, "Selecting a *Tools.h++* Collection Class" includes a decision tree diagram that lets you consider specific questions about your data and, through your answers, quickly focus on the collections that will best fit your data requirements. A preface to the decision tree discusses questions you'll see in the tree and includes some additional selection criteria that address issues such as whether to choose a pointer or value-based collections, when to use sequential collections, and what to use for disk-based access.

The second part of this appendix presents a rough comparison of how much time and memory different collections and collection families need to perform common operations such as insertions, finds, and removals.

Selecting a Tools.h++ Collection Class

The decision tree diagram includes questions about the data you plan to store in your collection. By traversing the tree you can quickly see which *Tools.h++* collection classes will best suit your data and programming project.

How to Use the Decision Tree

The questions that appear on the decision tree are brief, so that the diagram will be easy to read. The following questions expand upon the questions in the decision tree.

1. *Is the order of data in the collection meaningful?* Some collections allow you to control the location of data within the collection. For example, arrays or vectors, and linked lists present data *in order.* If you need to access data in a particular order, or based on a numeric index, then order is meaningful.

2. *Are duplicate entries allowed?* Some collections, usually called sets, will not allow you to insert an item equal to an item that is already in the collection. Other collections do permit duplicates, and have various ways to hold multiple matching items. *Tools.h++* collections provide mechanisms for both checking for duplication and holding the duplicates.

3. *Is the order determined intrinsically or externally?* If data within the collection is controlled by the way you insert it, we say that the order is determined *externally.* For example, a vector, or a linked list is externally ordered. If the data is stored at a place determined by an algorithm used by the collection, then the ordering is *intrinsic*. For example, a sorted vector, or a balanced tree has intrinsic ordering.

4. *Is data accessed by an external key?* If you access a value based on a key that is not the same as the value, we say that data is accessed by an external key. For example, a "phone list" associates data, in the form of telephone numbers, with keys, in the form of names. Conversly, a list of committee members is simply a set of data in the form of names. You do not need a key to get at the information.

5. *Is data accessed by a numeric index?* Objects stored in an array or vector are accessed by numeric index. For example, you access an object at location 12 by using the numeric index “12” to find it.

6. *Is data accessed by "comparison with self?"* Data that is stored with neither an explicit index nor an explicit key can be found by looking for a match between what you need to find and what is contained. The list of committee members mentioned in item 4 is and example of this type of data. Sets or bags are examples of collections that are accessed by comparison with self.

When data is accessed by comparison with self, it is also necessary to know what kind of match is used: matching may be based on *equality*, which directly compares one object with another, or based on *identity*, which compares object addresses to see if the objects are the same.

7. *Is the best method of access by following linked nodes?* Collections that make use of linked nodes are usually called *lists*. Lists provide quick access to data at each end of the collection, and allow you to insert data efficiently into the middle of the collection. However, if you need repeated access to data in the middle of the collection, lists are not as efficient as some other collections.

8. *Will most of your access to data be at the ends of a collection?* There are many occasions when you need to handle an unknown amount of data, but most of that data handling will apply to data that was most recently or least recently put into the collection. A collection that is particularly efficient at handling data that was most recently added is said to have a "last in, first out" policy. A last in, first out (LIFO) container is a *stack*. A collection that handles the data in a "first in first out" (FIFO) manner is called a *queue*. Finally, a collection that allows efficient access to both the most recently and least recently added data is called a *deque*, or double ended queue.

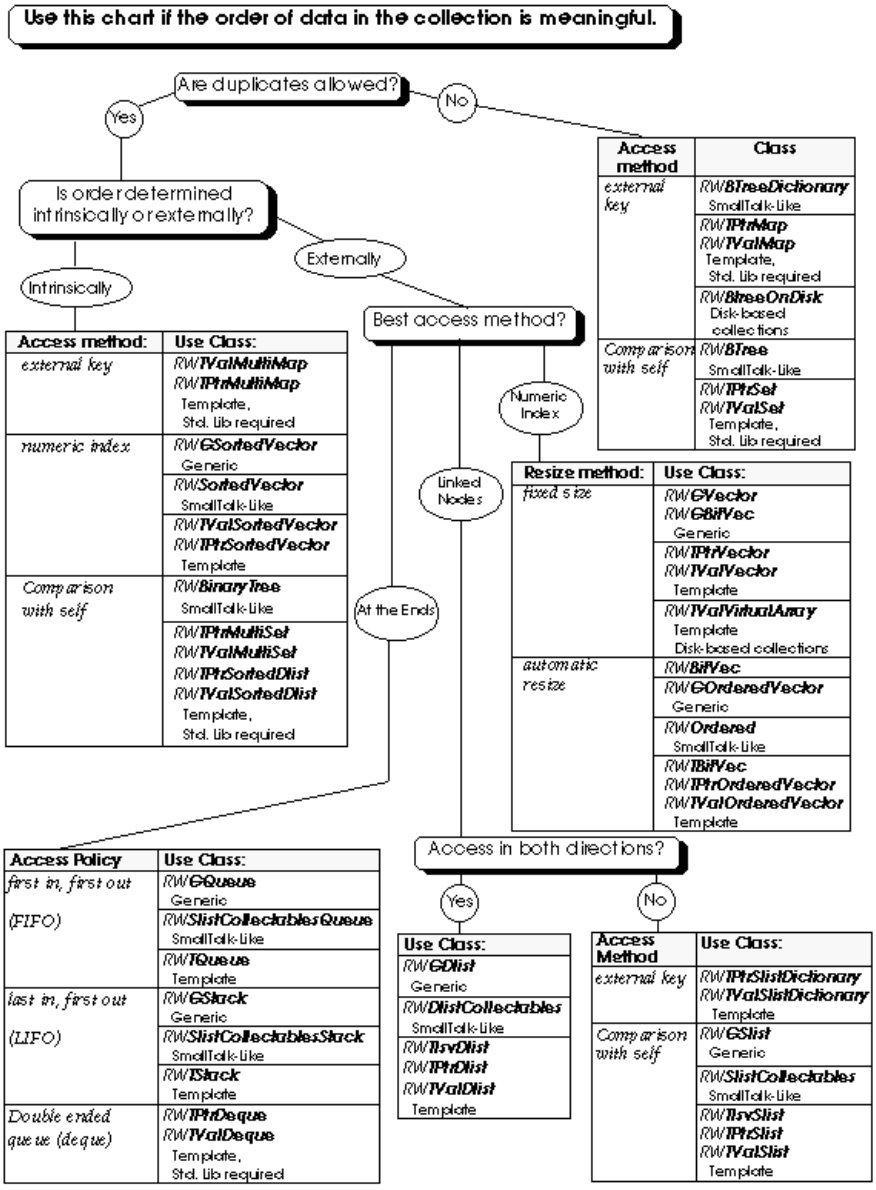
9. *For linked lists—Do you need to access the data from only one end of the list , or from both ends?* Singly-linked lists are smaller, but they allow access only from the "front" of the list. Doubly-linked lists have a more flexible access policy, but at the cost of requiring an additional pointer for every stored object.

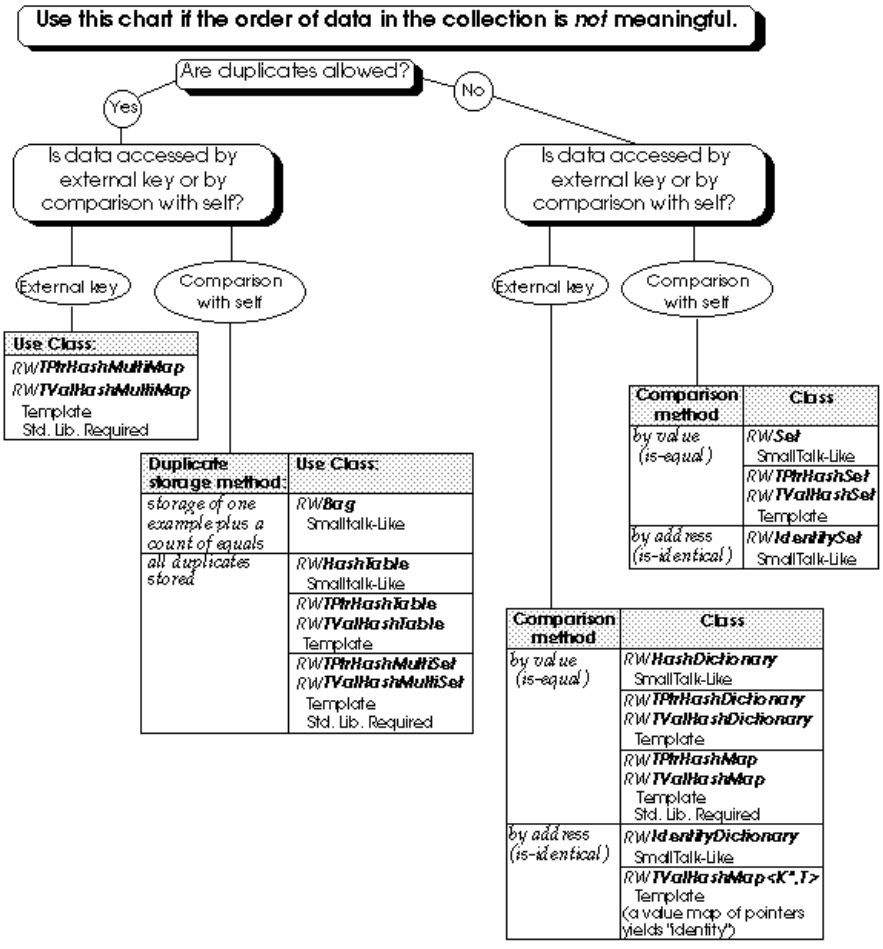
10. *For collections that are accessed by numeric index—Do you need the collection to automatically resize?* If you know the maximum number of items that will be stored in the collection, you can make insertion and removal slightly more efficient by choosing a collection with a fixed size. On the other hand, if you need to allow for nearly unlimited expansion, you should choose a collection that will automatically adjust itself to the amount of data it is currently storing.

Additional Selection Criteria

Which collection you choose will depend of many things (not the least of which will be your experience and intuition). In addition to the decision tree, the following questions will also influence your choice of container.

1. *Do I need to maintain a single object in multiple collections?* Use a pointer-based collection.
2. *Am I collecting objects that are very expensive to copy?* Use a pointer-based collection.
3. *Is there no compelling reason to use a pointer-based collection?* Use a value-based collection.
4. *Do I want to control the order of the objects within the collection externally?* Use a sequential collection such as a vector or list.
5. *Should the items within the collection be mutable (not fixed) after they are inserted?* Use a sequential or mapping (dictionary) collection. Maps and dictionaries have immutable keys but mutable values.
6. *Would I prefer that the collection maintain its own order based on object comparison?* Use a set, map, or sorted collection.
7. *Do I wish to access objects within the collection based on a numerical index?* Use a sequential or sorted collection.
8. *Do I need to find values based on non-numeric keys?* Use a map or dictionary.
9. *Would I prefer to access objects within the collection by supplying an object for comparison?* Use a set, map or hash-based collection.
10. *Am I willing to forego meaningful ordering, and use some extra memory in return for constant-time look-up by key?* Use a hash-based collection.
11. *Do I need fast lookup and insertion in a collection that grows or shrinks to meet the current need?* Use a b-tree, or an associative container based on the new Standard C++ Library.
12. *Do I need access the data without bringing it all into memory?* Use *RWBTreeOnDisk* or *RWTValVirtualArray*.


Use this chart if the order of data in the collection is meaningful.
Are duplicates allowed?
Yes
No
Is order determined intrinsically or externally?
Intrinsically
Externally
Best access method?
Numeric Index
Linked Nodes
At the Ends
Access method | Class
external key | RW/BTreeDictionary SmallTalk-like
RW/IPtrMap RW/TValMap Template, Std. Lib required
RW/BTreeOnDisk Disk-based collections
Comparison with self | RW/BTree SmallTalk-like
RW/IPtrSet RW/TValSet Template, Std. Lib required
Access method: | Use Class:
external key | RW/TValMultiMap RW/IPtrMultiMap Template, Std. Lib required
numeric index | RW/GSortedVector Generic
RW/SortedVector SmallTalk-like
RW/TValSortedVector RW/IPtrSortedVector Template
Comparison with self | RW/BinaryTree SmallTalk-like
RW/IPtrMultiSet RW/TValMultiSet RW/IPtrSortedDlist RW/TValSortedDlist Template, Std. Lib required
Resize method: | Use Class:
fixed size | RW/GVector RW/GBitVec Generic
RW/IPtrVector RW/TValVector Template
RW/TValVirtualArray Template Disk-based collections
automatic resize | RW/BitVec
RW/GOrderedVector Generic
RW/Ordered SmallTalk-like
RW/IBitVec RW/IPtrOrderedVector RW/TValOrderedVector Template
Access Policy | Use Class:
first in, first out (FIFO) | RW/GQueue Generic
RW/SlistCollectablesQueue SmallTalk-like
RW/TQueue Template
last in, first out (LIFO) | RW/GStack Generic
RW/SlistCollectablesStack SmallTalk-like
RW/TStack Template
Double ended queue (deque) | RW/IPtrDeque RW/TValDeque Template, Std. Lib required
Access in both directions?
Yes
No
Use Class:
RW/GDlist Generic
RW/DlistCollectables SmallTalk-like
RW/IIsvDlist RW/IPtrDlist RW/TValDlist Template
Access Method | Use Class:
external key | RW/IPtrSlistDictionary RW/TValSlistDictionary Template
Comparison with self | RW/GSlist Generic
RW/SlistCollectables SmallTalk-like
RW/IIsvSlist RW/IPtrSlist RW/TValSlist Template


Use this chart if the order of data in the collection is *not* meaningful.
Are duplicates allowed?
Yes
No
Is data accessed by external key or by comparison with self?
Is data accessed by external key or by comparison with self?
External key
Comparison with self
External key
Comparison with self
Use Class:
RW**TPtrHashMultiMap**
RW**TValHashMultiMap**
Template
Std. Lib. Required
Duplicate storage method:
Use Class:
storage of one example plus a count of equals
RW**Bag**
SmallTalk-Like
all duplicates stored
RW**HashTable**
SmallTalk-Like
RW**TPtrHashTable**
RW**TValHashTable**
Template
RW**TPtrHashMultiSet**
RW**TValHashMultiSet**
Template
Std. Lib. Required
Comparison method
Class
by value (is-equal)
RW**Set**
SmallTalk-Like
RW**TPtrHashSet**
RW**TValHashSet**
Template
by address (is-identical)
RW**IdentitySet**
SmallTalk-Like
Comparison method
Class
by value (is-equal)
RW**HashDictionary**
SmallTalk-Like
RW**TPtrHashDictionary**
RW**TValHashDictionary**
Template
RW**TPtrHashMap**
RW**TValHashMap**
Template
Std. Lib. Required
by address (is-identical)
RW**IdentityDictionary**
SmallTalk-Like
RW**TValHashMap<K*,T>**
Template
(a value map of pointers yields "identity")

Time and Space Considerations

This section presents a very approximate analysis and comparison of the time and space requirements for a variety of common operations on different specific collections and collection families. We've presented the information as a set of tables that lists the operation, the time cost and the space cost. Any applicable comments appear at the bottom of the table. A key to the abbreviations used in the tables appears at the bottom of each page.

As you read these analyses, keep in mind that various processors, operating systems, compilation optimizations, and many other factors will affect the exact values. The point of these tables is to provide you with some idea of how the behaviors of the various collections will compare, *all other things being equal*. For more details on algorithm complexity, refer to Knuth, Sedgewick, or any number of other books.

Because many of the *Tools.h++* collections have essentially similar interfaces, it is easy to experiment and discover what effect a different choice of collection will have on your program.

For each of the following tables:

- `N` is the number of items in the collection;
- `M` is the current size of the collection;
- `t` is the size of the item being stored (possibly a pointer);
- `i` is the size of an integer;
- `p` is the size of a pointer
- `C` is a “constant value”.
- Time costs for each pointer dereference, copy, destroy, allocate, or comparison are considered equal.
- Container overhead is as space cost that consists of two terms. The left term is the size of an empty container, while the right term shows the added cost for `N` items.
- Space cost is indicated both for insertions and deletions. Space cost that is marked "(recovered)" indicates that the space has been handed back to the heap allocator.

Whenever an allocation is mentioned, you should be aware that memory allocation policies differ radically among various implementations. However, it is generally true that a heap allocation (or deallocation) that translates to a

system call is more expensive than most of the other constant costs. "Amortized" cost is averaged over the life of the collection. Any individual action may have a higher or lower cost than the amortized value.

RWGVector, RWGBitVec, RWTBitVec<size>, RWTPtrVector, and RWTValVector

operation	time cost	space cost
Insert at an end	C	0
Insert in middle	C	0
Find (average item)	N/2	0
Change/replace item	C	0
Remove first	C	0
Remove last	C	0
Remove in middle	C	0
Container overhead		Mt + 0
Comments	Simple wrapper around an array of T (except `bitvec: array` of byte)	Resize only if told to.

Singly Linked Lists

operation	**time cost;**	**space cost**
Insert at an end	`C`	`t + p`
Insert in middle	`C` (assumes that you have an iterator in position)	`t + p`
Find (average item)	`N/2`	`0`
Change/replace item	`C`	`0`
Remove first	`C`	`t + p` (recovered)
Remove last	`C`	`t + p` (recovered)
Remove in middle	`C` (assumes that you have an iterator in position)	`t + p` (recovered)
Container overhead		`(2p+i) + N(t+p)`
Comments	Allocation with each insert Iterators go forward only	Grows or shrinks with each item. Smaller than doubly-linked list

Key to the comparison tables

`N`	`M`	`t`	`i`	`p`	`C`
count of items	count of space for items	`sizeof (item)`	`sizeof (int)`	`sizeof (void*)`	a constant

Doubly Linked Lists

operation	time cost;	space cost
Insert at an end	C	t + 2p
Insert in middle	C (assumes that you have an iterator in position)	t + 2p
Find (average item)	n/2	0
Change/replace item	C	0
Remove first	C	t + 2p (recovered)
Remove last	C	t + 2p (recovered)
Remove in middle	C (assumes that you have an iterator in position)	t + 2p (recovered)
Container overhead		(2p+i) + N(t+2p)
Comments	Allocation with each insert Iterate in either direction	Grows or shrinks with each item Larger than Slist

Ordered Vectors

operation	time cost;	space cost
Insert at end	C (amortized)	t (amortized)
Insert in middle	N/2	t (amortized)
Find (average item)	N/2	0
Change/replace item	C	0
Remove first	N	0
Remove last	C	0
Remove in middle	N/2	0
Container overhead		(Mt+ p + 2i) + 0
Comments	No iterators (use size_t index) Allocation only when the vector grows.	Expands as needed by adding space for many entries at once. Shrinks only via resize()

Key to the comparison tables

N	M	t	i	p	C
count of items	count of space for items	sizeof (item)	sizeof (int)	sizeof (void*)	a constant

Sorted Vectors

operation	**time cost;**	**space cost**
Insert	logN + N/2 (average)	t (amortized)
Find (average item)	logN	0
Change/replace item	N	0
Remove first	N	0
Remove last	C	0
Remove in middle	N/2	0
Container overhead		(Mt + p + 2i) + 0
Comments	Insertion happens based on sort order. No iterators (use size_t index) replace requires remove/add sequence to maintain sorting Allocation only when the vector grows.	Expands as needed by adding space for many entries at once. Shrinks only via resize()

Stacks and Queues

operation	time cost;	space cost
Insert at end	C	t + p
Remove (pop)	C	t + p (recovered)
Container overhead		(2p+i) + N(t+p)
Comments:	Implemented as singly -linked list. Templatized version allows choice of container: time and space costs will reflect that choice.	

Key to the comparison tables

N	M	t	i	p	C
count of items	count of space for items	sizeof (item)	sizeof (int)	sizeof (void*)	a constant

Deques

operation	time cost;	space cost
Insert at end	C	t (amortized)
Find (average item)	N/2	0
Change/replace item	C	0
Remove first	C	t (amortized, recovered)
Remove last	C	t (amortized, recovered)
Remove in middle	N/2	t (amortized, recovered)
Container overhead		(Mt + p + i) + 0
Comments	Implemented as circular queue in an array. Allocation only when collection grows	Expands and shrinks as needed, caching extra expansion room with each increase in size

Binary Tree

operation	**time cost;**	**space cost**
Insert	`logN+C`	`2p+t`
Find (average item)	`logN`	`0`
Change/replace item	`2(logN + C)`	`0`
Remove first	`logN + C`	`2p+t` (recovered)
Remove last	`logN + C`	`2p+t` (recovered)
Remove in middle	`logN + C`	`2p+t` (recovered)
Container overhead		`(p+i) + N(2p+t)`
Comments	Insertion happens based on sort order. Allocation with each insert Replace requires remove/add sequence to maintain order Does not automatically remain balanced. Numbers above assume a balanced tree.	Costs same as doubly linked list per item

Key to the comparison tables

`N`	`M`	`t`	`i`	`p`	`C`
count of items	count of space for items	`sizeof (item)`	`sizeof (int)`	`sizeof (void*)`	a constant

(multi)map and (multi)set family

operation	**time cost;**	**space cost**
Insert	`logN+C`	`2p+t`
Find (average item)	`logN`	0
Change/replace item	`2(logN+C)` *or* `C`	0
Remove first	`logN` (worst case)	`2p+t` (recovered)
Remove last	`logN` (worst case)	`2p+t` (recovered)
Remove in middle	`logN` (worst case)	`2p+t` (recovered)
Container overhead	re-balance may occur at each `insert` or `remove`	`(3p+i) + N(2p+t)`
Comments	Insertion happens based on sort order. Allocation with each insert Replace for sets requires remove/insert. For maps the value is copied in place. implemented as balanced (red-black) binary tree.	

Key to the comparison tables

`N`	`M`	`t`	`i`	`p`	`C`
count of items	count of space for items	`sizeof (item)`	`sizeof (int)`	`sizeof (void*)`	a constant

RWBTree, RWBTreeDictionary[1]

operation	**time cost;**	**space cost**
Insert	`logN+C`	`2p + t +` *small* (fully amortized)
Find (average item)	`logN`	`0`
Change/replace item	`2logN+2 or C`	`0`
Remove first	`2logN(log2(ORDER))+C` (worst case)	`2p+t` (recovered)
Remove last	`2logN(log2(ORDER))+C` (worst case)	`2p+t` (recovered)
Remove in middle	`2logN(log2(ORDER))+C` (worst case)	`2p+t` (recovered)
Container overhead	Re-balance may occur at each insert or remove. However it will happen less often than for a balanced binary tree.	This depends on how fully the nodes are packed. Each node costs `ORDER(2p+t+1)+i` and there will be no more than `2N/ORDER`, and no fewer than `min(N/ORDER,1)` nodes. Inserting presorted items will tend to maximize the size. Sedgewick says the size is close to `1.44 N/ORDER` for random data
Comments	Insertion based on sort order. The logarithm is approximately base `ORDER` which is the splay of the b-tree. (In fact the base is between `ORDER` and `2ORDER` depending on the actual loading of the b-tree) Replace for b-tree requires remove then insert. For `btreedictionary` the value is copied in place	

1. RWBTreeOnDisk has complexity *similar* to RWBTreeDictionary, but the time overhead is much greater since "following linked nodes" becomes "disk seek;" and the size overhead has a much greater impact on disk storage than on core memory.

Hash-based Collections[1]

operation	**time cost;**	**space cost**
Insert	`C`	`p+t`
Find (average item)	`C`	`0`
Change/replace item	`C` *or* `2C`	`0`
Remove	`C`	`p+t` (recovered)
Container overhead		`((M+2)p+i) + N(p+t)` *(1)* `(Mp+(2p+i)b_used) + N(p+t)` *(2)* *1:* standard compliant version *2:* `b_used` is "number of used slots" for the "V6.1" hashed collections
Comments	Insertion happens based on the hashing function. Constant time costs assume that the items are well scattered in the hash slots. Worst case is linear in the number of items per slot. Replace for dictionary or map: The new value is copied in place. Otherwise, requires remove then insert.	Does not automatically resize. We recommend that the number of items be between one half and double the number of slots for most uses.

Key to the comparison tables

`N`	`M`	`t`	`i`	`p`	`C`
count of items	count of space for items	`sizeof (item)`	`sizeof (int)`	`sizeof (void*)`	a constant

1. *RWSet* and *RWIdentitySet* as well as collections with "Hash" in their names.

Typedefs and Macros

Constants:

```
#define FALSE    0                      // RWBoolean value  (defs.h)
#define TRUE     1                      // RWBoolean value  (defs.h)
#define rwnil    0                      // nil pointer (defs.h)
#define RWTOOLS 0x700                   // (The actual current version
                                        //  number) (tooldefs.h)
const RWoffset RWNIL = -1L;             // “no offset” in an RWFile
                                        // (defs.h)
const size_t RW_NPOSRW_NPOS=~(size_t)0; // “not found” as index into
                                        // array (defs.h)
```

Typedefs:

```
typedef   unsigned short   RWClassID;       // (defs.h)      Unique for each
                                            //               class
typedef   int              RWBoolean;       // (defs.h)      TRUE or FALSE
typedef   unsigned char    RWByte;          // (defs.h)      Bitflag atomic
typedef   RWCollectable*   RWCollectableP   // (tooldefs.h)  Needed for
                                            //               tokenizing
typedef   unsigned short   RWErrNo          // (defs.h)      Used in error
                                            //               handler
typedef   long             RWoffset;        // (tooldefs.h)  Used for file
                                            //               offsets
typedef   unsigned long    RWspace;         // (tooldefs.h)  Used for file
                                            //               records
```

```
typedef   long            RWstoredValue;   // (tooldefs.h) Used for file
                                           //              offsets
typedef   void*           RWvoidRWvoid;    // (tooldefs.h) For arrays of
                                           //              void*'s
```

Pointers to Functions:

```
typedef   void          (*RWapplyCollectable)    (RWCollectable*, void*);
typedef   void          (*RWapplyGeneric)        (void*, void*);
typedef   void          (*RWapplyKeyAndValue)    (RWCollectable*,
                                                  RWCollectable*, void*);
typedef   void          (*RWauditFunction)       (unsigned char, void*);
typedef   void          (*RWdiskTreeApply)       (const char*,
                                                  RWstoredValue, void*);
typedef   int           (*RWdiskTreeCompare)     (const char*, const char*,
                                                  size_t);
typedef   RWBoolean     (*RWtestGeneric)         (const void*, const void*);
typedef   RWBoolean     (*RWtestCollectable)     (const RWCollectable*,
                                                  const void*);
typedef   RWBoolean     (*RWtestCollectablePair) (const RWCollectable*,
                                                  constRWCollectable*,void*);
typedef   RWCollectable* (*RWuserCreator)        ();
```

Enumerations:

```
enum RWSeverity {RWWARNING, RWDEFAULT, RWFATAL}
```

The following modify the behavior of member functions or constructors for the classes involved. The value in **bold** font is the default.

```
RWCString::enum stripType       {leading,trailing,both}   // where to strip
                                                          // characters
RWCString::enum caseCompare     {exact, ignoreCase}       // ignore case during
                                                          // comparison
RWCString::enum scopeType       {one, all}                // replace how many
                                                          // substrings
RWBTreeOnDisk::enum styleMode   {V6Style, V5Style}        // file format
RWBTreeOnDisk::enum createMode  {autoCreate, create}      //(reuse,make new)
                                                          // btree in file
```

```
RWeostream::enum Endian         { LittleEndian,
                                  BigEndian, HostEndian }  // constructor
                                                           // argument
RWLocale::enum CurrSymbol       { NONE, LOCAL, INTL }    // used in “asString”
                                                           // methods.
RWWString::enum stripType       {leading,trailing,both}  // where to strip
                                                           // characters
RWWString::enum caseCompare     {exact, ignoreCase}      // ignore case
                                                           // during comparison
RWWString::enum scopeType       {one, all}               // replace how many
                                                           // substrings
```

Tools.h++ Public Macros

These macros are defined to be used by programmers as part of the *Tools.h++* API.

In file collect.h

```
// Macro bodies are removed. See RWcollectable in Class Reference
//and User's Guide.
#define RWDECLARE_ABSTRACT_COLLECTABLE(className)
#define RWDEFINE_ABSTRACT_COLLECTABLE(className)
#define RWDECLARE_COLLECTABLE(className)
#define RWDEFINE_COLLECTABLE(className,id)
#define RWDEFINE_NAMED_COLLECTABLE(className,str)
```

In file defs.h

```
// Defined as shown when RWDEBUG is defined, otherwise defined to nothing.
#define RWPOSTCONDITION(a)    assert( (a) != 0 )
#define RWPRECONDITION2(a,b)  assert( (a) != 0 )
#define RWPOSTCONDITION2(a,b) assert( (a) != 0 )
#define RWPRECONDITION2(a,b)  assert((b, (a) !=0))
#define RWPOSTCONDITION2(a,b) assert((b, (a) !=0))
#define RWASSERT(a)           assert( (a) != 0 )
```

In file edefs.h

```
// Macro bodies removed. See section on persistence.
#define RWDECLARE_PERSISTABLE_IO(CLASS,ISTR,OSTR)
#define RWDECLARE_PERSISTABLE_TEMPLATE_IO(TEMPLATE, ISTR, OSTR)
#define RWDECLARE_PERSISTABLE_TEMPLATE_IO_2(TEMPLATE, ISTR, OSTR)
#define RWDECLARE_PERSISTABLE_TEMPLATE_IO_3(TEMPLATE, ISTR, OSTR)
#define RWDECLARE_PERSISTABLE_TEMPLATE_IO_4(TEMPLATE, ISTR, OSTR)
#define RWDECLARE_PERSISTABLE(CLASS)
#define RWDECLARE_PERSISTABLE_TEMPLATE(TEMPLATE)
#define RWDECLARE_PERSISTABLE_TEMPLATE_2(TEMPLATE)
#define RWDECLARE_PERSISTABLE_TEMPLATE_3(TEMPLATE)
#define RWDECLARE_PERSISTABLE_TEMPLATE_4(TEMPLATE)
```

In file epersist.h

```
// Macro bodies removed. See section on persistence.
#define RWDEFINE_PERSISTABLE_IO(CLASS,ISTR,OSTR)
#define RWDEFINE_PERSISTABLE_TEMPLATE_IO(TEMPLATE,ISTR,OSTR)
#define RWDEFINE_PERSISTABLE_TEMPLATE_IO_2(TEMPLATE,ISTR,OSTR
#define RWDEFINE_PERSISTABLE_TEMPLATE_IO_3(TEMPLATE,ISTR,OSTR)
#define RWDEFINE_PERSISTABLE_TEMPLATE_IO_4(TEMPLATE,ISTR,OSTR)
#define RWDEFINE_PERSISTABLE(CLASS)
#define RWDEFINE_PERSISTABLE_TEMPLATE(TEMPLATE)
#define RWDEFINE_PERSISTABLE_TEMPLATE_2(TEMPLATE)
#define RWDEFINE_PERSISTABLE_TEMPLATE_3(TEMPLATE)
#define RWDEFINE_PERSISTABLE_TEMPLATE_4(TEMPLATE)
```

In file strmshft.h

```
// Convenience macros.
#define RW_PROVIDE_DVSTREAM_INSERTER(DerivedOstream,vstreamable)
#define RW_PROVIDE_DVSTREAM_EXTRACTOR(DerivedIstream,vstreamable)
```

In files tphasht.h, tvhasht.h, tphdict.h, tvhdict.h, tphmmap.h, tvhmmap.h, tphset.h, tvhset.h

```
// Useful when writing code portable between “current”
// and “ANSI-compliant” compilers
// See templates.
#define RWDefHArgs(T) ,RWTHasher<T>,equal_to<T>
```

In files tpsrtvec.h, tvsrtvec.h

```
// Useful when writing code portable between “current”
// and “ANSI-compliant” compilers
// See templates.
#define RWDefCArgs(T) ,less<T>
```

Standard Smalltalk Interface (activated by defining RW_STD_TYPEDEFS):

```
typedef   RWBag                       Bag;
typedef   RWBagIterator               BagIterator;
typedef   RWBinaryTree                SortedCollection;
typedef   RWBinaryTreeIterator        SortedCollectionIterator;
typedef   RWBitVec                    BitVec
typedef   RWCollectable               Object;   // All-too-common type!
typedef   RWCollectableDate           Date;
typedef   RWCollectableInt            Integer;
typedef   RWCollectableString         String;
typedef   RWCollectableTime           Time;
typedef   RWCollection                Collection;
typedef   RWHashDictionary            Dictionary;
typedef   RWHashDictionaryIterator    DictionaryIterator;
typedef   RWIdentityDictionary        IdentityDictionary;
typedef   RWIdentitySet               IdentitySet;
typedef   RWOrdered                   OrderedCollection;
typedef   RWOrderedIterator           OrderedCollectionIterator;
typedef   RWSequenceable              SequenceableCollection;
typedef   RWSet                       Set;
typedef   RWSetIterator               SetIterator;
typedef   RWSlistCollectables         LinkedList;
typedef   RWSlistCollectablesIterator LinkedListIterator;
typedef   RWSlistCollectablesQueue    Queue;
typedef   RWSlistCollectablesStack    Stack;
```

B

Messages

Error messages are created by the macro `DECLARE_MSG` in files `coreerr.cpp` and `toolerr.cpp`..

Table C-1 Core messages.
These are messages used by all Rogue Wave libraries. The symbols are defined in <rw/coreerr.h>. These messages belong to category "rwcore7.0"

Symbol	***Message***
CORE_EOF	"[EOF] EOF on input"
CORE_GENERIC	"[GENERIC] Generic error number %d; %s"
CORE_INVADDR	"[INVADDR] Invalid address: %lx"
CORE_LOCK	"[LOCK] Unable to lock memory"
CORE_NOINIT	"[NOINIT] Memory allocated without being initialized"
CORE_NOMEM	"[NOMEM] No memory"
CORE_OPERR	"[OPERR] Could not open file %s"
CORE_OUTALLOC	"[OUTALLOC] Memory released with allocations still outstanding"
CORE_OVFLOW	"[OVFLOW] Overflow error -> \"%.*s\" <- (%u max characters)"
CORE_STREAM	"[STREAM] Bad input stream"
CORE_SYNSTREAM	"[SYNSTREAM] Syntax error in input stream: expected %s, got %s"

.

Table C-2 Tools.h++ messages.
These are messages used by the Tools.h++ library. The symbols are defined in <rw/toolerr.h>. These messages belong to category "rwtool7.0"

Symbol	***Message***
TOOL_ALLOCOUT	"[ALLOCOUT] %s destructor called with allocation outstanding"
TOOL_BADRE	"[BADRE] Attempt to use invalid regular expression"
TOOL_CRABS	"[CRABS] RWFactory: attempting to create abstract class with ID %hu (0x%hx)"
TOOL_FLIST	"[FLIST] Free list size error: expected %ld, got %ld"
TOOL_ID	"[ID] Unexpected class ID %hu; should be %hu"
TOOL_INDEX	"[INDEX] Index (%u) out of range [0->%u]"
TOOL_LOCK	"[LOCK] Locked object deleted"
TOOL_LONGINDEX	"[LONGINDEX] Long index (%lu) out of range [0->%lu]"
TOOL_MAGIC	"[MAGIC] Bad magic number: %ld (should be %ld)"
TOOL_NEVECL	"[NEVECL] Unequal vector lengths: %u versus %u"
TOOL_NOCREATE	"[NOCREATE] RWFactory: no create function for class with ID %hu (0x%hx)"
TOOL_NOTALLOW	"[NOTALLOW] Function not allowed for derived class"
TOOL_READERR	"[READERR] Read error"
TOOL_REF	"[REF] Bad persistence reference"
TOOL_SEEKERR	"[SEEKERR] Seek error"
TOOL_STREAM	"[STREAM] Bad input stream"
TOOL_SUBSTRING	"[SUBSTRING] Illegal substring (%d, %u) from %u element RWCString"
TOOL_UNLOCK	"[UNLOCK] Improper use of locked object"
TOOL_WRITEERR	"[WRITEERR] Write error"

Bibliography

Ammeraal, L. *Programs and Data Structures in C*, John Wiley and Sons, 1989, ISBN 0-471-91751-6.

Barton, John J., and Lee R. Nackman, *Scientific and Engineering C++*, Addison-Wesley, 1994, ISBN 0-201-53393-6

Booch, Grady, *Object-Oriented Design with Applications*, Second Edition, The Benjamin Cummings Publishing Company, Inc., 1994, ISBN 0-8053-5340-2.

Budd, Timothy, *Classic Data Structures in C++*, Addison-Wesley, 1993, ISBN 0-201-50889-3.

Budd, Timothy, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991, ISBN 0-201-54709-0.

Carrol, Martin D. and Margaret A. Ellis, *Designing and Coding Reusable C++*, Addison-Wesley, 1995, ISBN 0-201-51284-X.

Coplien, James O., *Advanced C++, Programming Styles and Idioms*, Addison-Wesley, 1992, ISBN 0-201-54855-0.

Eckel, Bruce, *C++ Inside and Out*, McGraw-Hill, Inc, 1993, ISBN 0-07-881809-5.

Eckel, Bruce, *Thinking in C++*, Prentice Hall, Inc, 1995, ISBN 0-13-917709-4.

Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990, ISBN 0-201-51459-1.

D

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1995, ISBN 0-201-63361-2.

Goldberg, Adele and David Robson, *Smalltalk-80, The Language*, Addison-Wesley, 1989, ISBN 0-201-13688-0.

Gorlen, Keith, *The NIH Class Library*, Computer Systems Laboratory, DCRT, National Institutes of Health, Bethesda, MD 20892.

Gorlen, Keith, Sanford M. Orlow and Perry S. Plexico, *Data Abstraction and Object-Oriented Programming in C++*, John Wiley and Sons, 1990, ISBN 0-471-92346 X.

Khoshafian, Setrag and Razmik Abnous, *Object orientation: Concepts, Languages, Databases, User Interfaces*, John Wiley and Sons, 1990, ISBN 0-471-51802-6.

Knuth, Donald E., *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, 1973, ISBN 0-201-03803.

Lippman, Stanley B. *C++ Primer*, Second Edition, Addison-Wesley, 1991, ISBN 0-201-54848-8.

Maguire, Steve, *Writing Solid Code*, Microsoft Press, 1993, ISBN 1-55615-551-4.

McConnell, Steve, *Code Complete*, Microsoft Press, 1993, ISBN 1-55615-484-4.

Meyer, Bertrand, *Object-oriented Software Construction*, Prentice-Hall, 1988, ISBN 0-13-629049-3.

Meyers, Scott, *Effective C++*, Addison-Wesley, 1992, ISBN 0-201-56364-9.

Murray, Robert B. *C++ Strategies and Tactics*, Addison-Wesley, 1993, ISBN 0-201-56382-7.

Petzold, Charles, *Programming Windows*, Microsoft Press, Third Edition, 1992, ISBN 1-55615-395-3.

Portable Operating System Interface (POSIX), Part 1: System Application Program Interface, International Standard ISO/IEC 9945-1: 1990, IEEE Std 1003.1-1990, ISBN 1-55937-061-0.

Portable Operating System Interface (POSIX), Part 2: Shell and Utilities, Vol. 1, International Standard ISO/IEC 9945-2: 1993, ANSI/IEEE Std 1003.2-1992, ISBN 1-55937-255-9.

Pree, Wolfgang, *Design Paterns for Object-Oriented Software Development*, Addison-Wesley, 1994, ISBN 0-201-42294-8.

Sedgewick, Robert. *Algorithms in C++*, Addison-Wesley, 1990, ISBN 0-201-51059-6.

Stroustrup, Bjarne. *The C++ Programming Language*, Second Edition, Addison-Wesley, 1991, ISBN 0-201-53992-6.

Stroustrup, Bjarne. *The Design and Evlolution of C++*, Addison-Wesley, 1994, ISBN 0-201-54330-3.

Taligent, Inc. *Taligent's Guide to Designing Programs*, Taligent Press, 1994, ISBN 0-201-40888-0.

Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++, DOC No. X3J16/96-0018, WG21/NO836, Date: 26 January 1996, Accredited Standards Committee X3, Information Processing Systems, operating under the procedures of the American National Standards Institute (ANSI).

D

Index

L

M

N

O

P

R

S

T

V

W

X