Tools.h++ Class Library

Introduction and Reference Manual

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 802-3046-10
Revision A, November, 1995

Adobe PostScript

Contents

Part 1— User's Guide

Introduction 1

Overview and Features

Tools.h++ is a rich, robust, and versatile C++ foundation class library. By "foundation" we mean a set of classes useful for virtually any programming chore.

Tools.h++ is an industry standard. It has been chosen by a wide variety of compiler vendors to be the standard library they include with every copy of their compiler. It has been ported to dozens of compilers, many operating systems, and is in use by thousands of users world wide. You can count on Tools.h++ being available on whatever platform you might chose to program.

The package includes:

- **Powerful single, multibyte, and wide character support**
  Class `RWCString` offers a full suite of operators and functions to manipulate single- and multibyte character strings. Class `RWWString` offers wide character string manipulation. Both classes make it easy to do concatenation, comparison, indexing (with optional bounds checking), I/O, case changes, stripping, and many other functions. Classes `RWCSubString` and `RWWSubString` allow extraction and assignment to substrings. Class `RWCRegexp` supports regular expression pattern searches. Classes `RWCTokenizer` and `RWWTokenizer` can be used to break single and wide character strings, respectively, into separate tokens.

- **Time and date handling classes**
  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; *etc.*
- **Internationalization support**
  `Class RWLocale` provides a convenient and easy-to-use framework for internationalizing your software. Class `RWTimeZone` makes manipulating time zones and daylight savings time easy.
- **Persistent store**
  A powerful and sophisticated persistent store facility that allows complex objects inheriting from class `RWCollectable` to be stored to disk or even exchanged through a heterogeneous network. The object's morphology (e.g., multiple pointers to the same object, or a circularly linked list) is maintained.
- **Template based classes**
  A complete set of collection classes based on C++ templates. These include singly- and doubly-linked lists, stacks, queues, ordered collections, hash tables, sets, dictionaries, etc.
- **Generic collection classes**
  For compilers that do not support templates yet, 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 enough to the template-based classes that if you decide to make the transition, it will be easy.
- **Smalltalk-like collection classes**
  A complete library of collection classes, modeled after the `Smalltalk-80` programming environment: `Set, Bag, OrderedCollection, SortedCollection, Dictionary, Stack, Queue`, *etc.* All of these classes can be saved to disk and restored in a new address space, even on a different operating system! An object need only inherit the abstract base class `RWCollectable` to have full access to all of the functionality of the collection classes. The interface to the `Smalltalk` names is done as typedefs, allowing the actual implementation of, say, a `Dictionary` to be changed from its default of a hash table of associations, to, say, a B-Tree.
- **RWFile class**
  Encapsulates standard file operations.
- **B-Tree disk retrieval**
  Efficient keyed access of disk records, using B-Trees.

- **File Space Manager**
  Allocates, deallocates and coalesces free space within a file.
- **Still more classes...**
  Bit vectors, virtual I/O streams, caching managers, virtual arrays, *etc*.
- **A complete error handling facility**
  Including a facility for handling exceptions.
- **Online Examples**
  The source code for examples used in the chapter “Examples” is typically found in `/opt/SUNWspro/examples/Tools.h++`, depending on where you installed C++. Also included are sample input and output files, if needed, and makefiles.

Supported C++ compilers

Tools.h++ can be compiled without modification by a wide variety of C++ compilers. Contact Rogue Wave Software for further information.

Philosophy

The C++ language has several design goals that set it apart from most other object-oriented languages. The first and foremost is efficiency: it is possible to write production quality code that is every bit as efficient and small as code that has been written in C, yet more maintainable. A second is a "less is more" philosophy: no feature has been included in the language that will make non-users of the feature suffer. For example, you will not find built-in garbage collection. The result is a skeletal, lean-and-mean language (at least as far as object-oriented languages go) that compiles fast and results in small, efficient, but not terribly general, code.

Towards getting the best out of the language, the Tools.h++ class library has adopted similar goals: Efficiency, simplicity, compactness, and predictability.

Efficiency. In general, you will find no feature that will slow things down for the non-user 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, we offer a choice between classes with extreme simplicity, but little generality, and classes that are a little more complex, but more general. We have chosen not to require that all classes inherit a secular base class (such as the class `Object` used by Smalltalk and *The NIH Classes*). Instead, only objects that are to be collected using the Smalltalk-like collection classes need inherit a base class `RWCollectable`. The advantage of this is that virtual base classes are not necessary, simplifying the inheritance tree and the difficult problem of casting from a base class back to its derived class.

Simplicity. There is a temptation to keep adding subclasses to add power or convenience for the user or to demonstrate one's prowess with the language. We have avoided this. Although the overall architecture is sophisticated and integrated, each class usually plays just one well-defined, pivotal role. Most functions are extraordinarily simple: a few lines of code.

Compactness. An important goal was to make sure that programs compiled small. This was to insure that they can be used in embedded systems.

Predictability. Many new users of C++ become so giddy with the power of being able to overload esoteric operators like "&=" that they forget about tried-and-true function calls and start redefining everything in sight. Again, we have avoided this and have tried hard to make all of the familiar operators work just as you might expect—there are no surprises. This approach gives great symmetry to the class library, making it possible to do such things as, say, change the implementation of a `Dictionary` from a hash table to a `B-Tree` with impunity.

In general, whenever we considered a new feature, we tried to think as Stroustrup would: if there was already a way to do it, we left it out!

Conventions

All class names start with the letters "RW". All function names start with a lower case letter, but subsequent words are capitalized. There are no underline characters used in names. An example of a class name is `RWHashDictionary`, of a function `compareTo()`. Generally, abbreviations are not used in function names, making them easy to remember.

Some of the class names may seem unnecessarily tedious (for example, a singly-linked list of collectables, accessible as a stack, is a `RWSlistCollectablesStack`). There are two reasons for this. First, using generic names like "Stack" invites a name collision with someone else's stack class, should you write a large program that combines many class libraries. We have tried very hard to avoid polluting the global name space with generic names like "String", "Vector", "Stack", *etc.* It is also for this reason that class names and many of the potentially generic names have an "RW" prepended (*e.g.*, "RWBoolean"). We have worked hard to make sure that Tools.h++ is compatible with other class libraries. Secondly, the names are intended to convey as precisely as possible what the class does.

Nevertheless, there is a set of typedefs that give these various classes generic names like `Stack` or `OrderedCollection` that are consistent with the Smalltalk-80 names.

Reading this manual

This manual is intended to serve two purposes: to be an introduction to using the Tools.h++ *Class Library* and to be an intermediate-level tutorial on the language C++, using the *Class Library* as an aid to discussing some of the more subtle aspects of C++. It assumes that you are familiar with the basics of C++, but not yet an expert. The discussion is generally more detailed than what is necessary to actually use the library—if you find yourself getting overwhelmed by details, by all means, abandon the coming chapters and rely on the many examples provided with the library.

If you are not familiar with C++ at all, we do not recommend trying to learn it from the three definitive reference books available: *Stroustrup* (1991), *Lippman* (1989), and *Ellis and Stroustrup* (1990; sometimes ominously referred to as "The ARM"—Annotated Reference Manual). Their terse (but precise) style make them better suited as references to the language.

In what follows, there are several references to Stroustrup and Lippman's books—it may be helpful to have a copy available.

Occasionally there will be a highlighted paragraph explaining either a key point, or *"An Aside"*. The latter can be ignored safely without losing the essential thread of the discussion.

Throughout this manual, Class names, examples, operating system commands, and code fragments are shown in a `courier` font. Vertical ellipses are used to indicate that some part of the code is missing:

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

Getting Started with Tools.h++ 2

This chapter will get you started using some commonly used Tools.h++ classes. For complete information on the classes discussed, read the appropriate section in this manual.

Compiling a program using `make`

To compile any of the programs in this chapter using `make,`create a file in your test directory, named `Makefile`, and insert these lines:

```
LDLIBS += -lrwtool
.KEEP_STATE:
```

Then compile any of the programs in this chapter with:

```
%make test
```

where `test.cc` is the name of your program.

Compiling a program from the command line

To compile `test.cc` from the command line type:

```
%CC test.cc -lrwtool
```

The resulting file, `a.out`, is executable.

Pointer based classes vs. value base classes

Tools.h++ has both pointer based and value based container templates. A pointer based container class `RWTPtrCont<MyClass>`, stores a pointer to `MyClass`. A value based container class `RWTValCont<MyClass>` physically contains objects of `MyClass`. Using `Ptr` classes will typically reduce the size of the object code of your application. The `Val` classes are generally better for containing objects for built-in types such as `int`, `char`, or `float`. For more on collections, see Section , “Types of templates,” on page 96.

Strings

Class: `RWCString`

This class is cleaner and easier to use than the C `char*` type. You can efficiently pass `RWCStrings` by value as well as by reference. Here’s an example:

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

void main () {
   RWCString ss("Cheer = ");
   for (int i=1; i<=10; i++) {
       ss += "Go! ";
   }
   cout << ss << endl;
}
```

prints:

```
Cheer = Go! Go! Go! Go! Go! Go! Go! Go! Go! Go!
```

For a full discussion of `RWCString`, see Chapter 6, “Strings”.

See also:

`RWCSubString`, `RWCRegexp`, `RWWString`, `RWCTokenizer`, `RWWSubString`, and `RWWTokenizer`.

Vectors

Classes: `RWValOrderedVector<T>`, `RWTPtrVector<T>`

These classes provide you with vectors with insertion and deletion. `RWValOrderedVector` is value-based, while `RWTPtrVector` is pointer-based. Here’s an example:

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

void main () {
   RWTValOrderedVector<int> sv;
   for (int i=0; i<10; i++) {
     sv.insert(10-i);
   }
   cout << "Found 3 at " << sv.index(3) << endl;
}
```

prints:

```
Found 3 at 7
```

For more on vector classes, see Chapter 23, “Templates”.

See also:

`RWT*Vector`, `RWT*SortedVector`, `RWTBitVec`, `RWTValVirtualArray`, and related Smalltalk-like classes.

Hash tables

Classes: `RWTValHashDictionary<K,V>`, `RWTPtrHashDictionary<K,V>`

These classes provide hash tables with both a key and value. The following example illustrates how to set up a hash table of names and associated birthdays, using the name as the key:

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

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

main()
{
  RWTValHashDictionary<RWCString, RWDate> birthdays(hashString);

  birthdays.insertKeyAndValue("John", RWDate(12, "April",
1975));
  birthdays.insertKeyAndValue("Ivan", RWDate(2, "Nov", 1980));

  // Alternative syntax:
  birthdays["Susan"] = RWDate(30, "June", 1955);
  birthdays["Gene"] = RWDate(5, "Jan", 1981);

  // Print a birthday:
  cout << birthdays["John"] << endl;
  return 0;
}
```

prints:

```
April 12, 1975
```

For more on hash table classes, see Chapter 23, “Templates”.

See also:

`RWT*HashSet`, `RWT*HashTable`, their iterators, and related Smalltalk-like classes.

Linked lists

Classes: `RWTValSlist<T>`, `RWTValSlistIterator<T>`, `RWTPtrSlist<T>`, `RWTPtrSlistIterator<T>`

These classes provide singly-linked lists. You manage storage belonging to your objects. Tools.h++ manages the storage it uses to store the lists and list nodes. Unless you use the "intrusive" list classes, use the iterator classes to move incrementally through the lists rather than trying to access the list nodes directly.

In this example notice the use of `new` and `*it.key` even though the template parameter is plain `RWCString` with no `*`.

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

void main()
{
   RWTPtrSlist<RWCString> strings;

   RWCString ss("+");
   for (int i=0; i<10; i++) {
       strings.insert(new RWCString(ss));
       ss += "+";
   }

   RWTPtrSlistIterator<RWCString>it(strings);
   while (++it) cout << *it.key() << endl;
}
```

prints:

```
+
++
+++
++++
+++++
++++++
+++++++
++++++++
+++++++++
++++++++++
```

For more on lists, see Chapter 23, “Templates”.

See also:

`RWTIsvSlist` (an "intrusive" list type), `RWT*Dlist`, their iterator classes, and similar list classes in the Smalltalk-like classes. `RWT*HashSet`, `RWT*HashTable`, their iterators, and related Smalltalk-like classes.

Compiling and Debugging 3

This section will tell you how to compile and debug programs using the Tools.h++ class library.

Compiling a program

Consider the following simple program called `test.cc` (C++ and make both recognize this suffix for C++ source files.):

```
#include <rw/rwdate.h>
#include <rw/rstream.h>
main()
{
    // Construct a date with today’s date:
    RWDate date;

    // Print it out:
    cout << date << endl;
    return 0;
}
```

Suppose the directory also contains a Makefile that looks like:

```
# For smarter tracking of dependencies
.KEEP_STATE:

# Link definitions from Tools.h++ as needed:
LDLIBS=-lrwtool
```

You could do:

```
% ls
Makefile     test.cc
% make test
CC -o test test.cc -lrwtool
% test
May 7, 1993
```

The header files for Tools.h++ are installed in a default location for header files and the library `librwtool.a` is in a default location for libraries so the compiler needs no further directions to find them.

All header files are specified as (using `rwdate.h` as an example):

`<rw/rwdate.h>`

with a leading `rw`. This is done to avoid potential naming conflicts with header files from other libraries.

You may want to look at header files from Tools.h++ or some other library. The C++ compiler will print out the name of every header file it reads if you pass it the `-H` option on the command line, for example:

```
% CC -H -c test.cc
```

Specialized compilation and linking options

Tools.h++ uses Solaris to support you in writing applications that are sensitive to different international locales, including Japanese and other wide character sets. CC automatically links with system library `-lw` (w for wide). Use `CC -v` to see detailed information about compilation and linking.

You can also use the Tools.h++ library to use in multithreaded applications. If your application is multithreaded, compile and link it with the `-mt` option. Tools.h++ uses safe system facilities, and has enough internal locking to maintain its own integrity. For example, strings work correctly even though they are implemented with hidden internal sharing of data. If you wish to share Tools.h++ objects among threads in your address space, you must put locks in your application. Your locks should prevent the same object from being operated on by more than one thread (or lwp) at the same time.

The `librwtool.a` library can be used by both single-threaded and multi-threaded applications. When linked with -mt certain entry points are resolved by `libthread`. libC supplies adequate entry points for single-threaded applications. A profiling version of `librwtool.a` is supplied with this release, supporting both `prof` and `gprof`.

Debugging a Program

While your project is in development mode you may wish to use the debugging and runtime chacking features of Tools.h+. By compiling with `RWDEBUG=1` and linking with `-lrwtool_dbg` rather than `-lrwtool` you turn on runtime checks including validity of arguments you pass to many of the Tools.h++ library functions. See Chapter 20, "Implementation Notes" for more information.

Definitions of member functions in template classes are encrypted to protect Tools.h++ source code in the binary distribution. If you have the full Tools.h++ source code you may compile the entire library with `-g` and debug with full source available.

Debugging with the SPARCworks Debugger or dbx

To debug your program with the SPARCworks debugger or dbx, compile your application with the `-g` option, and run it under the debugger. Both the SPARCworks debugger and dbx provide high-level debugging to your source code and they support C++. See *Debugging a Program* in the SPARCworks documentation for more information.

The print and display commands give high-level access to Tools.h++ objects. These commands let you execute a C++ expression and see the value of it. The expression can include calls to C++ functions that are visible in the current scope.

Suppose you are debugging a program `dirpart` that consists of two files:

dirpart1.cc:

```
#include <iostream.h>
#include <rw/cstring.h>
RWCString dirPart(RWCString);
main(int argc, char** argv) {
  cout << dirPart(argv[0]) << endl;
  return 0;
}

void db() {
   RWCString s("foo");
   s[0];
   s.length();
   s.data();
}
```

dirpart2.cc:

```
#include <rw/cstring.h>
#include <rw/regexp.h>
RWCString dirPart(RWCString arg) {
   RWCRegexp tail("/[^/]*$");
   RWCString dir(arg);
   dir(tail) = "";
   return dir;
}
```

This program is supposed to print just the “directory part” of the path used to run it.

Suppose you set a breakpoint in the function `dirPart` and want to examine its argument `arg`. You can simply call non-inline member functions compiled with `-g`. An inline function is not considerd part of the global scope in C++ so it may not be visible if the current file contains no use of it. Member functions in a library not compiled with `-g` such as Tools.h++ may also require the same special technique:

```
(debugger) stop in dirPart
(2) stop in dirPart(RWCString)
(debugger) run
Running: dirpart
(process id 19954)
```

In class `RWCString` the function `data` returns a `const char*` pointing to the actual characters in the string. Notice that the function `db` in `dirpart1` uses `data`. To access the `data` of `arg`, you could do this:

```
(debugger) file dirpart1.cc
(debugger) print ‘dirPart‘arg.data()
arg.data() = "dirpart"
```

or if your version of the debugger supports it:

```
(debugger) print arg.‘dirpart1.cc‘RWCString::data()
arg.data() = "dirpart"
```

The function `db` in `dirpart1` contains calls to some member functions you might want to use in debugging.

Class Overview 4

This section gives an overview of the library and highlights some of the points of commonality between 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 do not depend on other classes for their implementation or semantics: they can be pulled out and used just one or two at a time. In addition, the library also includes a rich set of *abstract classes* that define an interface for persistence, internationalization, and other issues, and a set of *implementation classes* that implement these interfaces.

Various kinds of collection classes are also a central feature of Tools.h++. These fall into three groups:

- Template collection classes;
- Generic collection classes;
- Smalltalk-like collection classes.

Regardless of their implementation, all collection classes generally follow the Smalltalk naming conventions and semantical model: `SortedCollection`, `Dictionaries`, `Bags`, `Sets`, *etc*. They all use similar interfaces, allowing for them to be interchanged easily. The first two kinds of classes (Template and Generic) will work with any kind of object. The last ("Smalltalk-like collection classes") requires that all collected items inherit from a cosmic object.

"Information flow" on page 25 lists the hierarchy of all the public Tools.h++ classes. In addition, there are some other classes that are used internally by the library.

Concrete classes

The concrete classes consist of

- A set of *simple classes* (such as dates, times, strings, etc.), discussed in Chapter 6, "Strings;" Chapter 7, "Using Class RWDate;" and Chapter 8, "Using Class RWTime."
- A set of collection classes based on templates, discussed in Chapter 14, "Templates;"
- A set of collection classes that use the preprocessor `<generic.h>` facilities, discussed in Chapter 15, ""Generic" Collection Classes."

Simple classes

Tools.h++ provides a rich set of lightweight simple classes. By "lightweight" we mean classes with low-cost initializers and copy constructors. Examples include `RWDate` (dates, following the Gregorian calendar), `RWTime` (times, including support for various time zones and locales), `RWCString` (single- and multi-byte strings), `RWWString` (wide character strings), and `RWCRegexp` (regular expressions). Most of these classes are four bytes or less, with very simple copy constructors (usually just a bit copy) and no virtual functions.

Template classes

Template classes offer the advantages of speed and type safe usage. Their code size can also be quite small when used sparingly. Their disadvantage is that when used with many different types, code size can become large because each type effectively generates a whole new class.

Generic collection classes

"Generic collection classes" are so called because they use the `<generic.h>` preprocessor macros supplied with your C++ compiler. They can approximate templates for those compilers that do not support them. As such, they are highly portable. However, because they depend heavily on the preprocessor, they can be difficult to debug.

Abstract data types

The library also includes a set of abstract data types (ADTs), and corresponding specializing classes, that provide a framework for persistence, localization, and other issues.

- Locale, discussed in Chapter 5, "Internationalization;"
- Time zones, discussed in Chapter 5, "Internationalization;"
- Virtual streams, discussed in Chapter 9, "Virtual Streams;"
- A comprehensive Smalltalk hierarchy, discussed in Chapter 16, "Smalltalk-like Collection Classes;"
- Isomorphic persistence, discussed in Chapter 17, "Persistence;"
- Virtual page heaps, discussed in Chapter 22, "Class Reference;"
- Model-View-Controller abstraction, discussed in Chapter 22, "Class Reference."

Smalltalk-like collection classes

The "Smalltalk-like collection classes" are so called because they offer much of the functionality of their Smalltalk namesakes, such as `Bag`, `SortedCollection`, etc. However, they are not slavish imitations and instead pay homage to the strengths and weaknesses of C++. Their greatest advantages are their simple programming interface, powerful I/O abilities, and high code reuse. Their biggest disadvantages are their relatively high object code size when used in only small doses (because of an initially high overhead in code machinery) and their relative lack of type safeness. All objects to be used by the Smalltalk-like collection classes must also inherit from the abstract base class `RWCollectable`.

Common member functions

Whatever category a class might fall into, they all have very similar programming interfaces. This section highlights functionality shared by a number of classes.

Persistence

The following functions are used to store an object of type *`ClassName`* to and from an `RWFile` and to and from the Tools.h++ virtual streams facility and then 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` encapsulates ANSI-C file I/O and is discussed in detail in Chapter 10, “Using Class RWFile” as well as in Chapter 22, “Class Reference.” Objects saved using `RWFile` are saved using a binary format, resulting in efficient storage and retrieval to files.

Classes `RWvistream` and `RWvostream` are abstract base classes used by the Tools.h++ 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, that 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 9, “Virtual Streams,” as well as in Chapter 22, “Class Reference.”

It is up to the user to decide whether to store to `RWFile`’s or virtual streams. The former offers the advantage of speed, but limits portability of the results to the host machine. The latter is not as fast, but there are specializing classes (`RWpistream` , `RWpostream`, `RWXDRistream`, and `RWXDRostream`) that allow the results to be moved to other types of machines.

Store size

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

These functions return the number of bytes of secondary storage necessary to store an object of type *`ClassName`* to an `RWFile` using function

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

They are useful for storing objects using classes `RWFileManager` and `RWBTreeOnDisk`. The second variant, `recursiveStoreSize()`, is used for objects that inherit from `RWCollectable` and can calculate the number bytes used in a recursive store, using function

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

Stream I/O

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

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

Note – this contrasts with the persistence operators

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

(see “Persistence” on page 22) which, although they may store and restore to a stream, 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

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

All of the Tools.h++ classes take a very simple approach: if you allocated something off the heap, then you are responsible for deallocating it. If the Tools.h++ library allocated something off the heap, then it is responsible for deallocating it.

There are two exceptions. The first are 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;
```

that 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.

Both of these exceptions are documented in detail in Chapter 22, "Class Reference."

Information flow

Generally, with the Tools.h++ libraries, information flows into a function via its arguments and out through a return value. Most functions do not modify their arguments. Indeed, if you see an argument being passed as a "const reference", *i.e.*, as follows:

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

or (of course) by value, then you can be confident that the argument will not be modified. However, if an argument is passed as a non-const reference (this is rare) then there is the possibility that the function will modify it.

If an argument is being passed in as a pointer, then 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*);
```

This is 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 have been to pass objects that are to be inserted into a collection by reference (as done by The NIH Classes). We rejected this approach for two reasons: it looks too similar to pass-by-value (making it easy for the programmer to forget about the retained reference) and it becomes too easy to store a reference to a stack-based variable.

Multi-thread safe

Almost all Tools.h++ functions behave in a multi-threaded environment as in a single-threaded environment, provided that the application program either takes care to avoid sharing of individual objects between threads or takes care 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. The few exceptions are noted in the manual.

Eight-bit clean

All classes in Tools.h++ are eight-bit clean. This means that 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 it is `unsigned`, this allows indexes up to 64k minus one on systems where `unsinged` has only 16 bits.

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

Version

The version of Tools.h++ is given by the macro `RWTOOLS`, expressed as a hexadecimal number. For example, version 6.0.1 would be 0x601. This can be used for conditional compilations.

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

The following list is the public class hierarchy of the Tools.h++ classes. Classes which use multiple inheritance are shown in italics. Classes listed in italics use multiple inheritance.

Note – this is the public class hierarchy—the class implementations may use private inheritance.

Public Class Hierarchy

```
RWBench
RWBitVec
RWBTreeOnDisk
RWCacheManager
RWCollectable
   RWCollection
      RWBag
      RWBinaryTree
      RWBTree
         RWBTreeDictionary
      RWSequenceable
         RWDlistCollectables
         RWOrdered
            RWSortedVector
         RWSlistCollectables
            RWSlistCollectablesQueue
            RWSlistCollectablesStack
      RWHashTable
         RWSet
            RWHashDictionary
               RWIdentityDictionary
            RWIdentitySet
   RWCollectableDate
   RWCollectableInt
   RWCollectableString
   RWCollectableTime
   RWModelClient
RWCRegexp
RWCString
RWCSubString
RWCTokenizer
RWDate
RWFactory
RWFile
   RWFileManager
RWGBitVec(size)
RWGDlist(type)
RWGDlistIterator(type)
RWGOrderedVector(val)
RWGQueue(type)
RWGSlist(type)
RWGSlistIterator(type)
```

```
RWGSortedVector(val)
RWGStack(type)
RWGVector(val)
RWInteger
RWIsvDlink<T>
RWIsvSlink<T>
RWIterator
   RWBagIterator
   RWBinaryTreeIterator
   RWDlistCollectablesIterator
   RWSetIterator
      RWHashDictionaryIterator
   RWOrderedIterator
   RWSlistCollectablesIterator
RWLocale
   RWLocaleDefault
   RWLocaleSnapshot
RWModel
RWTime
RWTimer
RWTBitVec<size>
RWTIsvDlist<T>
RWTIsvSlist<T>
RWTPtrDlist<T>
RWTPtrDlistIterator<T>
RWTPtrHashTable<T>
   RWTPtrHashSet<T>
RWTPtrHashTableIterator<T>
RWTPtrHashDictionary<K,V>
RWTPtrHashDictionaryIterator<K,V>
RWTPtrOrderedVector<T>
RWTPtrSlist<T>
RWTPtrSlistIterator<T>
RWTPtrSlistDictionary<K,V>
RWTPtrSlistDictionaryIterator<K,V>
RWTPtrSortedVector<T>
RWTPtrVector<T>
RWTStack<T,C>
RWTQueue<T,C>
RWTValDlist<T>
RWTValDlistIterator<T>
RWTValHashTable<T>
   RWTValHashSet<T>
RWTValHashTableIterator<T>
```

```
RWTValHashDictionary<K,V>
RWTValHashDictionaryIterator<K,V>
RWTValOrderedVector<T>
   RWTValSortedVector<T>
RWTValSlist<T>
RWTValSlistIterator<T>
RWTValSlistDictionary<K,V>
RWTValSlistDictionaryIterator<K,V>
RWTValVector<T>
RWTValVirtualArray<T>
RWvios
   RWvistream
      RWbistream
      RWpistream
      RWXDRistream
   RWvostream
      RWbostream
      RWpostream
      RWXDRostream
RWVirtualPageHeap
   RWBufferedPageHeap
      RWDiskPageHeap
RWWString
RWWSubString
RWWTokenizer
RWZone
   RWZoneSimple
```

Internationalization 5

Gone are the days when we could ignore our neighbors across the sea (or over the fence), writing software only for local consumption. Professional software development today demands not only awareness of the needs of users in other cultures, but accommodation of those needs. This accommodation is called *localization*; making software easily localized is called *internationalization*[1]

Internationalization actually involves many different activities, potentially as many as the ways in which cultures differ from one another. In practice, it usually means accommodating differences in alphabets, languages, currencies, numbers, and date- and time-keeping notations. Let us consider each of these in turn.

Accommodation of different alphabets begins with allowing them to be represented. A first step in this direction is making code "8-bit clean", which lets it tolerate extensions. Still, eight bits just isn't enough to represent all the character glyphs we use, even in English. Some extension beyond 8 bits is required, and in fact several are in use, falling into two families: 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.) This gives a compact encoding, but is inconvenient for indexing and substring operations.

1. "Internationalization" is a horrendous word, widely abbreviated "i18n"; 18 is the number of letters elided.

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.

Given any of these representations for strings, there remains much to do. 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 the Cyrillic names? How are wide-character strings represented on byte streams? These issues are being addressed, in standards bodies and in corporate labs, but the results are not very portable yet. Tools.h++ has no crystal ball, so we simply pass through the semantics your system vendor has provided.

Tools.h++ includes two efficient string types, `RWCString` and `RWWString`. `RWCString` represents 8-bit strings, with some support for multibyte strings. `RWWString` represents “wide strings”, 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, via both streams and 8-bit strings. The wide- and multibyte-character encodings used are those of the host system.

To accommodate a user’s choice of languages, a program must display titles, menu choices, and status messages in that language. Usually such texts are stored in a “message catalog” or “resource file”, separate from program code, so they may be easily edited or replaced. Tools.h++ issues no messages; though it does not yet offer much help in this area, it also imposes no policy.

While accounting principles are the same everywhere, the currencies used vary among cultures not only in unit value, but in notation. Indeed, even raw numbers are written differently in different places; in the U.S.A, we would say that 2.345 is less than 2,345; but in much of Europe the reverse is true. In many cases a program must be able to display values in the notations customary to both the vendor and the customer.

Scheduling, which appears in many kinds of software, involves time and calendar calculations. Local versions of the Gregorian calendar vary in their names for the months and the days of the week, and in the order in which the components of a date are written. Notations for the time of day vary as well. Time representations are complicated by time zone conventions, including Daylight Savings Time (DST) rules that vary wildly from place to place, and from year to year in some places.

The Standard C Library provides, with `<locale.h>`, some facilities to accommodate differences in currency, number, date, and time formats, but it is maddeningly incomplete. It offers no help for conversion from strings to these types, and is practically impossible to use if you must do conversions involving two or more locales. Common time zone facilities (such as those defined in POSIX.1) 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.

RWLocale and RWZone

Tools.h++ addresses these problems with the abstract classes `RWLocale` and `RWZone`. If you have used *`RWDate`* you have used `RWLocale` already, perhaps unknowingly. Every time you convert a date or time to or from a string, a default argument carries along a `RWLocale` reference. This is a reference to a global instance of the class `RWLocaleDefault` (derived from `RWLocale`) which was created at program startup. To use `RWLocale` explicitly, construct your own instance and pass it in place of the default. Similarly, when you manipulate times, a default `RWZone` reference is passed along, but you can substitute your own.

You can also install your own instance of `RWLocale` or `RWZone` as the global default. You can even install your `RWLocale` instance in a stream (this is called “imbuing the stream”) so that dates and times inserted on (or extracted from) that stream are formatted (or parsed) accordingly, without any special arguments.

Let us look at how all this works, with some examples. Here are the header files we will use:

```
#incluce <assert.h>
#include <rw/rstream.h>
#include <rw/cstring.h>
#include <rw/locale.h>
#include <rw/rwdate.h>
#include <rw/rwtime.h>
```

Begin by constructing a date, today's date:

```
RWDate today = RWDate::now();
```

We can display it using ordinary "C"-locale conventions, the usual way:

```
cout << today << endl;
```

But what if you are in some other locale? Perhaps you have set your environment variable `LANG` to "`fr`", because you want French formatting[1]. We would like the date to be displayed in your preferred local format. First, let's 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 straight-forward way to use this is to pass it directly to the `RWDate` member function `asString()`[2]:

```
cout << today.asString('x', here) << endl;
```

but there are more convenient ways. We 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 also want to format a date in German, but don't want that to be the default. Let us construct a German locale:

```
RWLocale& german = *new RWLocaleSnapshot(“de”);
```

Now we can format the same date for both local and German readers:

```
cout << today << endl
<< today.asString(‘x’, german) << endl;
```

Let us now suppose you want to read in a German date string. The straight-forward way, again, is to call everything explicitly:

1. Because of operating system limitations, you cannot change the locale under Solaris 1.*x*.

2. The function `asString()'s` first argument is a character, which may be any of the format options supported by the Standard C Library function `strftime()`.

```
RWCString str;
cout << "enter a date in German: " << flush;
str.readLine(cin);
today = RWDate(str, german);
if (today.isValid())
    cout << today << endl;
```

Sometimes you would prefer to use the extraction operator ">>". It must know to expect and parse a German-formatted date. We can 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, then 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, or who will read, the time string. The mishmash of Daylight Savings Time jurisdictions can make this annoyingly difficult. Numbers are somewhat messy to format because the insertion and extraction operators ("<<" and ">>") for them are already defined by

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.

<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.

Time

Let us consider the time zone problem. Our first observation is that there is no simple relationship between time zones and locales. All of Switzerland shares a single time zone, including DST rules, but has four official languages (French, German, Italian, and Romansch). Hawaii and New York, on the other hand, share a common language but occupy time zones five hours apart; or sometimes six hours apart, because Hawaii does not observe DST. Furthermore, time zone formulas have little to do with cultural formatting preferences. Thus, we use 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; we have implemented its interface 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 Daylight Savings Time in use. Whenever you convert an absolute time (as in the class RWTime) to or from a string, an instance of RWZone is involved. By default, the local time is assumed, but you can pass a reference to any RWZone instance.

It's time for some examples. Imagine you have scheduled a trip from New York to Paris. You will leave New York on December 20, 1993, at 11:00 PM, and return on March 30, 1994, leaving Paris at 5:00 AM, 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:
```
RWTime arriveParis  (leaveNewYork + long(7 * 3600));
RWTime arriveNewYork(leaveParis   + long(7 * 3600));
```

Let’s display the Paris arrival time and date in French, and the New York arrival time and date according to local convention:

```
RWLocaleSnapshot french("fr");
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;
```

This works even though your flight crosses several time zones and arrives on a different day than it departed. Furthermore, on the day of the return trip (in the following year), France has already begun observing Daylight Savings Time, 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 those DST rules Tools.h++ has built in. (Thus far, these are North America, Western Europe, and “noDST”.) 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 AM 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 `RWZoneSimple` documentation, and <rw/zone.h>, for details on what the numbers mean.) Then construct an `RWZone` object:
```
RWZoneSimple  ciudadSud( RWZone::Atlantic, &sudAmerica );
```

Now you can use `ciudadSud` identically as `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: they are probably used to it. If that just won't do, you can derive from RWZone and implement its interface for Britain alone. This is much easier than trying to make something general enough to handle all possibilities including Britain, and it's smaller and faster besides.

The remaining problem 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. Often, however, you can get the user in question to provide the necessary information. One manifestation 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. For example, for a user in Paris you could say:

```
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 operates on the Standard C Library type struct tm. RWDate and RWTime both provide conversions to this type. In some cases you may find using it directly is preferable to 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 as well) 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 might produce a string like "9:5". Here's another way:

```
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 again for the minute. This is an expensive operation. If you expect to work with the components of a time or date much, 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;
```

If you work with times before 1901 or after 2037, `RWTime` cannot be used, because it does not have the range needed. `struct tm` operations with `RWLocale` are not so restricted; you can use `RWLocale` to perform conversions for any time or date.

Numbers

Abstract class `RWLocale` provides 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`. This includes using appropriate digit group separators, decimal “point”, and currency notation. On conversion from strings it allows, and checks, the same digit group separators. Unfortunately, the standard iostream library provides definitions for number insertion and extraction operators which cannot be overridden, so stream operations are clumsier than we might like.

Instead, we 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) &&
     french.stringToNum(is, &i))  // verify conversion
   cout << f << “\t” << i << endl
        << fs << “\t” << is << endl;
```

The French use “,” for the decimal point, and “.” for the digit group separator, so this might display:

```
1.234567e+07987654
1.234.567,89987.654
```

Numbers with digit group separators are certainly easier to read.

Currency

Currency conversions are trickier, 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, in the U.S, to represent the balance “$10.00”, you might say

```
double sawbuck = 1000.;
```

This representation has the advantages of wide range, exactness, and portability. By wide range, we mean that it can *exactly* represent values from $0.00 up to (and beyond) $10,000,000,000,000.00. This is larger than any likely budget. By exactness, we mean 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 for instance "`price = 9.99;`".

By portability, we mean simply that `double` is a standard type, unlike common 64-bit integer or BCD representations. Of course, financial calculations may still be performed on such other representations, but because it is always possible to convert between them and `double`, this supports everyone. In the future `RWLocale` may directly support some other common representations as well.

Let us consider some 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 U.S. locale, this displays:

```
1000.00000  10.00  $10.00  USD 10.00
```

Wrap up

We have covered lots of territory—alphabets, languages, dates, times, time zones, numbers, money—and yet have only scratched the surface of what can be done by combining these facilities. Internationalization is a brave new world for software engineering, but with the proper tools it can be more exciting than distressing.

Strings 6

Manipulating strings is one of the most common and error prone tasks that a programmer does. It's a perfect opportunity for C++ to show its advantages.

Class `RWCString` has many powerful string processing features that are just as efficient as C, but far less prone to errors. For example, the class automatically takes care of memory management. It's just about impossible to delete something twice or not delete it at all.

Class RWWString offers support for *wide character* strings. These are strings of type `wchar_t` which, in general, may consist of more than one byte. The interface of `RWWString` is extremely similar to `RWCString`, allowing them to be interchanged easily.

Example

Here is a short example that exercises the `RWCString` class:

Code Example 6-1

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

main()
{
```

Code Example 6-1 (Continued)

```
  RWCString a;

  RWRegexp re("V[0-9]\\.[0-9]+");
  while( a.readLine(cin) ){
    a(re) = "V4.0";
    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...
```

This example reads lines from standard input and searches them for a pattern matching the regular expression "`V[0–9]\.[0–9]+`" (the extra backslash in the program is to escape the special character '\'. This expression matches "version numbers" between V0 and V9: V1.2, V1.22, but not V12.3. If a match is found, then the pattern is replaced with the string "`V4.0`." The magic here is in the expression;

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

The function call operator (*i.e.*, `RWCString::operator()`) has been overloaded to take an argument of type `RWRegexp`—the regular expression. It returns a "substring" that delimits the regular expression, or a null substring if a matching expression could not be found. The substring assignment operator is then called and replaces the delimited string with the contents of the right hand side, or does nothing if this is the null substring.

Here is another example that reads in two `RWCStrings`, concatenates them, converts to upper case, and then sends the results to `cout`:

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

Class `RWCString` has member functions to read, compare, store, restore, concatenate, prepend, and append `RWCStrings` and `char*`'s. Operators allow access to individual characters, with or without bounds checking. The details of the `RWCString` class capabilities are summarized in the *Class Reference, Part 2*.

Collation

The various comparison operators involving `RWCString` use case sensitive lexicographic comparisons:

```
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&);
```

If you wish to make case insensitive comparisons, then you use should use member function:

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

which returns an integer -1, 0, or 1, depending on whether `str` is lexicographically less than, equal to, or greater than self, respectively. The type `caseCompare` is an enum with values

```
exact Case sensitive
ignoreCaseCase insensitive
```

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

For locale-specific string collations, use member function

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

which is an encapsulation of the Standard C library function `strcoll()`. This function will return 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&);
```

and then use `compareTo()` or one of the logical operators (==, !=, etc.) on the results.

Substrings

A separate `RWCSubString` class supports substring extraction and modification. There are no public constructors, so no variables of type `RWCSubStrings` in your application are constructed indirectly by various member functions of `RWCString`, and then destroyed at the first opportunity. The resulting substring can be used in a variety of situations.

For example, a substring can be created by an overloaded version of `operator()()` This can then be used 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`.

`RWSubStrings` may also be used as lvalues in an assignment, either 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 – Assignment 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. Here is an example. The code fragment:

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

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

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

which will result in `i` being set to “14”. To make a case-insensitive comparison use:

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

which will also result in `i` being set to “0”.

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

Regular expressions

The Tools.h++ Class Library supports regular expression searches. See *Part 2: Class Reference*, under `RWCRegexp`, for details of the regular expression syntax. A regular expression can be used to return a substring. Here's an example that might be used to match all mail headers identifying the subject:

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

main()
{
  RWCString a("Subject: Roses");

  // Construct a Regular Expression to match Subject: Roses:
  RWCRegexp re("Subject: .*");
  cout << a(re) << endl;

  return 0;
}
```

Program output:

```
Subject: Roses
```

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.

String I/O

Class `RWCString` offers a rich I/O facility to and from both iostreams and Tools.h++ virtual streams.

iostreams

The standard l- and r-shift operators have been overloaded to work with iostreams and `RWCStrings`:

```
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 C++ standard library that comes with your compiler. That is, the l-shift (<<) operator writes a null-terminated string to the given output stream. The r-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. Function `readline()` allows strings separated by newlines. It has an optional parameter controlling whether whitespace is skipped before storing characters. Here’s an example showing the difference:

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

main()
{
    RWCString line;

    {
    int count = 0;
    ifstream istr(“testfile”);

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

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

    return 0;
}
```

Program input:

```
line 1
line 5
```

Program output:

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

Virtual streams

String operators to and from virtual streams are also supported:

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

This allows a string to be saved and restored without knowing the formatting that is to be used. See Chapter 17, “Persistence” for details on virtual streams.

Tokenizer

Class `RWCTokenizer` can be used to break a string up into tokens, separated by an arbitrary "white space". See *Part 2: Class Reference*, under `RWCTokenizer`, for additional details. 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 any other iterator. 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.

Multibyte strings

Class RWCString provides limited support for multibyte strings. Because a multibyte character can consist of two more 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 may contain multibyte characters, then 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:

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

The string in Sun is the day of the week Sunday in Kanji, using the EUC (Extended Unix Code) multibyte code set. With EUC, a single character may be one to four 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 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 can be summarized by the following program:

```
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"
```

Note that two different constructors were used above. The constructor in lines 1 and 2 take 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 in the usual manner: 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()`.

Note 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 `RWCStrings` containing multibyte strings.

Wide character strings

Note – Wide character strings are supported under Solaris 2.*x* only.

Class `RWWString` is extremely similar to `RWCString`, except that 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 that builds on the previous section:

```
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
```

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 above.

This is a relatively expensive conversion and the `multiByte` argument ensures that it is not done inadvertently. The conversion from a wide character string back to a multibyte string is done using function `toMultiByte()`. Again, this is a relatively expensive operation.

If you know that your `RWCString` consists entirely of Ascii characters then the cost of the conversion in both direction can be greatly reduced. This is because the conversion involves a simple manipulation of high-order bits:

```
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
```

Note how member functions `RWCString::isAscii()` and `RWWString::isAscii()` were used in ensure that the strings, in fact, consisted entirely of Ascii characters. The `RWWString` constructor

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

was used to convert from Ascii to wide characters. The parameter `ascii_` is an enum with a single possible value: `ascii`, as shown in the example above.

Member function `RWWString::toAscii()` was used to convert back.

Using Class RWDate 7

Class `RWDate` represents a date, stored as a Julian day number. It serves as a compact representation for calendar calculations, shields you from details such as leap years, and performs conversions to and from conventional calendar formats.

The algorithm to convert a Gregorian calendar date (for example January 10, 1990) to a Julian day number is given in: Algorithm 199 from Communications of the ACM, Volume 6, No. 8, Aug. 1963, p. 444.

The Gregorian calendar was introduced by Pope Gregory XIII in 1582, and was adopted by England on September 14, 1752. Class `RWDate` will not provide valid dates before 1582.

Example

This example prints out the date 6 January 1990 and then calculates and prints the date of the previous Sunday, using the global locale:

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

main()
{
  RWDate dd(6, "January", 1990);

  cout << dd << ", a " << dd.weekDayName() << endl;

  RWDate prev = dd.previous("Sunday");

  cout << "The previous Sunday is: " << prev << endl;
  return 0;
}
```

Program output:

```
01/06/90, a Saturday
The previous Sunday is: 12/31/89
```

Constructors

An `RWDate` may be constructed in several ways. For example:

1. Construct a `RWDate` with the current date[1]:

   ```
   RWDate d;
   ```

2. Construct a `RWDate` for a given day of the year (1–365) and a given year (*e.g.*, 1989 or 89)

   ```
   RWDate d1(24, 1990);// 1/24/1990
   RWDate d2(24, 90);// 1/24/1990
   ```

1. Because the default constructor for `RWDate` fills in todays'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.

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

   ```
   RWDate d(10, 3, 90);// 3/10/1990
   ```

4. Construct a `RWDate` from a `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" will be used. This locale uses US conventions and names:

```
RWDate d1(10, "June", 90);// 6/10/1990
RWDate d2(10, "JUN", 90);// 6/10/1990
```

Suppose you wished to use French month names and your system supported a French locale. Here's how you might do it:

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

main()
{
  RWLocaleSnapshot french("fr");  // 1

  RWDate d(10, "Juin", 90, french);// OK// 2
  assert(RWDate(10, "Juin", 90).isValid()         == FALSE);//
3
  assert(RWDate(10, "June", 90, french).isValid() == FALSE);//
4

  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:

1. A “snapshot” is taken of locale "`fr`". This assumes that your system supports this locale.

2. A date is constructed using the constructor

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

Note – 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 (as known in English) of June 10, 1990.

3. Here we attempt to construct the same date using the default locale, an instance of class `RWLocaleDefault`. This locale recognizes US formatting conventions only. Hence, the date 10 Juin 1990 is meaningless.

4. For the same reason, constructing a date using US names, but with a French locale, also fails.

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

```
June 10, 1990
```

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

```
June 10, 1990
```

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

```
10 Juin 1990
```

Using Class RWTime 8

Class `RWTime` represents a time, stored as the number of seconds since 1 January 1901 UTC. While UTC is a widely accepted time standard, it is not the usual time reference that most people use in their day-to-day lives. We tell time with a "local" time which may or may not observe daylight savings time (DST). In addition, DST may or may not actually be in effect.

Hence, when we create an `RWTime` object, we are unlikely to do so with UTC. More likely, the time we give it will be with respect to some other time zone. For `RWTime` to do the job properly, it must know which time zone you mean. By default, it uses a global "local" time, set by `RWZone::local()`. The same issue arises when you get the time back out to be printed: in which time zone do you want it to be printed? Again, the default is the global local time.

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 daylight savings time 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 if daylight savings time is not observed in your area. In places not governed by U.S. daylight savings time rules, you may need to re-initialize the local time zone—see "RWZone" on page 420 in Chapter 22, "Class Reference."

Constructors

A `RWTime` may be constructed in several ways:

1. Construct a `RWTime` with the local time:

```
RWTime t;
```

2. Construct a `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 a `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 a `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 hour, minute or second, or fill a struct tm for any time zone. A complete list of member functions is included in *Class Reference, Part 2*

For example, a code fragment to output the hour in local and Universal (GMT) zones, and then the complete local time and date, is:

```
RWTime t;
cout << t.hour() << endl;
cout << t.hour(RWZone::utc()) << endl;
cout << t.asString(‘c’) << endl;
```

Here is how you find out when daylight savings time starts for the current year and local time zone:

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

Virtual Streams 9

The “iostream” facility that comes with every C++ compiler is a resource that is familiar to every C++ programmer. Among its advantages are type safe insertion and extraction into and out of streams and extensibility to new types. Furthermore, the source and sink of the bytes of the stream are set by another class (`streambuf`) and are completely transparent to the user of the streams.

But it suffers from a number of limitations. The biggest is its limited formatting abilities: if you insert, say, a double into an `ostream`, you have very little say over what format is to be used. For example, there is no type-safe way to insert it as binary.

Another limitation of iostreams is their assumption that all byte sources and sinks can fit into the `streambuf` model. For many protocols (for example, XDR), the format is intrinsically wedded to the byte stream and cannot be separated.

The Tools.h++ “virtual streams” overcomes these limitations by offering an idealized model of a stream. No assumptions are made about formatting, or stream models. At the bottom of the virtual streams class hierarchy is class `RWvios`. This is an abstract base class with an interface very similar to the standard library class `ios`:

```
class RWvios
{
public:
   virtual inteof()= 0;
   virtual intfail()= 0;
   virtual intbad()= 0;
   virtual intgood()= 0;
   virtual intrdstate()= 0;
   virtual intclear(int v = 0)= 0;
};
```

Specializing versions of `RWvios` will supply definitions for these functions.

Inheriting from `RWvios` are 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 of 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 intget()= 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 builtins 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 builtins are “inserted” into a virtual output stream, “extracted” from a virtual input stream, *without any regard for formatting*. That is, 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 not only use, but also write, streaming operators without knowing anything about the final output medium or formatting that is 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, the same streaming operators can be used.

Specializing virtual streams

The Tools.h++ classes come with three types of classes that specialize `RWvistream` and `RWvostream`. The first uses a “portable ASCII” formatting, the second a binary formatting, and the third an XDR (eXternal Data Representation; a Sun Microsytems standard) formatting:

Table 9-1

	Input class	**Output class**
Abstract base class	`RWvistream`	`RWvostream`
Portable ASCII	`RWpistream`	`RWpostream`
Binary	`RWbistream`	`RWbostream`
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 new operating system. The “binary” versions do not reformat inserted items and, instead, store them in their native format. The XDR streams 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 *Part II: Class Reference.*

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>

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);// 1
    RWbostream bostr(f);// 3
    save(a, bostr);
  } // 4

  ifstream f("junk.dat", ios::in);// 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(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. Some additional comments on particular lines:

1 On this line, a file output stream `f` is created for the file "`junk.dat`".

3 On this line, a `RWbostream` is created from `f`.

4 Because this clause in 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 a `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.

This program could be simplified 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.

Recap

We have seen how an object can be stored and recovered from streams without regard to the final destination of the bytes of that stream. They could reside in memory, or on disk. We have also seen how we need not be concerned with the final formatting of the stream. It could be in ASCII or binary.

It is also quite possible to 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, there is no need to modify any of the code of the client classes—you just use your stream class as an argument to

```
RWvostream& operator<<(RWvostream&, constClassName&);
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 encapsulates ANSI-C style file I/O. Although more limited in its abilities than stream I/O, this form of storage and retrieval is faster to and from disk because the virtual dispatching machinery is not needed.

Using Class RWFile 10

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

The constructor for class `RWFile` has prototype:

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

This constructor will open a binary file called `filename` with mode `mode` (as defined by the Standard C function `fopen()`; for example "`r+`"). If `mode` is zero (the default) then an existing file will be opened for update (mode “r+” for Unix, “rb+” for DOS), while non existing files will be created (modes “w+” and “wb+”). The destructor for this class closes the file.

There are member functions for flushing the file, and for testing whether the file is empty, has had an error, or is at the end-of-file

Example

Class RWFile 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 *Part 2: Class Reference.*

For example, to create a RWFile with filename "test.dat", read an int (if the file is not empty), increment it, and write 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 11

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 longRWspace;
```

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 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.

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 this `RWFileManager`, or return `RWNIL`[1] if no space has been allocated, implying that this is a "new file".

The statement

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

uses `RWFileManager F` 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 then use `Write()`. It is an error to read or write to an unallocated location in the file. It is also 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 a `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.

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

The following example shows the use of class `RWFileManager` to construct a linked-list of ints on disk: The source code is included in the `toolexam` subdirectory as `example7.cc` and `example8.cc`.

Code Example 11-1

```
#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);
```

Code Example 11-1

```
    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, re-write 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 we read it? Here is a program that reads the list and prints the stored integer field.

```
#include <rw/filemgr.h>
#include <w/rstream.h>
struct DiskNode {
    int data;
    RWoffsetnextNode;
};

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
```

Here's 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 12

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 `chars`. The key length is set by the constructor. The ordering in the B-Tree is determined by comparing keys with an external function. You can change this function.

The type of the values is set by a typedef. This is:

```
typedef long RWstoredValue;
```

Typically, the values represent an offset to a location in a file where an object is stored. Hence, given a key, one can find where an object is stored and retrieve it. However, as far as class `RWBTreeOnDisk` is concerned, the value has no special meaning—it is up to you to interpret it.

This class uses class `RWFileManager` to manage the allocation and deallocation of space for the nodes of the B-Tree. The same `RWFileManager` can also be used to manage the space for the objects themselves if the B-Tree and data are to be in the same file. Alternatively, a different `RWFileManager`, managing a different file, could be used if it is desirable to store the B-Tree and data in separate files.

The member functions associated with class `RWBTreeOnDisk` are identical to those of class `RWBTree`, which manages a B-Tree in memory, except that keys are arrays of `chars` rather than `RWCollectable*`'s. There are member functions to add a key-value pair, find a key, remove 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

A RWBTreeOnDisk is always constructed from a 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 a RWFileManager for a new file called "filename.dat" and then constructs a 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 RWDates, representing birthdays, are stored. Given a name, a birthdate can be retrieved 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 BTree. The default constructor is used, resulting in a key length of 16 characters (the default).
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. Member function `binaryStoreSize()` is a member function that most Tools.h++ classes have. It returns the number of bytes necessary to store an object in a binary file.

5. Seek to the location where the `RWDate` will be stored.
6. Store the date at that location. Most Tools.h++ classes have an overloaded version of the streaming (<< and >>) operators.
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;
}
```

Program description:

1. The program accepts names until encountering an EOF.

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

3. Check to see whether the name was found.

4. If the name was valid, use the value to seek to the spot where the associated birthday is stored.

5. Read the birthdate from the file.

6. Print it out.

With a little effort it is easy to 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.rootLocation();
    }
    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,
        autoCreate,// Will read old tree
        16,        // Key length
        FALSE,       // Do not ignore nulls
        rootArray[itree] // Location of root
        );
    }
    return 0;
}
```

Introduction to Collection Classes 13

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

- A set of template-based collection classes;
- A set of "generic" collection classes, modeled after Stroustrup (1986), Chapter 7.3.5;
- A set of Smalltalk-like collection classes.

Despite their very different implementations, their functionality, as well as their user-interfaces (member function names, *etc.*), is very similar.

The objective of this chapter is to discuss a few basic concepts about objects and collection classes in general and to expose some of the jargon that you're likely to encounter both here and in the literature.

Concepts

The general objective of collection classes is to store and retrieve objects. What makes them different from one another is how they store the object and how they do a lookup.

Storage methods

There are two different ways to store an object in a collection: either store the object itself (*value-based* collections) or store a pointer or reference to the object (*reference-based* collections). The difference can have important consequences.

Value-based collections are simpler to understand and manipulate. You create, say, a linked list of integers or doubles. Or a hash table of shorts. The stored type can be more complicated: for example, `RWCStrings`. The important point is that, even though the stored type may contain pointers to other objects (as does `RWCString`), they *act as if* they are values. When an object is inserted into a value-based collection, a *copy* is made. An analogy is C's pass-by-value semantics in function calls.

In a reference-based collection, you store and retrieve *pointers* to other objects. For example, you could create a linked list of pointers to ints or doubles. Or a hash table of pointers to `RWCStrings`. This type of collection can be very efficient because pointers are small and inexpensive to manipulate, but you must be aware of the lifetime of the pointed-to objects, lest you create two pointers to the same object and then prematurely delete the object, leaving the second pointer pointing into nonsense.

In general, with a reference-based collection you are responsible for the creation, maintenance and destruction of the actual objects themselves, although the Tools.h++ classes have a few member function to help you do this.

Having said this, reference-based collections shouldn't scare you. In the vast majority of the cases, the "ownership" of the contained objects is perfectly obvious. Furthermore, reference-based collections enjoy certain performance and size advantages because the size of all pointers is the same, allowing a large degree of code reuse. You may also *want* to simply point to an object rather than contain it (a set of selected objects in a dialog list is an example that comes to mind). Finally, for some kinds of heterogeneous collections, they may be the only viable approach.

It is important to note that there is a transition between the two types: one person's value is another person's pointer! You can always use a value-based collection class to create a reference-based collection by using addresses (*i.e.*, pointers) as the inserted type. But, the resulting programming interface may well be unpleasant and clumsy.

Shallow versus deep copies

Reference-based collections

What happens when you make a copy of a reference-based collection? It turns out that this is a more general issue: it doesn't just affect collection classes, but *any* class that references another object. Hence, if the type of object that a value-based collection is collecting is an *address*, then this issue will arise there too.

There are two general approaches: *shallow copying* and *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` will have the same members as the old set. The new `Set` will share objects with the old `Set` (through pointers). Shallow copies are sometimes referred to as using *reference semantics.*
2. A *deep copy* of an object will make a copy with entirely new instance variables. The new object will not share objects with the old object. For example, a deep copy of a Set would not only make a new set, but the items inserted into it would also be copies of the old items. In a true deep copy, this copying is done recursively. Deep copies are sometimes referred to as using *value semantics.*

Note – The copy constructors of all reference-based Tools.h++ collection classes make *shallow copies.*

Some reference-based collection classes have a `copy()` member function that will return a new object with entirely new instance variables, although this copying is not done recursively (that is, the new instance variables are shallow copies of the old instance variables).

Here are graphical examples. Imagine a `Bag` (an unordered collection of objects with duplicates allowed) that looks like Figure 13-1 on page 86 before a copy :

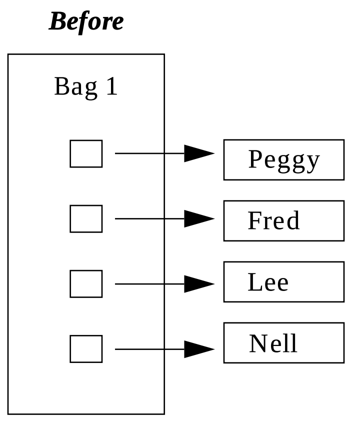

Figure 13-1 `Bag` before a copy

A shallow and deep copy of this collection would look like (respectively):

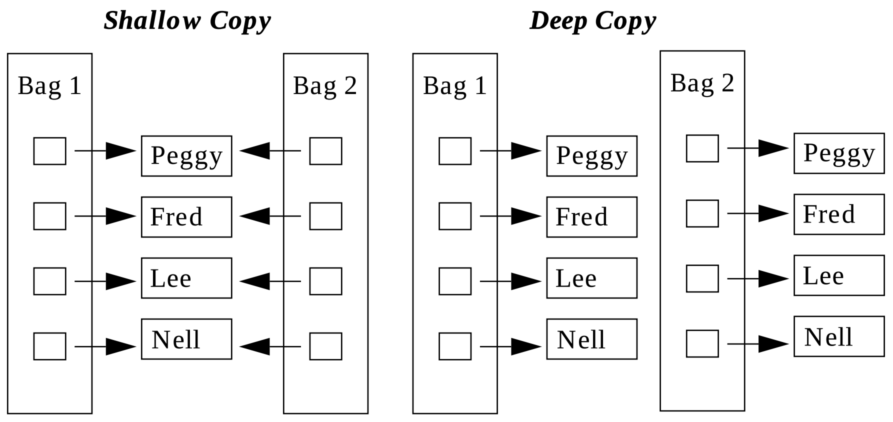

Figure 13-2 Shallow and deep copy of collection

Note – The deep copy not only made a copy of the bag itself, but also, recursively, any objects within it.

Although shallow copies can be useful (and fast, because less copying is done), one 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.

The issue of shallow versus deep copies can also arise when an object is written to disk. If an object includes two or more pointers or references to the same object, then when it is restored it is important that this *morphology* be preserved. Classes which inherit from `RWCollectable` inherit algorithms which guarantee to preserve an object's morphology. More on this later.

Value-based collections

Now contrast this situation with a value-based collection. For the sake of definiteness, consider the class `RWTValOrderedVector<RWCString>`, that is, an ordered vector template, instantiated for `RWCString`'s. In this case, each string is embedded within the collection. When a copy of the collection is made, not only is the collection itself copied, but also the objects in it. This results in distinct new copies of the collected objects:

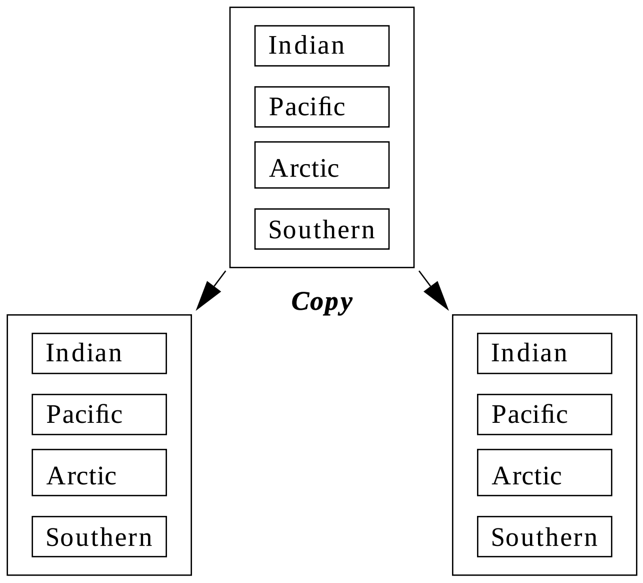

Figure 13-3 Coping a value-based collection

13

Retrieving objects

Properties of objects

Every object that you create has three properties associated with it:

1. Its "type" (*e.g.*, a `RWCString` or a "double"). In C++, an object's type is set at creation. It cannot change.

2. Its "state" (*i.e.*, the value of the string). The values of all the instance variables or *attributes* of an object determine its state. These can change.

3. Its "identity" (*i.e.*, identifies an object uniquely and for all time). In C++, an object's identity is determined by its 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] is necessary to disambiguate which object you mean within an inheritance hierarchy.

Note – Different languages use different methods for establishing an object's identity. For example, an object-oriented data base could use an "object ID" to identify a particular object. It is just a property of C++ that an object's address is used.

Retrieval methods

How an object is "found" within a collection class depends on how you use these three properties.

Note – A key point is that there are two general methods for "finding" an object and you will have to keep in mind which you mean. Some collection classes can support either method, some can support only one.

---

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

1. Find an object with a particular state. An example is testing two strings for the same value. In the literature, this is frequently referred to as two objects testing "isEqual", having "equality", "compares equal", having the same "value", or testing true for the "=" operator. Here, we will refer to the two objects as testing equal ("isEqual"). In general, it is necessary to have some knowledge of the two object's type (or subtype, in the case of inheritance) in order to find the appropriate instance variables to test for equality.[1]

2. Finding a particular object (that is, one with the same *identity*). In the literature, this is referred to as two objects testing "isSame", or having the same "identity", or testing true for the "==" operator. We will 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, whether two objects are, in fact, the same object) you must test to see if they have the same address.

Note – In C++, because of multiple inheritance, the address of a base class and its associated derived class may not be the same. Because of this, if you compare two pointers (*i.e.*, two addresses) to test for identity, the types of the two pointers should be the same.

Smalltalk uses the operator "=" to test for equality, the operator "==" to test for identity. However, in the C++ world, operator"=" has become firmly attached to meaning assignment, operator "==" to generally meaning equality of values. This is the approach we have taken.

Note – In the Tools.h++ Classes, the operator "==", when applied to two classes, generally means test for equality of values ("isEqual"). Of course, when applied to two pointers, it means test for identity.

Whether to test for equality or identity will depend on the context of your problem. Here are some examples that might help.

1. The Tools.h++ collection classes allow a generalized test of equality. It is up to you to define what it means for two objects to “be equal.” That is, 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, both are mammals.

Suppose you were maintaining a mailing list. Given a person's name, you want to find his or her address. In this case, you would want to search for a name that *is equal* to the name at hand. A `Dictionary` would be appropriate. The *key* to the `Dictionary` would be the name, the *value* would be the address.

Suppose you are writing a hypertext application. You need to know in which document a particular graphic occurs. This might be done by keeping a `Dictionary` of graphics and their corresponding document. In this case however, you would want an `IdentityDictionary` because you need to know in which document a *particular* graphic occurs. The graphic acts as the key, the document as the value.

Suppose you were maintaining a disk cache. You might want to know whether a particular object is resident in memory. In this case, an `IdentitySet` might be appropriate. Given an object, you can check to see whether it exists in memory.

Iterators

Many of the collection classes have an associated iterator.

The advantage of an iterator is that it maintains an internal state, allowing two important benefits: more than one iterator can be constructed from the same collection, and all of the items need not be visited in a single sweep.

Iterators are always constructed from the collection itself. For example:

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

Caution – Immediately after construction (or after calling `reset()`), the state of the iterator is undefined. You *must* either advance it or position it before using its current state (*i.e.*, position).

The rule is "advance and then return." All Tools.h++ iterators work this way.[1]

If you change the collection by adding or deleting objects while an iterator is active, the state of the iterator also becomes undefined—using it could bring unpredictable results. The member function `reset()` will restart the iterator, as if it had just been constructed.

At any given moment the iterator "marks" an object in the collection. You can think of it as the "current" object. There are various methods for moving this "mark."

Most of the time you will probably be using member function `operator()`. It is designed to *always* advance to the next object, then either return `TRUE` or a pointer to the next object, depending on whether the associated collection is value-based or reference-based, respectively. It always returns `FALSE` *(i.e.*, zero) when the end of the collection has been reached. Hence, a simple, canonical form for using an iterator is:

```
RWSlistCollectable list;
.
.
.
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 is value-based or reference-based, respectively.

1. This is actually patterned after Stroustrup (1986, Section 7.3.2).

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

Templates 14

Introduction

Ever since Version 2.0, Tools.h++ has traditionally offered two types of collection classes: the so-called "generic" collection classes and the Smalltalk-like collection classes. Each had its advantages and disadvantages. The Smalltalk-like classes offer a pleasant programming interface, code reuse, and a consistent interface. However, they actually collect pointers to a common base class (`RWCollectable`), requiring that all collected objects inherit from this class—the actual type of the class is unknown at runtime. This requires the programmer to rely on some other kind of information, either the `isA()` function or some logical inference, to decide whether it is safe to "downcast" the pointer to an `RWCollectable` to a pointer to the derived class. A recent language extension[1] holds the promise that such downcasting will at least be standardized by the language. However, the programmer will always have to remember to perform the check. The check may also fail: an object's inheritance hierarchy may make the type cast impossible and the prudent programmer must be prepared for this.

1. See, for example, Josée Lajoie's article "*The new language extensions*" in the July-August 1993 issue of *The C++ Report*.

Enter templates

Templates, or parameterized types, offer an elegant solution. Over the next few years they promise to revolutionize the way we approach C++ programming. Indeed, 5 years from now the polymorphic collection approach may be just a footnote in the history of C++ design.

Templates are extremely similar to the "generic.h" approach, described in the following section, which relies on preprocessor macros to define a family of classes parameterized on a type. However, a macro does a lexical substitution *in situ*, that is at the site of invocation, making them extremely hard to debug (a single line may be expanded into thousands of tokens!). They are also nasty to write, requiring as they do that all newlines be escaped with a backslash.

Templates are logically a class declaration parameterized on a type. They are a prescription for how a particular type of collection should behave. For example, a Vector template would describe such things as how to index an element, how long it is, how to resize, *etc*. The actual *type* of the elements is independent of these larger, more general issues.

Now if you procede to request a vector of a particular type, say a `Vector<double>`, that is, a vector of doubles, then the compiler goes back to the template declaration and fills it in for the type double. The effect is as if you had hand written a class "`VectorOfDoubles`". But, of course, you didn't. Instead, the compiler automatically generated the logical equivalent of such a class. The result is extreme *source* code reuse.

What's the catch?

Which brings us to the one disadvantage of templates. It is the *source* code that is being reused, not the *object* code. A declaration for `Vector<double>` gets compiled separately from `Vector<int>`. Unless some provisions have been made by the class designer, the two declarations will generate totally independent object code, with the potential for bloat of the executable.

Indeed, to some extent, this is unavoidable. Still, the careful class designer will recognize points of commonality that do not depend on the actual type and factor them out. These are put in a separate "type-independent" class which can be compiled once, resulting in more compact code. A trivial example, one that builds on our discussion of vectors, might be the vector length. We might want to declare a `BaseVector` class that holds the length of any vector. We would then derive the template-based vectors Vector from this. Of course, the code required to return the length of a vector is trivial, and so it turns out that in practice it isn't worth doing this.

Factoring out commonality

For a more substantial example, take a look at the intrusive linked-list class `RWTIsvSlist<T>`. This is a linked list of types `T` which are required to inherit from class `RWIsvSlink`. This class contains a "next" field, consisting of a pointer to the next link. A few moments reflection will convince you that we don't really need to know the derived type of the link to actually walk the list. We need only know that it inherits from `RWIsvSlink`. Furthermore, given an existing link, we don't even need to know its type to add or remove it from the list!

Hence, all this code can be factored out, compiled once, and forgotten.

Naming scheme

All of the template class names start with "RWT", 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 `T`.

Types of templates

Intrusive lists

For a collection of type 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.

Value-based collections

Value-based collections *copy* the object in and out of the collection. A very familiar example of a value-based "collection" is the C array:

```
int v[100];/* A 100 element array of ints */
int i = 7;
v[2] = i;
```

The statement "`v[2] = i`" *copies* the value of `i` into the array at index 2. What resides within the array is a distinct copy, separate from the original object `i`.

Value-based collections can contain more complicated objects too. Here's an example of a vector of `RWCStrings`:

```
/* A 100 element array of RWCStrings: */
RWTValVector<RWCString> v(100);
RWCString s("A string");
v[2] = s;
```

Just as with the array of ints, the statement "`v[2] = s`" *copies* the value of `s` into the vector at index 2. The object that lies within the vector is distinct and separate from the original object `s`.

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

Pointer-based collections

The final type of collection, pointer-based, is similar to the Smalltalk-like collection classes. The data that gets inserted into the collection is not the object itself, but rather its *address*. That is, a pointer to the data is inserted. Now the collection class refers to the original object:

```
/* A 100 element array of pointers to RWCStrings: */
RWTPtrVector<RWCString> v(100);
RWCString s("A string");
v[2] = &s;
```

Both the object `s` and the array element `v[2]` refer to the *same* object. Naturally, this approach requires some care: should the string be deleted the array element will be left pointing into nonsense. You must take care that you are aware of who is responsible for the allocation and deallocation of objects collected this way.

Neverthless, this type of collection can be appropriate for many situations. You may need to have the same group of objects referred to in many ways, requiring that each collection *point* to the target objects, rather than wholly contain them. For example, a collection may refer to a set of "picked" objects that your graphical program has obtained from a user. You need to know which ones must be highlighted.

An example

It's time to look at an example. Let's start with a simple one: a vector of doubles.

Code Example 14-1

```
#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
```

Code Example 14-1 (Continued)

```
    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.

Note – For all Tools.h++ classes, `operator[](int)` always performs a bounds check on its argument.
Some classes also support `operator()(int)`. Bounds checking is generally not performed for this operator unless you define `RWBOUNDS_CHECK` before including the header file.

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

Note – If a vector is lengthened using `reshape(int)`, then the values of the new elements are undefined.
If a vector is lengthened using `resize(int)`, then the new values are initialized to something sensible, generally zero or blanks, depending on the vector type.

All Tools.h++ vectors work this way.

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

A more complicated example

Here's another example, this one involving a hashing dictionary.

```
#include <rw/tvhdict.h>
#include <rw/cstring.h>
#include <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> map(hashString);// 6

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

    RWTValHashDictionaryIterator<RWCString,Count> next(map);// 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
```

The problem is to read an input file, break it up into tokens separated by whitespace, count the number of occurences 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 zeroes 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.
6. Here the dictionary is constructed. Given a key, the dictionary will 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.
7. Tokens are read from the input stream into a `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", otherwise zero. 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 guide and the `ios` entry in the class reference guide that comes with your compiler for more details.

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. For all template iterators, the prefix increment operator advances the iterator *then* tests whether it has gone past the end of the collection.

12. The key and value at the position of the iterator are printed.

“Generic” Collection Classes 15

This chapter describes the second kind of collection classes included in the *Tools.h++ Class Library:* generic collection classes. They are so-called because they use the macros defined in `<generic.h>`, an early approximation to parameterized types, first described in Stroustrup (1986, p. 209). While they can be more unwieldy than true templates[1], they do offer the advantage of being portable to any C++ compiler, even older compilers.

Most of the "generic" collection classes use reference-based semantics (that is, they store and retrieve pointers to other objects). Like the other reference-based Tools.h++ collection classes, you are responsible for the allocation and deallocation of the objects themselves. However, the three vector-based collection classes, `RWGVector`(*val*), `RWGOrderedVector`(*val*), and `RWGSortedVector`(*val*) use value-based semantics and 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 very easy to use, they are just difficult to write. Because they are preprocessor macros, they must be “all on one line,” making them very difficult to debug.

Example

Here is an example of using a `RWGStack` (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
    return 0;
}
```

Program output:

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

Each program line 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, it serves as a *restricted interface* to the underlying implementation, which is a singly-linked list (class `RWSlist`). It is restricted because only certain member functions of `RWSlist` can be used (those appropriate to stacks) and because the items to be inserted into the stack must be of type *type*.

2. `<rw/rstream.h>` is a special Tools.h++ supplied header file that includes `<iostream.h>` with the appropriate suffix, depending 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 *Class*`declare` with argument *type*. Hence, in this case, the macro `RWGStackdeclare` will be called (which was defined in `<rw/gstack.h>`) with argument `int`.

Note – To summarize, the result of calling the declare(`RWGStack, int`) macro is that a new class has been created, especially for your program. It will be a stack of pointers to ints. Its name, for all practical purposes, is `RWGStack(int)`.

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

5–8. Four `ints` were 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, a pointer to the int "1" will be 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.

Declaring generic collection classes

All of the Tools.h++ generic collection classes are declared using the `declare` macro, defined in the header file `<generic.h>`, in a manner similar to the example above. 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). This 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 Tools.h++ generic classes are all declared using the item name itself. This is true for both the reference- 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, discussed in the following two sub-headings.

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. The user can choose to compare addresses (test for two objects being "identical"), or look for certain values within the object (test for "isEqual"). The first variable `ty` will point to a member of the collection and can be thought of as a "candidate". The second variable a is available for your use to be tested against ty for a match. It can be thought of as "client data", used to make the decision of whether there is a match.

Here is an example that expands on the Example in “Example” on page 104. The problem is to push some values onto a stack and then to see if a certain value exists on the stack (test for "isEqual").

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";
    return 0;
}
```

Program output:

```
Yup.
```

A description of each program line follows.

1. This is the tester function. The first argument is a pointer to the type of objects in the collection, `ints` in this case. The second argument points to an object that can be of any type. In this example, it also points to an `int`. 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 is then compared to the dereferenced first argument, which is also a `const int`. The net result is that this tester function considers a match to have occurred when the two `ints` have the same values (*i.e.*, they are *equal*).

Note – We could have chosen to have identified a *particular* int (*i.e.*, test for *identity*).

3–7. These lines are the same as in “Example” on page 104. A generic stack of (pointers to) ints 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, waiting for the tester function to signal a match. If it does, `contains()` returns `TRUE`, otherwise `FALSE`.

Note – 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).

The following is an example where they are not of the same type:

```
#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";
    return 0;
}
```

Here, 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 this 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).

Here is an example, using class `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);
    return 0;
}
```

Program output:

```
1
2
3
4
```

The items are appended at the tail of the list (see *Part 2: Class Reference*). 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.

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

Smalltalk-like Collection Classes 16

Introduction

The third general type of collection classes provided with the Tools.h++ Class Library is a set of "Smalltalk-80-like Collection Classes". In this approach, objects to be collected must inherit the abstract base class "`RWCollectable`", using either single or multiple inheritance. The principal advantage of this approach is that the programming-interface is much cleaner and the collection classes are far more powerful. The disadvantage is that the objects are slightly larger, the collection classes slightly slower, and (as we shall see) not as typesafe..

Many of these 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`. Table 16-1 summarizes.

Table 16-1 Smalltalk-like classes, iterators, and implementations.

Class	Iterator	"Smalltalk" typedef	Implemented as
`RWBag`	`RWBagIterator`	`Bag`	Dictionary of occurrences
`RWBinaryTree`	`RWBinaryTree-Iterator`	`SortedCollect-ion`	Binary tree
`RWBTree`			B-Tree in memory

Table 16-1 Smalltalk-like classes, iterators, and implementations. *(Continued)*

Class	Iterator	"Smalltalk" typedef	Implemented as
`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
`RWSlistCollectablesQueue`	*(n/a)*	`Queue`	Singly-linked list
`RWSlistCollectablesStack`	*(n/a)*	`Stack`	Singly-linked list
`RWSortedVector`	`RWSortedVector Iterator`		Vector of pointers, using insertion sort

Table 16-2 lists the class hierarchy of the Smalltalk-like collection classes.

Note – Some of these classes use multiple-inheritance: this hierarchy is shown relative to the `RWCollectable` base class.

Table 16-2 The class hierarchy of the Smalltalk-like collection classes

- 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 get us oriented, it is always good to look at an example. This code uses a `SortedCollection` to store and order a set of `RWCollectableStrings`. We will go through it line-by-line and explain what is happening:

```
#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
    return 0;
}
```

Program output:

```
Bill
George
Mary
Throkmorton
```

`SortedCollection` is actually a typedef for a `RWBinaryTree`. Objects inserted into it are stored in order according to relative values returned by the virtual function `compareTo()` (see “Virtual function compareTo()” on page 143 in Chapter 18, “Designing an RWCollectable Class”).

1. By defining the preprocessor macro `RW_STD_TYPEDEFS` we enable the set of Smalltalk-like typedefs. This enables us to use the generic name "`SortedCollection`" instead of `RWBinaryTree`, its true identity.

2. The second `#include` declares class `RWCollectableString` which is a derived class with classes `RWCString` and `RWCollectable` as base classes. Multiple inheritance was used to create this class. Most of its functionality is inherited from class `RWCString`. Its ability to be "collected" was inherited from class `RWCollectable`.

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

4–7. Four `RWCollectableStrings` were created off the heap and inserted into the collection. See *Part 2: Class Reference* for details on constructors for `RWCollectableStrings`. The insertions were not done in any particular order.

8. A pointer to a `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 (*i.e.*, `operator()`) has been overloaded for the iterator to mean "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” on page 90 in Chapter 13, “Introduction to Collection Classes,” of this manual. The typecast

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

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

11. Finally, the pointer `str` is dereferenced and printed. The ability of a `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 in lexicographical order.

16

Overview

This section gives a general overview of the various Smalltalk-like collection classes to help you chose an appropriate one for your problem.

Bags versus Sets versus Hash Tables

Class `RWHashTable` is the simplest to understand. It uses a simple hashed lookup to find the "bucket" that a particular object occurs in, then does a linear search of the bucket to find the object. A key concept is that more than one object with the same value (that is, that tests "isEqual") 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. Subsequent occurrences merely increment an "occurrence" count. It is implemented as a dictionary where the key is the inserted object and the value is the occurrence count. This is how 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 are not true Bags.

Class `RWSet` inherits from `RWHashTable`. It is similar except that duplicates are not allowed. That is, if you try to insert more than one object with a given value, duplicates are rejected.

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

Note – The ordering of objects in any of these hash-table based classes is not meaningful. If ordering is important, then you should chose a sequenceable class.

Sequenceable classes

Classes inheriting from `RWSequenceable` have an innate ordering. That is, it is meaningful to speak of the "6'th object", or the "first" or "last" object.

These classes are generally implemented either as a vector or as a singly- or doubly-linked list. You should be aware of the differences: vector based classes make good stacks and queues, but are poor at insertions in the middle. If you exceed the capacity of a vector-based collection class it will automatically resize, but there may be a very significant performance penalty for it to do so.

Note – The binary and B-Tree classes could be considered "sequenceable" in the sense that they are sorted and, therefore, have an innate ordering. However, the ordering is determined internally, by the relative value of the collected objects, rather than by an insertion order. That is, you cannot arbitrarily insert an object into a sorted collection in any position you wish: it might not remain sorted. Hence, these classes are subclassed separately.

Dictionaries

Dictionaries (sometimes called “maps”) use an external *key* to find a *value.* The key and value may be (and usually are) of different types. You can think of them 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 contrasts with the approach one might make using 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). Both keys and values must inherit from the abstract base class `RWCollectable`.

Virtual functions inherited from RWCollection

The Smalltalk-like collection classes inherited from the abstract base class `RWCollection` which, in turn, inherits from the abstract base class `RWCollectable`, described in Chapter 18, “Designing an RWCollectable Class” (making it is possible to have collections of collections).

An "abstract base class" is a class that is not intended to be used *per se*, but rather to be inherited by some other class. Its virtual functions provide a template of functionality that act as a surrogate for the derived class. The class `RWCollection` is such a class. It provides a template for "collection" classes by declaring various virtual functions such as `insert()`, `remove()`, `entries()`, *etc.*

This section describes the virtual functions inherited by all of the Smalltalk-like collections. Any collection can be expected to understand them.

insert()

```
virtual RWCollectable*insert(RWCollectable*);
```

A pointer to an object is put into a collection by using `insert()`. It inserts in the "most natural way" for the collection. For a stack, this means it is pushed onto the stack. For a queue, the item is appended to the queue. For a sorted collection, the item is inserted such that items before it compare less than (or equal to if duplicates are allowed) itself, items after it compare greater than itself. See the example in “Example” on page 116 for an example of `insert()`.

find() and friends

```
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;
```

These functions test how many objects the collection contains and whether it contains a particular object. 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 (with the `hash()` function used to narrow the choices). Sorted collections search for an item that "compares equal" (*i.e.* `compareTo()` returns zero) to the argument.

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.

Here is an example that builds on the example in “Example” on page 116 and uses some of these functions:

Code Example 16-1

```
#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
```

Code Example 16-1 (Continued)

```
    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
    return 0;
}
```

Program output:

```
5
Mary
2
```

Here's the line-by-line description:

1–7. These lines are as in “Example” on page 116.

8. Insert another instance with the value "Mary".

9. This statement prints out the total number of entries in the sorted collection: 5.

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 a `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

```
    virtual RWCollectable*remove(const RWCollectable*);
v   irtual 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 was found.

The function `removeAndDestroy()` is similar except that rather than return the item, it deletes 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 *(i.e.* not the stack) and that it is not shared with another collection.

Expanding on the example above:

```
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 was no such item.

18. Delete 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 not only removed, but also deleted.

apply() functions

```
virtual void apply(RWapplyCollectable ap, void* x);
```

An efficient method for examining the members of a Smalltalk-like collection is member function `apply()`. The first argument (`RWapplyCollectable`) is a typedef:

```
typedef void(*RWapplyCollectable)(RWCollectable*, void*);
```

that 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 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. It could be used, for example, to hold a handle to a window on which the object is to be drawn, *etc.*

Note – Notice the similarity to the `apply()` function of the generic collection classes (see "Apply functions" on page 110 in Chapter 15, ""Generic" Collection Classes"). The difference is in the type of the first argument of the user-supplied function (`RWCollectable`* rather than *type**). As with the generic classes, 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 the user to supply a separate function.[1]

Functions clear() and clearAndDestroy()

```
virtual void   clear();
virtual void   clearAndDestroy();
```

The function `clear()` removes all items from the collection.

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.

The function `clearAndDestroy()` not only removes the items, but also deletes each one. Although it *does* check to see if the same item occurs more than once in a collection (by building an IdentitySet internally) and thereby deletes each item only once, it must still be used with care. It cannot check to see whether an item is shared between two *different* collections. You must also be certain that all possible members of the collection were 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

```
RWBagasBag()                      const;
RWSetasSet()                      const;
RWOrderedasOrderedCollection()    const;
RWBinaryTreeasSortedCollection()  const
```

These functions allow any collection class to be converted into a `RWBag`, `RWSet`, `RWOrdered`, or a `SortedCollection` (*i.e.*, a `RWBinaryTree`).

Inserting and removing other collections

```
voidoperator+=(const RWCollection&);
voidoperator-=(const RWCollection&);
```

These functions insert or remove (respectively) the contents of their argument.

Selection

```
typedef RWBoolean
    (*RWtestCollectable)(const RWCollectable*, const void*);
RWCollection*select(RWtestCollectable tst, void*);
```

The function `select()` 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*, hence you are responsible for deleting it when done.

Virtual functions inherited from RWSequenceable

The abstract base class `RWSequenceable` is derived from `RWCollection`. Collections that inherit from it have an innate ordering. That is, the ordering is meaningful (unlike, say, a hash table).

```
virtual RWCollectable*&          at(size_t i);
virtual const RWCollectable*     at(size_t i) const;
```

These virtual functions allow access to the i'th item in the collection, similar to subscripting an array. The compiler choses which function to use on the basis of whether or not your collection has been declared "const". If it has, the second variant is used, otherwise the first. The first can be used as an lvalue:

```
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
```

These operation are very efficient for the class `RWOrdered` (which is implemented as a vector) but, as you might expect, relatively inefficient for classes implemented as a linked-list (the entire list must be traversed in order to find a particular index).

```
virtual RWCollectable*first() const;
virtual RWCollectable*last() const;
```

These functions return the first or last item in the collection, respectively, or nil if the collection is empty.

```
virtual size_tindex(const RWCollectable*) const;
```

This 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:

```
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:

```
1
```

Use the following function to insert an item at a particular index:

```
virtual RWCollectable*  insertAt(size_t, i,
     RWCollectable* c);

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

Caution – It is important to note that it is the virtual functions of the object *within the collection* that gets called when comparing or testing a target for equality, not that of the target.

For example, consider the following code fragment:

```
SortedCollection sc;
RWCollectableString member;
sc.insert(&member);
RWCollectableString target;
RWCollectableString* p =
(RWCollectableString*)sc.find(&target);
```

It is the virtual functions of the objects within the collection, such as `member`, that will get called, not the virtual functions of `target`:

```
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 of the collection classes that use it use the same general strategy. First, member function `hash()` of the target is called to find the proper bucket within the hash table. Buckets are implemented as a singly-linked list. Because all of the members of a bucket have the same hash value, the bucket must be 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.

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 not be visited in any particular order.

Persistence 17

All of the examples of persistence that we have looked at so far involve simple objects that do not reference other objects. It is time to look at some more complicated cases.

Operators

The storage and retrieval of objects that inherit from `RWCollectable` is a powerful and adaptable feature of the Tools.h++ Class Library. It is done through the following eight functions:

```
// Storage of a RWCollectable reference:
RWvostream&operator<<(RWvostream&, const RWCollectable&);// 1
RWFile&operator<<(RWFile&,     const RWCollectable&);// 2

// Storage of a RWCollectable pointer:
RWvostream&operator<<(RWvostream&, const RWCollectable*);// 3
RWFile&operator<<(RWFile&,     const RWCollectable*);// 4

// Retrieval into an existing RWCollectable:
RWvistream&operator>>(RWvistream&, RWCollectable&);// 5
RWFile&operator>>(RWFile&,     RWCollectable&);// 6

//Retrieval into an RWCollectable to be allocated off the heap:
RWvistream&operator>>(RWvistream&, RWCollectable*&);// 7
RWFile&operator>>(RWFile&,     RWCollectable*&);// 8
```

These function not only allow the storage and retrieval of collections and their inserted objects, but also their *morphology*. For example, a collection with multiple pointers to the same object could be be saved and restored. Or a circularly linked list.

Note – this ability to restore the morphology of an object is a property of the base class `RWCollectable`. It can be used by any object that inherits from `RWCollectable`, not just the "Smalltalk-like" collection classes (which inherit from `RWCollection`, a derived class of `RWCollectable`). We will see how to do this in folowing chapter.

Example

Here's an example of the use of these functions that builds on "Example" on page 116 in Chapter 16, "Smalltalk-like Collection Classes."

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

main()
{

  // Construct an empty collection:
  SortedCollection sc;

  // Insert, but to make things interesting,
  // add an object twice.

  RWCollectableString* george = new
    RWCollectableString("George");

  sc.insert(george);// Insert once
  sc.insert(new RWCollectableString("Mary"));
  sc.insert(george);// Insert twice
  sc.insert(new RWCollectableString("Bill"));
  sc.insert(new RWCollectableString("Throkmorton"));
  // Store in ascii to standard output:
  RWpostream ostr(cout);      // 1
  ostr << sc;                            // 2

  sc.clearAndDestroy();          // 3

  return 0;
}
```

Note – we have inserted one item into the collection twice. That is, two items in the collection are identical.

```
RWvostream&operator<<(RWvostream&, const RWCollectable&);
```

which stored a *shallow copy* of the collection. That is, only only one copy of "George" was stored.

As a side note, the expression on line 3 deletes all the members of the binary tree. The function `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.

The resulting image can be read back in and faithfully restored using the companion member function

```
RWvistream&operator>>(RWvistream&, RWCollectable&);
```

Here is how to do this:

```
#define RW_STD_TYPEDEFS
#include <rw/bintree.h>
#include <rw/collstr.h>
#include <rw/pstream.h>

main()
{
  RWpistream istr(cin);
  SortedCollection sc2;

  // Read the collection back in:
  istr >> sc2;// 4

  RWCollectableString temp("George");// 5
  // Find a "George":
  RWCollectableString* g =
    (RWCollectableString*)sc2.find(&temp);// 6

  // "g" now points to a string with the value "George"
  // How many occurrences of g are there in the collection?

  unsigned count = 0;// 7
  SortedCollectionIterator sci(sc2);// 8
  RWCollectableString* item;
  while ( item = (RWCollectableString*)sci() )// 9
    if ( item==g )//10
      count++;

  cout << count;

  sc2.clearAndDestroy();//11

  return 0;
}
```

Program output:

```
2
```

Here’s the line-by-line description:

4 On this line, the function

`RWvistream& operator>>(RWvistream&, RWCollectable&);`
restores the contents of the `SortedCollection` from the input virtual stream `istr`.

5 A temporary string with value “George” is created in order to search for a string within the newly created `SortedCollection` with the same value.

6 The `SortedCollection` is searched for an occurrence of a string with value “George”. The pointer "`g`" points to such a string.

7 Here’s how we can prove that there are actually two entries in the collection that point to the same George. Initialize a counter to zero.

8 As before, create an iterator from the collection.

9 Iterate through the collection, item by item, returning a pointer for each item.

10 Test whether this pointer equals g. That is, test for identity, not just equality.

11 Delete the objects created in line 4.

The program's output is "2", indicating that there are actually two pointers to the same object "George".

It is worth looking at what happened in line 4 in more detail. The expression

```
istr >> sc2;
```

calls the function

```
RWvistream& operator>>(RWvistream& str, RWCollectable&
 obj);
```

This function has been written to call the object `obj`'s virtual function `restoreGuts()`. In this case, `obj` is a binary tree 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 r-shift 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 function encounters a reference to a previous object, it just returns the old address. These pointers are inserted into the collection by the binary tree's `restoreGuts()`.

This is why only one instance of "George" was returned.

However, you do not have to know these details until you write your own class `RWCollectable` class, something we will do in the following chapter. However, at this point, you should note that when Smalltalk-like collection classes are restored they necessarily do not know the types of objects they will

1. Actually, the Smalltalk collection classes are so similar that they all share the same version of `restoreGuts()`, inherited from `RWCollection`.

be restoring. Hence, they must allocate them off the heap. This means that you are responsible for deleting the restored contents. This happened in line 11 of the example.

Summary

In the example in the previous section we saw the use of two operators for storage and retrieval, respectively, into virtual streams:

```
RWvostream&operator<<(RWvostream&, const
     RWCollectable&);// 1
RWvistream&operator>>(RWvistream&,
     RWCollectable&);// 2
```

There are two similar operators for storage and retrieval into `RWFile`'s:

```
RWFile&operator<<(RWFile&,     const
     RWCollectable&);// 3
RWFile&operator>>(RWFile&,
     RWCollectable&);// 4
```

In addition to these operators, there are also two operators for storage of pointers to `RWCollectables`. These operators have nearly identical semantics to their "const reference" counterparts (functions 1 and 3, respectively). The only difference is that they can detect and restore nil pointers.

```
RWvostream&operator<<(RWvostream&, const
     RWCollectable*);// 5
RWFile&operator<<(RWFile&,     const
     RWCollectable*);// 6
```

Finally, there are two operators for retrieval into a reference to a `RWCollectable` pointer:

```
RWvistream&operator>>(RWvistream&, RWCollectable*&
            obj);// 7
```

```
RWFile&operator>>(RWFile&,     RWCollectable*&
        obj);// 8
```

As mentioned in the previous section, these last two operators *allocate an object off the heap,* restore into it, then return a reference to the resultant pointer. You are responsible for deleting this object when done. In “Example” on page 130, why did we chose to use function 2, instead of 7? That is, why say

```
RWpistream istr(cin);
RWBinaryTree bt2;

istr >> bt2;

...

bt2.clearAndDestroy();
```

instead of

```
RWpistream istr(cin);
RWBinaryTree* pTree;

istr >> pTree;

...

pTree->clearAndDestroy();
delete pTree;
```

The answer is that we could have done it either way. However, if you know the type of the object, then you are usually better off allocating it yourself, then restoring via function 2. The reason is that it takes time for the persistence machinery to figure out the type of object and have the factory (see “An aside: the RWFactory” on page 142 in Chapter 18, “Designing an RWCollectable Class”) allocate one. Furthermore, you can tailor the allocation to suite your needs. For example, you might decide to set an initial capacity for a collection class.

Designing an RWCollectable Class 18

Up until now we have been persisting classes that come with Tools.h++. In this section we will look at how to create our own subclass of RWCollectable. This will allow it to use the persistence machinery or to be used by the Smalltalk-like collection classes.

Virtual functions inherited from RWCollectable

Class RWCollectable declares the following virtual functions (See *Part II: Class Reference* for a complete description of class RWCollectable):

```
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;
```

(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 the class designer to provide definitions that are tailored to the class at hand.

Example

Suppose we are running a bus company. We want to write a class that represents a bus, its set of potential customers, and its set of actual passengers. In order to be a passenger, you must be a customer. Hence, the set of customers is a superset of the set of passengers. Because 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, we want to make sure there are no duplicates in either list.

We must be careful when it comes time to persist the bus. If we naïvely iterate over the set of customers and then over the set of passengers, saving each one, then if a person is both a customer and a passenger we will save him twice. When the bus is restored, instead of the list of passengers referring to people already in the customer list, it will have its own separate instances.

We need some way of recognizing when a person has already been saved to the stream and, instead of saving him or her again, merely save a reference to the previous instance.

This is the job of class `RWCollectable`. Objects inheriting from `RWCollectable` have the ability to save not only their contents, but also their relationships with other objects inheriting from `RWCollectable`. We call this isomorphic persistence.

Here’s how we might declare our class Bus:

```
class Bus : public RWCollectable
{

  RWDECLARE_COLLECTABLE(Bus)

public:
  Bus();
  Bus(int busno, const RWCString& driver);
  ~Bus();

  // Inherited from class "RWCollectable":
  RWspacebinaryStoreSize() const;
  int   compareTo(const RWCollectable*) const;
  RWBooleanisEqual(const RWCollectable*) const;
  unsignedhash() 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_tcustomers() const;
  size_tpassengers() const;
  RWCStringdriver() const{return driver_;}
  intnumber() const{return busNumber_;}

private:

  RWSet customers_;
  RWSet*passengers_;
  int   busNumber_;
  RWCStringdriver_;
};
```

Note – how class Bus inherits from `RWCollectable`. We have chosen to implement the set of customers by using class `RWSet`. 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, for novelty, we have chosen to have this set live on the heap. This will serve to illustrate some points in the coming discussion.

Steps to making an RWCollectable object

Here are the general steps to making your object inherit from `RWCollectable`. Specific details are given below.

- Define a default constructor;
- Add the macro `RWDECLARE_COLLECTABLE` to your class declaration;
- Add the macro `RWDEFINE_COLLECTABLE` to a `.cc` file, to be compiled;
- Add definitions for the following inherited virtual functions as necessary (you may be able to use inherited definitions):

```
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;
```

Add a default constructor

All collectable classes are required to have a default constructor (*i.e.*, one that takes no arguments). This constructor will be used by the persistence mechanism: first an “empty” object is created, then it will be “restored” with its contents.

Because default constructors are necessary in order to create vectors of objects in C++, providing a default constructor is a good habit to get into anyway. Here’s a possible definition for our Bus class.

```
Bus::Bus() :
  busNumber_  (0),
driver_     (“Unknown”),
 passengers_ (rwnil)
{
}
```

RWDECLARE_/RWDEFINE_COLLECTABLE()

Note – the macro invocation “`RWDECLARE_COLLECTABLE`(*className*)” in the declaration for `Bus`. You must put this macro in your class declaration[1], using the class name as the argument. This macro automatically inserts declarations for functions `isA()` and `newSpecies()`.

A corresponding macro “`RWDEFINE_COLLECTABLE`(*className*, *classID*)” must be put in your `.cc` file, using the class name as the first argument, and its chosen class ID as the second number (see below). It will automatically supply the definitions for `isA`() and `newSpecies()`.

Virtual function isA( )

```
virtual RWClassIDisA() const;
```

The virtual function `isA()` returns a “class ID”, a unique number that identifies an object’s class. It can be used to determine the class to which an object belongs. The name "`RWClassID`" is actually a typedef to an `unsigned short`. Numbers from 0x8000 (hex) and up are reserved for use by Tools.h++. 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()`. The macro `RWDEFINE_COLLECTABLE`(*className, classID*) 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. The declaration of this function is automatically provided by the macro `RWDECLARE_COLLECTABLE`(*className*). The definition is automatically provided by the macro `RWDEFINE_COLLECTABLE`(*className,classID*).

1. At the time of this writing, this requirement is not being enforced to insure backwards compatibility with earlier versions of Tools.h++. However, it will be in the future.

An aside: the RWFactory

As mentioned, you must include the macro RWDEFINE_COLLECTABLE somewhere in a .cc file where it will be compiled. For our example, the macro might look like this:

```
RWDEFINE_COLLECTABLE(Bus, 200)
```

This magic incantation will allow a new instance of your class to be created given only its RWClassID:

```
Bus* newBus = (Bus*)theFactory->create(200);
```

This is used internally by the persistence machinery to create a new instance of your persisted object where its type is not known at runtime. You will not normally use this capability in your own source code. The pointer theFactory is a global pointer that points to a one-of-a-kind global instance of class RWFactory, used to create new instances of any class, given its RWClassID. The use of RWFactory is generally transparent to the user. See Part II: *Class Reference* for more details on RWFactory.

Object destruction

All objects inheriting from class RWCollectable inherit a virtual destructor. Hence, the actual type of the object need not be known until runtime 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 any resources that they hold. In the case of Bus, the names of passengers and customers are RWCollectableStrings 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 confidently 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 only necessary, indeed, it is essential, that we only delete the contents of customers_.

Here's a possible definition:

```
Bus::~Bus()
{
  customers_.clearAndDestroy();
  delete passengers_;
}
```

Note – the language guarantees that it is okay to call delete on the pointer passengers_ even if it is nil.

Virtual function compareTo()

```
virtual intcompareTo(const RWCollectable*) const;
```

The virtual function `compareTo()` is used to order objects relative to each other in the collection classes that depend on such ordering, such as `RWBinaryTree` or `RWBTree`.

Note – 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. This last case is sometimes referred to as "comparing equal", not be be confused with "is equal" (see "Virtual function isEqual()" on page 145).

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, as defined in class `RWCollectable`, is to compare the two addresses of the objects.

The default definition (comparing the addresses of the two objects) should really be thought of as a placeholder—in practice, it is not very useful and could vary from run-to-run of a program.

Here is an example that uses `compareTo()` as well as `isEqual()` and `hash()`.

```
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 – the `compareTo()` function for `RWCollectableStrings` has been defined to compare strings lexicographically in a case sensitive manner. See “RWCString” on page 256 in Chapter 22, “Class Reference” for details.

Here is a possible definition of `compareTo()` for our example:

```
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 busses. Hence, were a group of busses to be inserted into a `RWBinaryTree`, they would be sorted by their bus number.

Note – there are many other possible choices—we could have used the driver name, in which case they would have been sorted by that. Which choice you use will depend on your particular problem.

Of course, 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, a `RWCollectableString` into the collection, then the results of the cast `(const Bus*)c` would be invalid and dereferencing it could bring disaster. This is a glaring deficiency in C++ that the user must constantly be aware of. The necessity for all virtual functions to have all the same signatures requires that they return the lowest common denominator, in this case, class `RWCollectable`. The result is that all compile-time type checking breaks down.

One 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 function `isA()` can be used for this.

Virtual function isEqual()

```
RWBooleanisEqual(const RWCollectable* c) const;
```

The virtual function `isEqual()` plays a similar role to the "tester function" of the generic collection classes (see "Tester functions" on page 106 in Chapter 15, ""Generic" Collection Classes").

Note – The function "`RWBoolean isEqual(const RWCollectable*)`" should return `TRUE` if the object and its argument are considered "equal", `FALSE` otherwise. The definition of "equal" 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.*

"`isEqual`" need not necessarily be defined as "being identical", that is, having the same address (although this is the default), but rather, that they 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 inherit type `RWCollectable`. However, you are responsible for making sure that any typecasts you do are appropriate.

There is no formal requirement that two objects which "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.

For our example above, an appropriate definition might be:

```
RWBoolean Bus::isEqual(const RWCollectable* c) const
{
  const Bus* b = (const Bus*)c;
  return busNumber_ == b->busNumber_;
}
```

Here we are considering busses to be "equal" if their bus numbers are the same. Again, other choices are possible.

Virtual function hash()

```
unsigned   hash() const;
```

Note – The function `hash()` should return an appropriate hashing value for the object.

A possible definition for `hash()` for our example might be:

```
unsigned Bus::hash() const
{
  return (unsigned)busNumber_;
}
```

Here, we have just returned the bus number as a hash value. Alternatively, we could have hashed the driver's name:

```
unsigned Bus::hash() const
{
  return driver_.hash();
}
```

How to add persistence

To add persistence to your `RWCollectable` class, you must override the `saveGuts()` and `restoreGuts()` virtual member functions so they write out enough information to recreate the state of your object.

Virtual functions saveGuts(RWFile&) and saveGuts(RWvostream&)

These virtual functions are responsible for storing the internal state of an `RWCollectable` object on either a binary file (using class `RWFile`) or on a virtual output stream (an `RWvostream`). This allows the object to be recovered at some later time.

Start by saving the state of your base class by calling its version of `saveGuts()`.

Then, for each type of member data, save its state.

How to do this will depend on the type of the member data:

- For primitives, save the data directly. This means when storing to `RWFiles` use `RWFile::Write()`; for virtual streams, use the l-shift operator `RWvostream::operator<<()`.
- For Tools.h++ classes, most offer an overloaded version of the l-shift operator. For example, for `RWCStrings`:

```
RWvostream& operator<<(RWvostream&, const
                      RWCString& str);
```

Hence, these can simply be shifted onto the stream.

- For objects inheriting from `RWCollectable` this means using the global function

```
RWvostream& operator<<(RWvostream&, const
                          RWCollectable& obj);
```

This function will call `saveGuts()` recursively for the object.

With these rules in mind, here is a possible definition for our `Bus` example:

```
void Bus::saveGuts(RWFile& f) const
{
  RWCollectable::saveGuts(f);// Save base class
  f.Write(busNumber_);// Write primitive directly
  f << driver_ << customers_;// Use Tools.h++ 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 Tools.h++ provided
       versions
  strm << passengers_;// Will detect nil pointer
       automatically
}
```

Member data `busNumber_` is a C++ primitive, that is, an `int`. It is stored directly using `RWFile::Write(int)` in the case of `RWFiles`, or `RWvostream::operator<<(int)` in the case of virtual streams.

Member data `driver_` is a `RWCString`. It does not inherit from `RWCollectable`. It is stored using

```
RWvostream& operator<<(RWvostream&, const RWCString&);
```

Member data `customers_` is a `RWSet`. It does inherit from `RWCollectable`. It is stored using

```
RWvostream& operator<<(RWvostream&, const
                          RWCollectable&);
```

Finally, member data passengers_ is a little tricky. It is a pointer to an `RWSet` which inherits from `RWCollectable`. However, there is the possibility that the pointer is nil. If it were, then passing it to

```
RWvostream& operator<<(RWvostream&, const
                          RWCollectable&);
```

would be disastrous as we would have to dereference `passengers_`:

```
strm << *passengers_;
```

Instead, we pass it to

```
RWvostream& operator<<(RWvostream&, const
                              RWCollectable*);
```

which automatically detects the nil pointer and stores a record of it.

Multiply-referenced objects

Note – 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 this relationship is maintained.

We don't have to do anything! Consider the call:

```
Bus aBus;
RWFile aFile("busdata.dat");

aBus.addPassenger("John");
aFile << aBus;
```

Note – the name "John" will be on both the customer list and the passenger list. Note that *both* lists will be saved in this single call: first the customer list, then the passenger list. The persistence machinery will save the first reference to "John", but for the second reference it will merely store a reference to the first copy. During the restore (see below), both references will resolve to the same object, replicating the original morphology of the collection.

Virtual functions restoreGuts(RWFile&) and restoreGuts(RWvistream&)

In a similar manner to `saveGuts()`, these virtual functions are used to restore the internal state of an `RWCollectable` from a file or stream. Here is a definition for `Bus`:

```
void Bus::restoreGuts(RWFile& f)
{
  RWCollectable::restoreGuts(f);// Restore base class
  f.Read(busNumber_);// Restore primitive
  f >> driver_ >> customers_;// Uses Tools.h++ 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 how the pointer passengers_ is restored using

```
RWvistream& operator>>(RWvistream&, RWCollectable*&);
```

If the original `passengers_` was non-nil, then this function will restore a new `RWSet` off the heap and return a pointer to it. Otherwise, it will return a nil pointer. Either way, the old contents of `passengers_` will be replaced. Hence, we must call "`delete passengers_`" first.

Virtual function binaryStoreSize()

The virtual function

```
virtual RWspacebinaryStoreSize() const;
```

is used to calculate the number of bytes necessary to store an object using `RWFile`. This is useful for classes `RWFileManager` and `RWBTreeOnDisk` which require space to be allocated for an object before it can be stored.

Writing a version of `binaryStoreSize()` is usually very straightforward. Just follow the pattern set by `saveGuts(RWFile&)`, except that instead of saving member data, add up their sizes. The only real difference is a syntactic one: instead of l-shift operators, you use member functions and `sizeof()`:

- For primitives, use `sizeof()`;
- For objects that inherit from `RWCollectable`, use member function
  ```
  RWspace RWCollectable::recursiveStoreSize();
  ```
- For other objects, use member function `binaryStoreSize()`.

Here's a sample definition for class `Bus`:

```
RWspace Bus::binaryStoreSize() const
{
  RWspace count = RWCollectable::binaryStoreSize() +
    customers_.recursiveStoreSize() +
    sizeof(busNumber_) +
    driver_.binaryStoreSize();

  if (passengers_)
    count += passengers_->recursiveStoreSize();

  return count;
}
```

Persisting custom collections

Class `RWCollection` has versions of `saveGuts()` and `restoreGuts()` built into it that are sufficient for most collection classes.

`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*&);
```

which will allocate a new object of the proper type off the heap, followed by `insert()`. Because all of the Tools.h++ Smalltalk-like collection classes inherit from `RWCollection`, they all use this mechanism.

If you decide to write your own collection classes and 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 (essentially, you would get the degenerate case of a linked list) when the tree was read back in. Hence, `RWBinaryTree`'s version of `saveGuts()` stores the tree level-by-level.

Summary

In general, you may not have to supply definitions for all of these virtual functions. 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 do it 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":
  RWspacebinaryStoreSize() const;
  int   compareTo(const RWCollectable*) const;
  RWBooleanisEqual(const RWCollectable*) const;
  unsignedhash() 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_tcustomers() const;
  size_tpassengers() const;
  RWCStringdriver() const{return driver_;}
  int   number() const{return busNumber_;}

private:

  RWSet customers_;
  RWSet*passengers_;
  int   busNumber_;
  RWCStringdriver_;
};

#endif
```

BUS.CC:

```
#include "bus.h"
#include <rw/pstream.h>
#include <rw/rwfile.h>
#ifdef __ZTC__
# include <fstream.hpp>
#else
```

```
# ifdef __GLOCK__
#   include <fstream.hxx>
# else
#   include <fstream.h>
# endif
#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 Tools.h++ 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 Tools.h++ 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 Tools.h++ 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)
{
  RWCollectableString* s = new RWCollectableString(name);
  customers_.insert( s );

  if (!passengers_)
    passengers_ = new RWSet;

  passengers_->insert(s);
}

void
Bus::addCustomer(const char* name)
{
  customers_.insert( new RWCollectableString(name) );
}

main()
{
  Bus theBus(1, "Kesey");
  theBus.addPassenger("Frank");
  theBus.addPassenger("Paula");
  theBus.addCustomer("Dan");
  theBus.addCustomer("Chris");

  {
    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 Kesey.
It has 4 customers and 2 passengers.
```

Errors 19

Thinking about error handling is like thinking about where the garbage man hauls your trash—it's a messy, unpredicatable, and sour topic, one that we don't like to think about. Yet, to write robust code, think about it we must.

The Tools.h++ class libraries use an extensive and complete error handling facility, all based on the same model. Errors are divided into two broad categories: internal errors and external errors. The distinguishing characteristic of *internal errors* is that they 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 default response is commonly 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 following sections describe the error model in detail.

Internal errors

Internal errors are due to faulty logic or coding in the program. Examples are

- Bounds errors;
- Inserting a null pointer into a collection;
- Attempting to use a bad date.

In theory, all of these errors are preventable. For example, the permissible range of indices for an array is always known, and so a bounds error should be avoidable. As another example, while your program may not know that a date is bad, once it has found this out, to use it would be an obvious logic error.

Internal errors are further divided into two categories, dependent on the cost of error detection and whether or not the error will be detected at runtime:

- Non-recoverable internal errors
- Recoverable internal errors

Non-recoverable internal errors

Distinguishing characteristics:

- Easily predictable in advance.
- Usually occur at a relatively low level.
- Cost of detection is high.
- Detected only in the “debug” version of the library.

Examples:

- Bounds error
- Inserting a nil pointer into a collection

Response:

- No recovery mechanism.

Detecting errors costs time. For performance reasons, a library may have to demand some minimal level of correctness on the part of your program. Anything that falls short we term a *non-recoverable internal error.* They are “non-recoverable” because in the production version of the library there is no attempt to detect such errors and, hence, no opportunity to recover from one.

An example is bounds checking: 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 that the user always use a valid index.

Because a minimum level of correctness is being demanded, non-recoverable errors must be relatively easy to avoid and simple in concept.

Non-recoverable errors are best discovered and eliminated by compiling and linking your application with the debug version of the library. See Section , “Debug version of the library” 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 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

Distinguishing characteristics:

- Similar to “Non-recoverable internal errors” (see above) except:
- Cost of detection is low.
- Detected in the debug and production version of the library.

Examples:

- Attempt to use an invalid date
- Bounds error in a linked list

Response:

- Throw an exception inheriting from `RWInternalErr`.

If the cost of error detection is relatively low, then it starts to make sense to detect an error even in a production version of the library. An example is 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, then the library will throw an exception inheriting from `RWInternalErr`. Here’s an example from Tools.h++:

```
// Find link “i”; the index must be in range:
RWIsvSlink* RWIsvSlist::at(size_t i) const
{
  if (i >= entries())
    RWTHROW( RWBoundsErr( RWMessage(RWTOOL_INDEX,
    (unsigned)i,
    (unsigned)entries()-1) ) );
  register RWIsvSlink* link = head_.next_;
  while (i--) link = link->next_;
  return link;
}
```

Note – Note how the function always attempts to detect a bounds error. If it finds one, then it throws an instance of RWBoundsErr, a class that inherits from RWInternalErr. This instance contains an (internationalized) message (discussed in "Localizing messages" on page 174" in Chapter 20, "Implementation Notes"). The RWTHROW macro is discussed in "Error handlers" on page 164.

Throwing an exception gives you the opportunity to catch the exception and, possibly, recover. However, because the internal logic of the program has been compromised, most likely you will want to attempt to save whatever document is being worked on then abort the program.

External errors

Distinguishing characteristics:

- Cannot reasonably predict them in advance.
- Usually occur at a more abstract level.
- Hence, cost of detection is relatively low.
- Detected in all versions of the library.

Examples:

- Attempt to set a bad date (E.g., “31 June 1992”).
- Attempt to invert a singular matrix.
- Stream write error.
- Out of memory.

Response:

- Throw an exception inheriting from RWInternalErr;
- Or provide a test for object validity.

The distinguishing characteristic of *external errors* is that they are caused by external conditions and, hence, cannot reasonably be predicted in advance. In an object-oriented environment, runtime errors frequently show up as an attempt to set an object into an invalid state, perhaps as a result of invalid user input. An example is initializing a date object with a bad date (*e.g.*, 31 June 1992, a date that doesn’t exist).

Note – Note that the line between an internal and external error can sometimes be fuzzy. For example, the rules could say “don’t give me an invalid date” and then the programmer would be responsible for detecting a bad date before using a Tools.h++ routines. Of course, this is a lot of work and probably the reason why you bought the library in the first case: the `RWDate` object is probably in a much better position than you to detect invalid dates.

In theory, the response to an external error is either to throw an exception or to provide a test for object validity. It should never abort the program. However, in practice, exceptions have not been widely adopted by compilers and so Tools.h++ provides an opportunity to either test for a status value or to recover in an error handler.

Here’s an example:

```
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 previous sections. This hierarchy assumes the presence of class `xmsg`, nominally provided by your compiler vendor. This is a class that is 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`, then the Tools.h++ classes `RWxmsg` and `RWxalloc` emulate them.

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 by the specializing versions of `xmsg` as described in "Localizing messages" on page 174 in Chapter 20, "Implementation Notes."

Error handlers

When Tools.h++ throws an exception it does so using the macro `RWTHROW`. If your compiler supports exceptions then this macro is defined as follows:

```
#define RWTHROW(a) throw a
```

and a true exception will be thrown. Otherwise, if your compiler does not support exceptions, then it will 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);
```

Debug version of the library

Tools.h++ can be built in a "debug" mode. This is a very powerful tool for uncovering and correcting internal errors in your code.

To build a debug version of the library, the entire library must be compiled with the preprocessor flag "`RWDEBUG`" defined. *The entire library must be compiled with a single setting of the flag—either defined or not defined.* The resultant library 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.

The pre- and postconditions are implemented with “asserts”—a failure will cause the offending condition to be printed out, along with the file and line number where it occurred.

RWPRECONDITION
RWPOSTCONDITION

Bertrand Meyer, in his landmark book “Object-oriented Software Construction”[1], suggests regarding functions as a “contract” between a caller and a callee. If the caller agrees to abide by a set of “preconditions”, then the callee guarantees to return results that satisfy a set of “postconditions”. Here’s an example with a bounds error in C:

```
char buff[20];
char j = buff[20];// Bounds error!
```

Such bounds error are extremely tough to detect in C, but easy in C++:

```
RWCString buff(20);
char j = buff[20];// Detectable bounds error
```

The reason why is that `operator[]` can be overloaded to perform an explicit bounds check in the debug version of the library:

```
char& RWCString::operator[](size_t i)
{
  RWPRECONDITION(i < length();
  return rep[i];
}
```

1. Prentice Hall International, ISBN 0-13-629049-3.

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) );
}
```

The job of this function 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 nil. If this condition is satisfied, then the function guarantees to successfully insert the object. This is checked by the postcondition clause.

The macros `RWPRECONDITION` and `RWPOSTCONDITION` are defined in `<rw/defs.h>` and compile out to no-ops unless the preprocessor macro `RWDEBUG` has been defined:

```
#ifdef RWDEBUG
#  define RWPRECONDITION(a)assert(a)
#  define RWPOSTCONDITION(a)assert(a)
#else
#  define RWPRECONDITION(a)((void*)0)
#  define RWPOSTCONDITION(a)((void*)0)
#endif
```

Implementation Notes 20

Copy on write

Classes RWCString, RWWString, and RWTValVirtualArray 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 counting.

When a RWCString is initialized with another RWCString via the copy constructor

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

then the two strings will share the same data until one of them tries to write to it. At this point, a copy of the data is made and the two strings go their separate ways. This makes copies, particularly read-only copies, of strings very inexpensive. Consider the following example:

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

  setGlobal(a);// Still only one copy of "kernel"! // 4

  b += "s";// Now b has its own data: "kernels"// 5
}
```

1. The `RWCString` object "`a`" is where the actual allocation and initialization of the memory to hold the string "`kernel`" happens.

2–3 When objects "`b`" and "`c`" are created from it, 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 the global `RWCString g` 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. It is at this point that 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 `RWCStrings` 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. Here's an example. Suppose you have a window whose background and foreground colors can be set. A simple minded approach to do this would be:

```
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. Hence, calling `setForeground()` is efficient. But, 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 anyone 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. It's efficient too, because a copy of the data will not be made unless the string should change:

```
Smart window;
RWCString color("white");

window.setForeground(color);// Two references to white

color = "Blue";// One reference to white, one to blue
```

More on storing and retrieving RWCollectables

In Chapter 17, "Persistence" we saw how to use the 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*&);
```

to save and restore the morphology of a class (*i.e.*, the correct relationship between pointers).

When working with `RWCollectables`, it is useful to understand how these functions work. Here is a brief description.

When you call one of the l-shift (`<<`) operators for the first time for any collectable object, an `IdentityDictionary` is created internally. The object's address (*i.e.*, "`this`") is put in the table, along with its ordinal position in the output file (the first, the second, *etc*.). Once this has been done, a call is made to the object's virtual function `saveGuts()`. Because this is a *virtual* function, the call will go to the derived class's definition of `saveGuts()`. As we have seen, the job of `saveGuts()` is to store the internals of the object. If the object contains other objects inheriting from `RWCollectable` (as all of the collection classes do, as well as many other classes), then the object's `saveGuts()`, if it has been written correctly, will call `operator<<()` recursively for each of these objects. Subsequent invocations of `operator<<()` do not create a new `IdentityDictionary`, but do 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 (say, the sixth).

When the entire collection has been traversed and the initial call to `saveGuts()` returns, then the `IdentityDictionary` is deleted and the initial call to `operator<<()` returns.

The function `operator>>()` essentially reverses this process and, when encountering a reference to an object that has already been created, 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 feature:

```
#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=0; i<10; 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;
}
```

The class `Tangle` implements a (potentially) circularly linked list. What happens? When function `operator<<()` is called for the first time for an instance of `Tangle`, it sets up the `IdentityDictionary`, as described above, and then calls the `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 (i.e., `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<<()` will eventually be made for the first instance of `Tangle` again, 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*&);
```

will either 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 case of the last two choices, the recursion stops and the stack unwinds.

Multiple inheritance

In Chapter 18, "Designing an RWCollectable Class," we built a `Bus` class by inheriting from `RWCollectable`. If we had an existing `Bus` class at hand, we might have been able to use multiple inheritance to create a new class with the functionality of both `Bus` *and* `RWCollectable`, perhaps saving ourselves some work:

```
class CollectableBus : public RWCollectable, public Bus {
    .
    .
    .
};
```

This is the approach taken by many of the Tools.h++ collectable classes (*e.g.*, class `RWCollectableString`, which inherits both class `RWCollectable` and class `RWCString`). The general idea is to create your object first, and then tack on the `RWCollectable` class, making the whole thing collectable. This way, you will be able to use your objects for other things, in other situations, where you might not want to inherit from class `RWCollectable`.

There is another good reason for using this approach: avoiding 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()`. 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 — 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. The only solution is to exhaustively search all possible paths in the object's inheritance hierarchy, looking for a match. (This is the approach of the *NIH Classes*.) Such a solution is slow (it must be done for every cast, although the search can be speeded up by "memoizing" the resulting addresses), bulky and always complicated. We decided that this was unacceptable.

Hence, we chose the first route. This can be made acceptable by keeping the inheritance trees simple by not making everything derive from the same base class. Hence, rather than using a large secular base class (sometimes dubbed the "cosmic object"; an example is Smalltalk's "`Object`") 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 that you need (such as collectability), thus avoiding multiple base classes of the same type and the resulting ambiguous calls.

Localizing messages

Tools.h++ includes a facility for localizing messages, that is, formatting them in the native language of the user. This facility is used to localize exception messages, to be passed to `xmsg` (see Section , “Exception architecture,” on page 163 in Chapter 19, “Errors”). 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`

The Sun version of Tools.h++ is delivered to you using catgets().

Function `catgets()` uses a message number to look up a localized version of a message.

Common Mistakes 21

We have made every effort to "tune" our libraries so as to minimize the chance of a programming error. Nevertheless, C++ is an extremely complex language with countless opportunities for making some very subtle mistakes.

In writing this chapter, we went though our technical support documents to uncover the most common mistakes that our users were making. For those that could be prevented, we tried to rewrite the library to make them impossible. This is always the best approach, but may not always be possible. For example, unacceptable performance degradations may result. Or the language may not let you make the change.

This chapter summarized the most common mistakes that are left over. Take a look through this list and, of course, make sure you have read the manual, if you are having a problem.

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 particulary when creating new `RWCollectable` classes.

For example:

```
class MyObject : public RWCollectable {
public:
  RWBooleanisEqual();// 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

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". The return value after advancing will be "special", either `FALSE` or nil, depending on the type of iterator, if the end of the collection has been reached.

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:

```
main()
{
  RWCString* string = new RWCString;;
  RWFile file("mydata.dat");

  // WRONG:
  file >> string;// Memory leak!

  // RIGHT:
  delete string;
  file >> string;

}
```

Include path

Make sure that when you specify an include path to the Tools.h++ header files 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
```

Match library version with your application’s compiler options

If you compile with `-DRWDEBUG`, use `librwtool_dbg.a`, otherwise don’t use it.

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];
   ifstream fstr(argv[1]);
   RWCString line;

   while (fstr.readline(buf,sizeof(buf)) {
      line = buf;
      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:

```
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, it probably did to us. Most likely there's a better way!

Part 2— Class Reference

Class Reference 22

C++ is still a young language; therefore there is no standard way to structure a reference manual for a class or group of classes. The reference is presented here as an alphabetical listing of classes with their member and global functions grouped in categories according to their general use. The categories are not a part of the C++ language, but do provide a way of organizing the many functions.

Each class includes a brief description, an illustration showing its inheritance hierarchy, and a synopsis, indicating the header file(s) and `Smalltalk` typedef (if appropriate) associated with the class. The synopsis also shows a declaration and definition of a class object, and any typedefs that are used.

Member functions for each class are listed alphabetically. Member functions fall into three general types:

1. Functions that are *unique* to a class. The complete documentation for these functions is presented in the class where they occur. An example is `balance()`, a member of the class `BinaryTree`.
2. Functions that are *inherited* from a base class without being redefined. The complete documentation for these functions is presented in the defining *base class*. An example is `clearAndDestroy()`, for class `RWBinaryTree`, which is inherited from class `RWCollection`.

3. Functions that are *redefined* in a derived class. These are usually virtual functions. The documentation for these functions usually directs you to the base class, but may also mention peculiarities that are relevant to the derived class. An example is `apply()`, for class `RWBinaryTree`.

Throughout this chapter, there are frequent references to "self." This should be understood to mean "`*this`."

RWBag

RWBag
|
RWCollection
|
RWCollectable

Synopsis

```
typedef RWBag Bag;        // Smalltalk typedef.
```

```
#include <rw/rwbag.h>
RWBag h;
```

Description

Class `RWBag` corresponds to the `Smalltalk` class `Bag`. It represents a group of unordered elements, not accessible by an external key. Duplicates are allowed.

An object stored by `RWBag` must inherit abstract base class `RWCollectable`, with suitable definition for virtual functions `hash()` and `isEqual()` (see class `RWCollectable`). The function `hash()` is used to find objects with the same hash value, then `isEqual()` is used to confirm the match.

Class `RWBag` is implemented by using an internal hashed dictionary (`RWHashDictionary`) which keeps track of the number of occurrences of an item. If an item is added to the collection that compares equal (`isEqual`) to an existing item in the collection, then the count is incremented.

Note – This means that only the first instance of a value is actually inserted: subsequent instances cause the occurrence count to be incremented. This behavior parallels the `Smalltalk` implementation of `Bag`.

Member function `apply()` and the iterator are called repeatedly according to the count for an item.

See class `RWHashTable` if you want duplicates to be stored, rather than merely counted.

Public constructors

`RWBag(size_t n = RWDEFAULT_CAPACITY);`
Construct an empty bag with `n` buckets.

`RWBag(const RWBag& b);`
Copy constructor. A shallow copy of `b` will be made.

Public member operators

```
void                         operator=(const RWBag& b);
```
Assignment operator. A shallow copy of `b` will be made.

```
RWBoolean                    operator==(const RWBag& b)
                             const;
```
Returns `TRUE` if self and bag `b` have the same number of total entries and if for every key in self there is a corresponding key in `b` which isEqual and which has the same number of entries.

Public member functions

```
virtual void                 apply(RWapplyCollectable ap,
                              void*)
```
Redefined from class `RWCollection`. This function has been redefined to apply the user-supplied function pointed to by `ap` to each member of the collection in a (generally) unpredictable order. If an item has been inserted more than once *(i.e.*, more than one item isEqual), then `apply()` will be called that many times. The user-supplied function should not do anything that could change the hash value of the items.

```
virtual RWspace              binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void                 clear();
```
Redefined from class `RWCollection`.

```
virtual void                 clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                  compareTo(const RWCollectable*
                              a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean            contains(const RWCollectable*
                              target) const;
```
Inherited from class `RWCollection`.

```
virtual size_t                entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*       find(const RWCollectable*
                              target) const;
```
Redefined from class `RWCollection`. The first item that was inserted into the Bag and which equals `target` is returned or nil if no item is found. Hashing is used to narrow the search.

```
virtual unsigned          hash() const;
```
Inherited from class `RWCollectable`.

```
virtual RWCollectable*    insert(RWCollectable* c);
```
Redefined from class `RWCollection`. Inserts the item c into the collection and returns it, or if an item was already in the collection that *isEqual to* c, then returns the old item and increments its count.

```
RWCollectable*            insertWithOccurrences
                           (RWCollectable*c,size_t n);
```
Inserts the item `c` into the collection with count `n` and returns it, or if an item was already in the collection that *isEqual to* `c`, then returns the old item and increments its count by `n`.

```
virtual RWClassID         isA() const;
```
Redefined from class `RWCollectable` to return `__RWBAG`.

```
virtual RWBoolean         isEmpty() const;
```
Redefined from class `RWCollection`.

```
virtual RWBoolean         isEqual(const RWCollectable*
                           a)const;
```
Redefined to return TRUE is the object pointed to by `a` is of the same type as self, and self == `t`.

```
virtual size_t            occurrencesOf(const
                           RWCollectable* target) const;
```
Redefined from class `RWCollection`. Returns the number of items that *are equal to* the item pointed to by `target`.

```
virtual RWCollectable*    remove(const RWCollectable*
                           target);
```
Redefined from class `RWCollection`. Removes and returns the item that *isEqual to* the item pointed to by `target`. Returns nil if no item was found.

```
virtual void              removeAndDestroy(const
                           RWCollectable*  target);
```
Inherited from class `RWCollection`.

```
void                      resize(size_t n = 0);
```
Resizes the internal hash table to have `n` buckets. This will require rehashing all the members of the collection. If `n` is zero, then an appropriate size will be picked automatically.

```
virtual void              restoreGuts(RWvistream&);
virtual void              restoreGuts(RWFile&);
virtual void              saveGuts(RWvostream&) const;
virtual void              saveGuts(RWFile&) const;
```

Inherited from class `RWCollection`.

RWBagIterator

RWBagIterator
    |
RWIterator

Synopsis

```
#include <rw/rwbag.h>
RWBag b;
RWBagIterator it(b);
```

Description

Iterator for class `RWBag`, which allows sequential access to all the elements of `RWBag`.

Note – Because a `RWBag` is unordered, elements are not accessed in any particular order. If an item was inserted `N` times into the collection, then it will be visited `N` times.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWBagIterator(const RWBag&);`
Construct an iterator for `RWBag`. After construction, the position of the iterator is undefined.

Public member operator

`virtual RWCollectable*     operator()();`
Redefined from class `RWIterator`. Advances the iterator to the next line item and returns it. Returns nil when the end of the collection has been reached.

Public member functions

```
Virtual RWCollectable*     findNext (const RWCollectable*
                            target);
```

Redefined from class `RWIterator`. Moves iterator to the next item which *isEqual to* the object pointed to by `target` and returns it. Hashing is used to find the target. If no item is found, returns nil and the position of the iterator will be undefined.

```
Virtual RWCollectable*     key() const;
```

Redefined from class `RWIterator`. Returns the item at the current iterator position.

```
virtual void               reset();
```

Redefined from class `RWIterator`. Resets the iterator to its starting state.

RWBench

Synopsis

`#include <rw/bench.h>`
(*Abstract base class*)

Description

This is an abstract class that can automate the process of benchmarking a piece of code. To use it, derive a class from `RWBench`, including a definition for the virtual function `doLoop(unsigned long N)`. This function should perform N operations of the type that you are trying to benchmark. `RWBench` will call `doLoop()` over and over again until a preset amount of time has elapsed. It will then sum the total number of operations performed.

To run, construct an instance of your derived class and then call `go()`. Then call `report()` to get a standard summary. For many compilers, this summary will automatically include the compiler type and memory model. You can call `ops()`, `outerLoops()`, *etc.* for more detail.

If you wish to correct for overhead, then provide an `idleLoop()` function which should do non-benchmark related calculations.

Example

This example benchmarks the time required to return a hash value for a Tools.h++ string versus a Borland string.

Code Example 22-1 (1 of 3)

```
#include <rw/bench.h>/* Benchmark software */
#include <rw/cstring.h>/* Tools.h++ string class */
#include <strng.h>/* Borland string class */
#include <stdlib.h>

// The string to be hashed:
const char* cs = "A 22 character string.";

class TestBCCString : public RWBench {
public:
  TestBCCString() { }
  virtual voiddoLoop(unsigned long n);
  virtual voididleLoop(unsigned long n);
  virtual voidwhat(ostream& s) const
    { s << "Borland hashing string \"" << cs << "\"\n"; }
};

class TestRWCString : public RWBench {
public:
```

Code Example 22-1 (2 of 3)

```
  TestRWCString() { }
  virtual void  doLoop(unsigned long n);
  virtual void  idleLoop(unsigned long n);
  virtual void  what(ostream& s) const
    { s << "Tools.h++ hashing string \"" << cs << "\"\n"; }
};

main(int argc, char* argv[])
{
  cout << "Testing string \"" << cs << "\"\n\n";

  // Test Borland strings:
  TestBCCString bccstring;
  bccstring.parse(argc, argv);
  bccstring.go();
  bccstring.report(cout);

  // Test RW Strings:
  TestRWCString rwstring;
  rwstring.parse(argc, argv);
  rwstring.go();
  rwstring.report(cout);

  return 0;
}

void TestBCCString::doLoop(unsigned long n){
  String string(cs);
  hashValueType h;
  while(n--){
    h = string.hashValue();
  }
}

void TestRWCString::doLoop(unsigned long n){
  RWCString string(cs);
  unsigned h;
  while(n--){
    h = string.hash();
  }
}

void TestBCCString::idleLoop(unsigned long n){
```

Code Example 22-1 (3 of 3)

```
  String string(cs);// Subtract out constructor time
  hashValueType h;
  while(n--){ /* No-op */ }
}

void TestRWCString::idleLoop(unsigned long n){
  RWCString string(cs);// Subtract out constructor time
  unsigned h;
  while(n--){ /* No-op */ }
}
```

Program output:

```
Testing string "A 22 character string."

Borland C++ V3.0 Large memory model.

Borland hashing string "A 22 character string."

Iterations:163
Inner loop operations:1000
Total operations:163000
Elapsed (user) time:4.945055
Kilo-operations per second:32.962222

Borland C++ V3.0 Large memory model.

Tools.h++ hashing string "A 22 character string."

Iterations:417
Inner loop operations:1000
Total operations:417000
Elapsed (user) time:4.835165
Kilo-operations per second:86.243182
```

Public constructors

```
RWBench(double duration = 5, unsigned long ILO=1000,
                             const char* machine = 0);
```

The parameter `duration` is the nominal amount of time that the benchmark should take in seconds. The virtual function `doLoop(unsigned long)` will be called over and over again until at least this amount of time has elapsed. The parameter `ILO` is the number of "inner loop operations" that should be performed. This parameter will be passed in as parameter `N` to `doLoop(N)`. Parameter `machine` is an optional zero terminated string that should describe the test environment (perhaps the hardware the benchmark is being run on).

Public member functions

```
virtual void             doLoop(unsigned long N)=0;
```

A pure virtual function whose actual definition should be supplied by the specializing class. This function will be repeatedly called until a time duration has elapsed. It should perform the operation to be benchmarked `N` times. See the example.

```
double                   duration() const;
```

Return the current current setting for the benchmark test duration. This should not be confused with function `time()` which returns the actual test time.

```
virtual void             go();
```

Call this function to actually run the benchmark.

```
virtual void             idleLoop(unsigned long N);
```

This function can help to correct the benchmark for overhead. The default definition merely executes a `"for()"` loop `N` times. See the example.

```
virtual void             parse(int argc, char* argv[]);
```

This function allows an easy way to change the test duration, number of inner loops and machine description from the command line:

Table 22-1

Argument	**Type**	**Description**
`argv[1]`	`double`	Duration (sec.)
`argv[2]`	`unsigned long`	No. of inner loops
`argv[3]`	`const char*`	Machine

`virtual void                report(ostream&) const;`
Calling this function provides an easy and convenient way of getting an overall summary of the results of a benchmark.

`double                      setDuration(double t);`
Change the test duration to time `t`.

`unsigned long               setInnerLoops(unsigned long N);`
Change the number of "inner loop operations" to `N`.

`virtual void                what(ostream&) const;`
You can supply a specializing version of this virtual function that provides some detail of what is being benchmarked. It is called by `report()` when generating a standard report.

`void                        where(ostream&) const;`
This function will print information to the stream about the compiler and memory model that the code was compiled under.

`unsigned long               innerLoops() const;`
Returns the current setting for the number of inner loop operations that will be passed into function `doLoop(unsigned long N)` as parameter `N`.

`double                      time() const;`
Returns the amount of time the benchmark took, corrected for overhead.

`unsigned long               outerLoops() const;`
Returns the number of times the function `doLoop()` was called.

`double                      ops() const;`
Returns the total number of inner loop operations that were performed (the product of the number of times `outerLoop()` was called times the number of inner loop operations performed per call).

`double                      opsRate() const;`
Returns the number of inner loop operations per second.

22

RWBinaryTree

RWBinaryTree
|
RWCollection
|
RWCollectable

Synopsis

```
typedef RWBinaryTree SortedCollection;      // Smalltalk
                                               typedef.
#include <rw/bintree.h>
RWBinaryTree bt;
```

Description

Class `RWBinaryTree` represents a group of ordered elements, internally sorted by the `compareTo()` function. Duplicates are allowed. An object stored by a `RWBinaryTree` must inherit abstract base class `RWCollectable`.

Public constructors

```
RWBinaryTree();
```
Construct an empty sorted collection.

```
RWBinaryTree(const RWBinaryTree& t);
```
Copy constructor. Constructs a shallow copy from `t`. Member function `balance()` (see below), is called before returning.

```
virtual                ~RWBinaryTree();
```
Redefined from `RWCollection`. Calls `clear()`.

Public member operators

```
void                   operator=(const RWBinaryTree&
                        bt);
```
Sets self to a shallow copy of `bt`.

```
RWBoolean              operator<=(const RWBinaryTree&
                        bt) const;
```
Returns `TRUE` if self is a subset of the collection `bt`. That is, every item in self must compare equal to an item in `bt`.

```
RWBoolean              operator==(const RWBinaryTree&
                        bt) const;
```
Returns `TRUE` if self and `bt` are equivalent. That is, they must have the same number of items and every item in self must compare equal to an item in `bt`.

Public member functions

```
virtual void                apply(RWapplyCollectable ap,
                             void*)
```
Redefined from class RWCollection to apply the user-supplied function pointed to by `ap` to each member of the collection, in order, from smallest to largest. This supplied function should not do anything to the items that could change the ordering of the collection.

```
void                        balance();
```
Special function to balances the tree. In a perfectly balanced binary tree with no duplicate elements, the number of nodes from the root to any external (leaf) node differs by at most 1 node. Since this collection allows duplicate elements, a perfectly balanced tree is not always possible.

```
virtual RWspace             binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void                clear();
```
Redefined from class `RWCollection`.

```
virtual void                clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                 compareTo(const RWCollectable*
                             a)const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean           contains(const RWCollectable*
                             target)const;
```
Inherited from class `RWCollection`.

```
virtual size_t              entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*      find(const RWCollectable*
                             target)const;
```
Redefined from class `RWCollection`. Returns the first item that compares equal to the item pointed to by `target`, or nil if no item was found.

```
virtual unsigned            hash() const;
```
Inherited from class `RWCollectable`.

```
virtual RWCollectable*     insert(RWCollectable* c);
```
Redefined from class `RWCollection`. Inserts the item `c` into the collection and returns it. Returns nil if the insertion was unsuccessful. The item `c` is inserted according to the value returned by `compareTo()`.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWBINARYTREE`.

```
virtual RWBoolean          isEmpty() const;
```
Redefined from class `RWCollection`.

```
virtual RWBoolean          isEqual(const RWCollectable* a)
                            const;
```
Redefined to return TRUE is the object pointed to by `a` is of the same type as self, and self == `t`.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```
Redefined from class `RWCollection`. Returns the number of items that compare equal to the item pointed to by `target`.

```
virtual RWCollectable*     remove(const RWCollectable*
                            target);
```
Redefined from class `RWCollection`. Removes the first item that compares equal to the object pointed to by `target` and returns it. Returns nil if no item was found.

```
virtual void               removeAndDestroy(const
                            RWCollectable* target);
```
Inherited from class `RWCollection`.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
```
Inherited from class `RWCollection`.

```
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```
Redefined from class `RWCollection` to store objects by level, rather than in order. This results in the tree maintaining its morphology.

RWBinaryTreeIterator

RWBinaryTreeIterator
|
RWIterator

Synopsis

```
// Smalltalk typedef:
typedef RWBinaryTreeIterator SortedCollectionIterator;
```

```
#include <rw/bintree.h>
RWBinaryTree bt;
RWBinaryTreeIterator iterate(bt);
```

Description

Iterator for class `RWBinaryTree`. Traverses the tree from the "smallest" to "largest" element, where "smallest" and "largest" are defined by the virtual function `compareTo()`.

Note – This approach is generally less efficient than using the member function `RWBinaryTree::apply()`.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

```
RWBinaryTreeIterator(const RWBinaryTree&);
```

Constructs an iterator for a `RWBinaryTree`. Immediately after construction, the position of the iterator is undefined until positioned.

Public member operator

```
virtual RWCollectable*    operator()();
```

Redefined from class `RWIterator`. Advances iterator to the next "largest" element and returns a pointer to it. Returns nil when the end of the collection is reached.

Public member functions

```
virtual RWCollectable*     findNext(const RWCollectable*
                            target);
```
Redefined from class `RWIterator`. Moves iterator to the next item which compares equal to the object pointed to by `target` and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

```
virtual void               reset();
```
Redefined from class `RWIterator`. Resets iterator to its state at construction.

```
virtual RWCollectable*     key() const;
```
Redefined from class `RWIterator`. Returns the item at the current iterator position.

RWbistream

```
        RWbistream
          |           \
RWvistream      ios
          |
  RWvios
```

Synopsis

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

RWbistream bstr(cin);      // Construct a RWbistream,
                           // using cin's streambuf
```

Description

Class `RWbistream` specializes the abstract base class `RWvistream` to restore variables stored in binary format by `RWbostream`.

You can think of it as a binary veneer over an associated streambuf. Because the `RWbistream` retains no information about the state of its associated `streambuf`, its use can be freely exchanged with other users of the streambuf (such as an `istream` or `ifstream`).

`RWbistream` can be interrogated as to the stream state using member functions `good(), bad(), eof()`, *etc.*

Example

See `RWbostream` for an example of how the file `"data.dat"` might be created.

```
#include <rw/bstream.h>
#include <rw/fstream.h>
main()
{
 ifstream fstr("data.dat");     // Open an input file
 RWbistream bstr(fstr);         // Construct RWbistream from it
 int i;
 float f;
 double d;
  bstr >> i;          // Restore an int that was stored in binary
  bstr >> f >> d;     // Restore a float & double}
```

Public constructors

`RWbistream(streambuf* s);`
Construct a `RWbistream` from the streambuf `s`.

`RWbistream(istream& str);`
Construct a `RWbistream` using the `streambuf` associated with the `istream` `str`.

Public member functions

```
virtual int            get();
```
Redefined from class `RWvistream`. Get and return the next `char` from the input stream. Returns EOF if end of file is encountered.

```
virtual RWvistream&    get(char& c);
```
Redefined from class `RWvistream`. Get the next `char` and store it in `c`.

```
virtual RWvistream&    get(wchar_t& wc);
```
Redefined from class `RWvistream`. Get the next wide `char` and store it in `wc`.

```
virtual RWvistream&    get(unsigned char& c);
```
Redefined from class `RWvistream`. Get the next `unsigned char` and store it in `c`.

```
virtual RWvistream&    get(char* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of char's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&    get(wchar_t* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of wide char’s and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&    get(double* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of double's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&    get(float* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of float's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&    get(int* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of int's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(long* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of long's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(short* v, size_t N);
```
Redefined from class `RWvistream`. Get a vector of short's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned char* v, size_t
                           N);
```
Redefined from class `RWvistream`. Get a vector of unsigned char's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned short* v, size_t
                           N);
```
Redefined from class `RWvistream`. Get a vector of `unsigned short's` and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned int* v, size_t
                           N);
```
Redefined from class `RWvistream`. Get a vector of `unsigned int's` and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned long* v, size_t
                           N);
```
Redefined from class `RWvistream`. Get a vector of unsigned long's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       getString(char* s, size_t N);
```
Redefined from class `RWvistream`. Restores a character string from the input stream and stores it in the array beginning at `s`. The function stops reading at the end of the string or after `N-1` characters, whichever comes first. If the latter, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte.

`virtual RWvistream&        getString(wchar_t* ws, size_t N);`

Redefined from class `RWvistream`. Restores a wide character string from the input stream and stores it in the array beginning at `ws`. The function stops reading at the end of the string or after `N-1` characters, whichever comes first. If the latter, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte.

`virtual RWvistream&        operator>>(char& c);`

Redefined from class `RWvistream`. Get the next `char` from the input stream and store it in `c`.

`virtual RWvistream&        operator>>(double& d);`

Redefined from class `RWvistream`. Get the next `double` from the input stream and store it in `d`.

`virtual RWvistream&        operator>>(float& f);`

Redefined from class `RWvistream`. Get the next `float` from the input stream and store it in `f`.

`virtual RWvistream&        operator>>(int& i);`

Redefined from class `RWvistream`. Get the next `int` from the input stream and store it in `i`.

`virtual RWvistream&        operator>>(long& l);`

Redefined from class `RWvistream`. Get the next `long` from the input stream and store it in `l`.

`virtual RWvistream&        operator>>(short& s);`

Redefined from class `RWvistream`. Get the next `short` from the input stream and store it in `s`.

`virtual RWvistream&        operator>>(unsigned char& c);`

Redefined from class `RWvistream`. Get the next `unsigned char` from the input stream and store it in `c`.

`virtual RWvistream&        operator>>(unsigned short& s);`

Redefined from class `RWvistream`. Get the next `unsigned short` from the input stream and store it in `s`.

`virtual RWvistream&        operator>>(unsigned int& i);`

Redefined from class `RWvistream`. Get the next `unsigned int` from the input stream and store it in `i`.

`virtual RWvistream&        operator>>(unsigned long& l);`
Redefined from class `RWvistream`. Get the next `unsigned long` from the input stream and store it in `l`.

RWBitVec

Synopsis

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

RWBitVec v;
```

Description

Class `RWBitVec` is a bitvector whose length can be changed at run time. Because this requires an extra level of indirection, this makes it slightly less efficient than classes `RWGBitVec`(*size*) or `RWTBitVec`<*size*>, whose lengths are fixed at compile time.

Example

```
#include <rw/bitvec.h>
#include <rw/rstream.h>
main()
{
  // Allocate a vector with 20 bits, set to TRUE:
  RWBitVec av(20, TRUE);

  av(2) = FALSE;            // Turn bit 2 off
  av.clearBit(7);           // Turn bit 7 off
  av.setBit(2);             // Turn bit 2 back on

  for(int i=11; i<=14; i++) av(i) = FALSE;

  cout << av << endl;       // Print the vector out}
}
```

Program output:

```
[
1 1 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 1 1 1
]
```

Public constructors

`RWBitVec();`
Construct a zero lengthed (null) vector.

`RWBitVec(size_t N);`
Construct a vector with `N` bits. The initial value of the bits is undefined.

`RWBitVec(size_t N, RWBoolean initVal);`
Construct a vector with `N` bits, each set to the Boolean value `initVal`.

`RWBitVec(const RWByte* bp, size_t N);`
Construct a vector with `N` bits, initialized to the data in the array of bytes pointed to by `bp`. This array must be at least long enough to contain `N` bits. The identifier `RWByte` is a typedef for an unsigned char.

`RWBitVec(const RWBitVec& v);`
Copy constructor. Uses value semantics—the constructed vector will be a copy of `v`.

`~RWBitVec();`
The destructor. Releases any allocated memory.

Assignment operators

```
RWBitVec&                  operator=(const RWBitVec& v);
```
Assignment operator. Value semantics are used—self will be a copy of `v`.

```
RWBitVec&                  operator=(RWBoolean b);
```
Assignment operator. Sets every bit in self to the boolean value `b`.

```
RWBitVec&                  operator&=(const RWBitVec& v);
RWBitVec&                  operator^=(const RWBitVec& v);
RWBitVec&                  operator|=(const RWBitVec& v);
```
Logical assignments. Set each element of self to the logical AND, XOR, or OR, respectively, of self and the corresponding bit in `v`. Self and `v` must have the same number of elements (*i.e.*, be conformal) or an exception of type `RWInternalErr` will occur.

Indexing operators

```
RWBitRef                   operator[](size_t i);
```
Returns a reference to bit `i` of self. A helper class, `RWBitRef`, is used. The result can be used as an lvalue. The index `i` must be between 0 and the length of the vector less one. Bounds checking is performed. If the index is out of range, then an exception of type `RWBoundsErr` will occur.

`RWBitRef                  operator()(size_t i);`

Returns a reference to bit `i` of self. A helper class, `RWBitRef`, is used. The result can be used as an lvalue. The index `i` must be between 0 and the length of the vector less one. Bounds checking is performed only if the preprocessor macro `RWBOUNDS_CHECK` has been defined before including the header file `bitvec.h`. If so, and if the index is out of range, then an exception of type `RWBoundsErr` will occur.

`RWBoolean                 operator[](size_t i) const;`

Returns the boolean value of bit `i`. The result cannot be used as an lvalue. The index `i` must be between 0 and the length of the vector less one. Bounds checking is performed. If the index is out of range, then an exception of type `RWBoundsErr` will occur.

`RWBoolean                 operator()(size_t i) const;`

Returns the boolean value of bit `i`. The result cannot be used as an lvalue. The index `i` must be between 0 and the length of the vector less one. Bounds checking is performed only if the preprocessor macro `RWBOUNDS_CHECK` has been defined before including the header file `<rw/bitvec.h>`. If so, and if the index is out of range, then an exception of type `RWBoundsErr` will occur.

Logical operators

```
RWBoolean                 operator==(const RWBitVec& u)
                           const;
```

Returns `TRUE` if self and `v` have the same length and if each bit of self is set to the same value as the corresponding bit in `v`. Otherwise, returns `FALSE`.

```
RWBoolean                 operator!=(const RWBitVec& u)
                           const;
```

Returns `FALSE` if self and `v` have the same length and if each bit of self is set to the same value as the corresponding bit in `v`. Otherwise, returns `TRUE`.

`RWBoolean                 operator==(RWBoolean b) const;`

Returns `TRUE` if every bit of self is set to the boolean value `b`. Otherwise `FALSE`.

`RWBoolean                 operator!=(RWBoolean b) const;`

Returns `FALSE` if every bit of self is set to the boolean value `b`. Otherwise `TRUE`.

Public member functions

```
void                         clearBit(size_t i);
```
Clears (*i.e.*, sets to `FALSE`) the bit with index `i`. The index `i` must be between 0 and the length of the vector less one. No bounds checking is performed. The following are equivalent, although `clearBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
        a(i) = FALSE;
        a.clearBit(i);
```

```
const RWByte*                data() const;
```
Returns a const pointer to the raw data of self. Should be used with care.

```
size_t                       firstFalse() const;
```
Returns the index of the first OFF (False) bit in self. Returns `RW_NPOS` if there is no OFF bit.

```
size_t                       firstTrue() const;
```
Returns the index of the first ON (True) bit in self. Returns `RW_NPOS` if there is no ON bit.

```
unsigned                     hash() const;
```
Returns a value suitable for hashing.

```
RWBoolean                    isEqual(const RWBitVec& v)
                              const;
```
Returns `TRUE` if self and `v` have the same length and if each bit of self is set to the same value as the corresponding bit in `v`. Otherwise, returns `FALSE`.

```
size_t                       length() const;
```
Returns the number of bits in the vector.

```
ostream&                     printOn(ostream& s) const;
```
Print the vector `v` on the output stream `s`. See the example above for a sample of the format.

```
void                         resize(size_t N);
```
Resizes the vector to have length `N`. If this results in a lengthening of the vector, the additional bits will be set to `FALSE`.

```
istream&                     scanFrom(istream&);
```
Read the bit vector from the input stream `s`. The vector will dynamically be resized as necessary. The vector should be in the same format printed by member function `printOn(ostream&)`.

```
void                          setBit(size_t i);// Set bit i
```

Sets (*i.e.*, sets to `TRUE`) the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following are equivalent, although `setBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
      a(i) = TRUE;
      a.setBit(i);
```

```
RWBoolean                     testBit(size_t i) const;
```

Tests the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following are equivalent, although `testBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
      if( a(i) )          doSomething();
      if( a.testBit(i) )  doSomething();
```

Related global functions

```
RWBitVec                      operator!(const RWBitVec& v);
```

Unary operator that returns the logical negation of vector `v`.

```
RWBitVec        operator&(const RWBitVec&,const RWBitVec&);
RWBitVec        operator^(const RWBitVec&,const RWBitVec&);
RWBitVec        operator|(const RWBitVec&,const RWBitVec&);
```

Returns a vector that is the logical AND, XOR, or OR of the vectors `v1` and `v2`. The two vectors must have the same length or an exception of type `RWInternalErr` will occur.

```
ostream&        operator<<(ostream& s, const RWBitVec& v);
```

Calls `v.printOn(s)`.

```
istream&        operator>>(istream& s, RWBitVec& v);
```

Calls `v.scanFrom(s)`.

```
RWvostream&     operator<<(RWvostream&, const
                 RWBitVec& vec);
RWFile&         operator<<(RWFile&,     const
                 RWBitVec& vec);
```

Saves the `RWBitVec` `vec` to a virtual stream or `RWFile`, respectively.

```
RWvistream&      operator>>(RWvistream&,   RWBitVec& vec);
RWFile&          operator>>(RWFile&,       RWBitVec& vec);
```

Restores an `RWBitVec` into `vec` from a virtual stream or `RWFile`, respectively, replacing the previous contents of `vec`.

```
size_t           sum(const RWBitVec& v);
```

Returns the total number of bits set in the vector `v`.

RWbostream

```
       RWbostream
    |              \
RWvostream      ios
    |
RWvios
```

Synopsis

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

// Construct a RWbostream, using cout's streambuf:
RWbostream bstr(cout);
```

Description

Class `RWbostream` specializes the abstract base class `RWvostream` to store variables in binary format. The results can be restored by using its counterpart `RWbistream`.

You can think of it as a binary veneer over an associated `streambuf`. Because the `RWbostream` retains no information about the state of its associated `streambuf`, its use can be freely exchanged with other users of the streambuf (such as `ostream` or `ofstream`).

Note – Variables should not be separated with whitespace. Such whitespace would be interpreted literally and would have to be read back in as a character string.

`RWbostream` can be interrogated as to the stream state using member functions `good(), bad(), eof()`, *etc.*

Example

See RWbistream for an example of how the file "data.dat" might be read back in.

```
#include <rw/bstream.h>
#include <fstream.h>
main()
{
  ofstream fstr("data.dat");  // Open an output file
  RWbostream bstr(fstr);      // Construct an RWbostream from it
  int i = 5;
  float f = 22.1;
  double d = -0.05;

  bstr << i;                  // Store an int in binary
  bstr << f << d;             // Store a float & double}
```

Public constructors

`RWbostream(streambuf* s);`
Construct a RWbostream from the streambuf s.

`RWbostream(ostream& str);`
Construct a RWbostream from the streambuf associated with the output stream str.

Public member functions

`virtual RWvostream&       operator<<(const char* s);`
Redefined from class RWvostream. Store the character string starting at s to the output stream in binary. The character string is expected to be null terminated.

`virtual RWvostream&       operator<<(const wchar_t* ws);`
Redefined from class RWvostream. Store the wide character string starting at ws to the output stream in binary. The character string is expected to be null terminated.

`virtual RWvostream&       operator<<(char c);`
Redefined from class RWvostream. Store the char c to the output stream in binary.

`virtual RWvostream&       operator<<(wchar_t wc);`
Redefined from class RWvostream. Store the wide char wc to the output stream in binary.

`virtual RWvostream&        operator<<(unsigned char c);`
Redefined from class `RWvostream`. Store the unsigned char `c` to the output stream in binary.

`virtual RWvostream&        operator<<(double d);`
Redefined from class `RWvostream`. Store the double `d` to the output stream in binary.

`virtual RWvostream&        operator<<(float f);`
Redefined from class `RWvostream`. Store the float `f` to the output stream in binary

`virtual RWvostream&        operator<<(int i);`
Redefined from class `RWvostream`. Store the int `i` to the output stream in binary.

`virtual RWvostream&        operator<<(unsigned int i);`
Redefined from class `RWvostream`. Store the unsigned int `i` to the output stream in binary.

`virtual RWvostream&        operator<<(long l);`
Redefined from class `RWvostream`. Store the long `l` to the output stream in binary.

`virtual RWvostream&        operator<<(unsigned long l);`
Redefined from class `RWvostream`. Store the unsigned long `l` to the output stream in binary.

`virtual RWvostream&        operator<<(short s);`
Redefined from class `RWvostream`. Store the short `s` to the output stream in binary.

`virtual RWvostream&        operator<<(unsigned short s);`
Redefined from class `RWvostream`. Store the unsigned short `s` to the output stream in binary.

`virtual RWvostream&        put(char c);`
Redefined from class `RWvostream`. Store the `char` `c` to the output stream.

`virtual RWvostream&        put(unsigned char c);`
Redefined from class `RWvostream`. Store the `unsigned char` `c` to the output stream.

```
virtual RWvostream&       put(const char* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of `chars` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const wchar_t* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of wide `chars` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const unsigned char* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of `unsigned chars` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const short* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of `shorts` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const unsigned short* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of `unsigned shorts` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const int* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of `ints` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const unsigned int* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of `unsigned ints` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const long* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of `longs` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const unsigned long* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of `unsigned longs` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const float* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of `floats` starting at `p` to the output stream in binary.

```
virtual RWvostream&       put(const double* p, size_t
                           N);
```

Redefined from class `RWvostream`. Store the vector of `doubles` starting at `p` to the output stream in binary.

RWBTree

RWBTree
|
RWCollection
|
RWCollectable

Synopsis

```
#include <rw/btree.h>
RWBTree a;
```

Description

Class `RWBTree` represents a group of ordered elements, not accessible by an external key. Duplicates are not allowed. An object stored by class `RWBTree` must inherit abstract base class `RWCollectable`—the elements are ordered internally according to the value returned by virtual function `compareTo()` (see class `RWCollectable`).

This class has certain advantages over class `RWBinaryTree`. First, the `B-Tree` is automatically *balanced*. (With class `RWBinaryTree`, you must call member function `balance()` explicitly to balance the tree.) Nodes are never allowed to have less than a certain number of items (called the *order*). The default order is 50, but may be changed by resetting the value of the static constant `"order"` in the header file `<btree.h>` and recompiling. Larger values will result in shallower trees, but less efficient use of memory.

Because many keys are held in a single node, class `RWBTree` also tends to fragment memory less.

Public constructor

```
RWBTree();
```
Construct an empty `B-Tree`.

```
RWBTree(const RWBTree& btr);
```
Construct self as a shallow copy of `btr`.

```
virtual                    ~RWBTree();
```
Redefined from `RWCollection`. Calls `clear()`.

Public member operators

```
void                       operator=(const RWBTree& btr);
```
Set self to a shallow copy of `btr`.

```
RWBoolean                  operator<=(const RWBTree& btr)
                            const;
```
Returns `TRUE` if self is a subset of `btr`. That is, for every item in self, there must be an item in `btr` that compares equal.

```
RWBoolean                  operator==(const RWBTree& btr)
                            const;
```
Returns `TRUE` if self and `btr` are equivalent. That is, they must have the same number of items and for every item in self, there must be an item in `btr` that compares equal.

Public member functions

```
virtual void               apply(RWapplyCollectable ap,
                            void*)
```
Redefined from class `RWCollection` to apply the user-supplied function pointed to by `ap` to each member of the collection, in order, from smallest to largest. This supplied function should not do anything to the items that could change the ordering of the collection.

```
virtual RWspace            binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void               clear();
```
Redefined from class `RWCollection`.

```
virtual void               clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                compareTo(const RWCollectable*
                            a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```

Inherited from class `RWCollection`.

```
virtual size_t             entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```
Redefined from class `RWCollection`. The first item that compares equal to the object pointed to by `target` is returned or nil if no item is found.

```
virtual unsigned           hash() const;
```
Inherited from class `RWCollectable`.

```
unsigned                   height() const;
```
Special member function of this class. Returns the height of the tree, defined as the number of nodes traversed while descending from the root node to an external (leaf) node.

```
virtual RWCollectable*     insert(RWCollectable* c);
```
Redefined from class `RWCollection`. Inserts the item `c` into the collection and returns it. Returns nil if the insertion was unsuccessful. The item `c` is inserted according to the value returned by `compareTo()`.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWBTREE`.

```
virtual RWBoolean          isEmpty() const;
```
Redefined from class `RWCollection`.

```
virtual RWBoolean          isEqual(const RWCollectable* a)
                            const;
```
Redefined to return TRUE is the object pointed to by `a` is of the same type as self, and self == `t`.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```
Redefined from class `RWCollection`. Returns the number of items that compare equal to `target`. Since duplicates are not allowed, this function can only return 0 or 1.

```
virtual RWCollectable*     remove(const RWCollectable*
                            target);
```
Redefined from class `RWCollection`. Removes and returns the first item that compares equal to the object pointed to by `target`. Returns nil if no item was found.

```
virtual void               removeAndDestroy(const
                            RWCollectable* target);
```
Inherited from class `RWCollection`.

```
virtual void              restoreGuts(RWvistream&);
virtual void              restoreGuts(RWFile&);
virtual void              saveGuts(RWvostream&) const;
virtual void              saveGuts(RWFile&) const;
```

Inherited from class `RWCollection`.

RWBTreeDictionary

RWBTreeDictionary
     |
RWBTree
     |
RWCollection
     |
RWCollectable

Synopsis

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

RWBTreeDictionary a;
```

Description

Dictionary class implemented as a `B-Tree`, for the storage and retrieval of key-value pairs. Both the keys and values must inherit abstract base class `RWCollectable`—the elements are ordered internally according to the value returned by virtual function `compareTo()` of the key (see class `RWCollectable`). Duplicate keys are not allowed.

The `B-Tree` is *balanced*. That is, nodes are never allowed to have less than a certain number of items (called the *order*). The default order is 50, but may be changed by resetting the value of the static constant `"order"` in the header file `<btree..h>` and recompiling. Larger values will result in shallower trees, but less efficient use of memory.

Public constructors

`RWBTreeDictionary();`
Constructs an empty `B-Tree` dictionary.

Public member functions

```
void                 applyToKeyAndValue(RWapplyKeyAndValue
                     ap,void*)
```
Applies the user-supplied function pointed to by `ap` to each key-value pair of the collection, in order, from smallest to largest.

```
virtual RWspace              binaryStoreSize() const
```
Inherited from class `RWCollection`.

```
virtual void              clear();
```
Redefined from class `RWCollection`. Removes all key-value pairs from the collection.

```
virtual void              clearAndDestroy();
```
Redefined from class `RWCollection`. Removes all key-value pairs in the collection, and deletes *both* the key and the value.

```
virtual int               compareTo(const RWCollectable*
                           a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean         contains(const RWCollectable*
                           target) const;
```
Inherited from class `RWCollection`.

```
virtual size_t            entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*    find(const RWCollectable* key)
                           const;
```
Redefined from class `RWCollection`. Returns the *key* in the collection which compares equal to the object pointed to by `target`, or nil if no key is found.

```
RWCollectable*            findKeyAndValue(const
                           RWCollectable* target,
                            RWCollectable*& v) const;
```
Returns the key in the collection which compares equal to the object pointed to by `target`, or nil if no key was found. The value is put in `v`. You are responsible for defining `v` before calling this function.

```
RWCollectable*            findValue(const RWCollectable*
                           target) const;
```
Returns the *value* associated with the key which compares equal to the object pointed to by `target`, or nil if no key was found.

```
RWCollectable*            findValue(const RWCollectable*
                           target,RWCollectable*
                            newValue);
```
Returns the *value* associated with the key which compares equal to the object pointed to by `target`, or nil if no key was found. Replaces the value with `newValue` (if a key was found).

`virtual unsigned` `hash() const;`
Inherited from class `RWCollectable`.

`unsigned` `height() const;`
Inherited from class `RWBTree`.

`RWCollectable*` `insertKeyAndValue(RWCollectable* key,RWCollectable* value);`
Adds a key-value pair to the collection and returns the key if successful, nil if the key is already in the collection.

`virtual RWClassID` `isA() const;`
Redefined from class `RWCollectable` to return `__RWBTREEDICTIONARY`.

`virtual RWBoolean` `isEmpty() const;`
Inherited from class `RWBTree`.

`virtual RWBoolean` `isEqual(const RWCollectable* a) const;`
Redefined to return TRUE is the object pointed to by `a` is of the same type as self, and self == `t`.

`virtual size_t` `occurrencesOf(const RWCollectable* target) const;`
Redefined from class `RWCollection`. Returns the number of keys that compare equal with `target`. Because duplicates are not allowed, this function can only return 0 or 1.

`virtual RWCollectable*` `remove(const RWCollectable* target);`
Redefined from class `RWCollection`. Removes the key and value pair for which the key compares equal to the object pointed to by `target`. Returns the key, or nil if no match was found.

`virtual void` `removeAndDestroy(const RWCollectable* target);`
Redefined from class `RWCollection`. Removes *and* deletes the key and value pair for which the key compares equal to the object pointed to by `target`.

Note – Both the key and the value are deleted. Does nothing if the key is not found.

```
RWCollectable*            removeKeyAndValue(const
                           RWCollectable* target,
                            RWCollectable*& v);
```

Removes the key and value pair for which the key compares equal to the object pointed to by `target`. Returns the key, or nil if no match was found. The value is put in `v`. You are responsible for defining `v` before calling this function.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```

Inherited from class `RWCollection`.

RWBTreeOnDisk

Synopsis

```
typedef long RWstoredValue;
typedef int (*RWdiskTreeCompare)(const char*,
                                 const char*,
                                 size_t);

#include <rw/disktree.h>
#include <rw/filemgr.h>
RWFileManager fm(”filename.dat”);
RWBTreeOnDisk bt(fm);
```

Description

Class `RWBTreeOnDisk` represents an ordered collection of associations of keys and values, where the ordering is determined by comparing keys using an external function. The user can set this function. Duplicate keys are not allowed. Given a key, the corresponding value can be found.

This class is specifically designed for managing a B-Tree in a disk file. Keys, defined to be arrays of `chars`, and values, defined by the typedef `RWstoredValue`, are stored and retrieved from a B-Tree. The values can represent offsets to locations in a file where objects are stored.

The key length is set by the constructor. By default, this value is 16 characters. By default, keys are null-terminated. However, the tree can be used with embedded nulls, allowing multibyte and binary data to be used as keys. To do so you must:

- Specify `TRUE` for parameter `ignoreNull` in the constructor (see below);
- Make sure all buffers used for keys are at least as long as the key length (remember, storage and comparison will not stop with a null value);
- Use a comparison function (such as `memcmp()`) that ignores nulls.

This class is meant to be used with class `RWFileManager` which manages the allocation and deallocation of space in a disk file.

When you construct an `RWBTreeOnDisk` you give the location of the root node in the constructor as argument `start`. If this value is `RWNIL` (the default) then the location will be retrieved from the `RWFileManager` using function `start()` (see class `RWFileManager`). You can also use the enumeration `createMode` to set whether to use an existing tree (creating one if one doesn't exist) or to force the creation of a new tree. The location of the resultant root node can be retrieved using member function `rootLocation()`.

More than one B-Tree can exist in a disk file. Each must have its own separate root node. This can be done by constructing more than one `RWBTreeOnDisk`, each with `createMode` set to `create`.

The *order* of the B-Tree can also be set in the constructor. Larger values will result in shallower trees, but less efficient use of disk space. The minimum number of entries in a node can also be set. Smaller values will result in less time spent balancing the tree, but less efficient use of disk space.

Enumeration

`enum styleMode {V6Style, V5Style};`
This enumeration is used by the constructor to allow backwards compatibility with older V5.X style trees, which supported only 16 byte key lengths. It is used only when creating a new tree. If opening a tree for update, its type is determined automatically at runtime.

`V6Style`	Initialize a new tree using V6.X style trees. This is the default.
`V5Style`	Initialize a new tree using V5.X style trees. In this case, the key length is fixed at 16 bytes.

`enum createMode {autoCreate, create};`
This enumeration is used by the constructor to determine whether to force the creation of a new tree.

`autoCreate`	Look in the location given by the constructor argument start for the root node. If valid, use it. Otherwise, allocate a new tree. This is the default.
`create`	Forces the creation of a new tree. The argument `start` is ignored.

Public constructor

```
RWBTreeOnDisk(       RWFileManager& f,
                     unsigned nbuf         = 10,
                     createMode omode      = autoCreate,
                     unsigned keylen       = 16,
                     RWBoolean ignoreNull  = FALSE,
                     RWoffset start        = RWNIL,
                     styleMode smode       = V6Style,
                     unsigned halfOrder    = 10,
                     unsigned minFill      = 10);
```

Construct a B-Tree on disk. The parameters are as follows:

`f`	The file in which the B-Tree is to be managed. This is the only required parameter.
`nbuf`	The maximum number of nodes that can be cached in memory.
`omode`	Determines whether to force the creation of a new tree or whether to attempt to open an existing tree for update (the default).
`keylen`	The length of a key in bytes. Ignored when opening an existing tree.
`ignoreNull`	Controls whether to allow embedded nulls in keys. If `FALSE` (the default), then keys end with a terminating null. If `TRUE`, then all `keylen` bytes are significant. Ignored when opening an existing tree.
`start`	Where to find the root node. If set to `RWNIL` (the default), then uses the value returned by the `RWFileManager`'s `start()` member function. Ignored when creating a new tree.
`smode`	Sets the type of B-Tree to create, allowing backwards compatibility (see above). The default specifies new V6.X style B-Trees. Ignored when opening an existing tree.
`halfOrder`	One half the order of the B-Tree (that is, one half the number of entries in a node). Ignored when opening an existing tree.
`minFill`	The minimum number of entries allowed in a node (must be less than or equal to `halfOrder`). Ignored when opening an existing tree.

Public member functions

```
void                    applyToKeyAndValue(
                           (*ap)(const char*,
                           RWstoredValue), void* x);
```

Visits all items in the collection in order, from smallest to largest, calling the user-provided function pointed to by `ap` with the key and value as arguments. This function should have prototype:

```
void yourApplyFunction( const char* ky,
                        RWstoredValue val,
                        void* x);
```

The function `yourApplyFunction` *cannot* change the key. The value x can be anything and is passed through from the call to `applyToKeyAndValue()`. A possible exception that could occur is `RWFileErr`.

```
void                        clear();
```
Removes all items from the collection. A possible exception that could occur is `RWFileErr`.

```
RWBoolean                   contains(const char* ky) const;
```
Returns `TRUE` if the tree contains a key that is equal to the string pointed to by `ky, FALSE` otherwise. A possible exception that could occur is `RWFileErr`.

```
size_t                      entries();
```
Returns the number of items in the `RWBTreeOnDisk`. A possible exception that could occur is `RWFileErr`.

```
RWstoredValue               findValue(const char* ky);
```
Returns the value for the key that compares equal to the string pointed to by `ky`. Returns `RWNIL` if no key is found. A possible exception that could occur is `RWFileErr`.

```
int                         height();
```
Returns the height of the `RWBTreeOnDisk`. A possible exception that could occur is `RWFileErr`.

```
int                         insertKeyAndValue(const char*
                             ky, RWstoredValue v);
```
Adds a key-value pair to the `B-Tree`. Returns `TRUE` for successful insertion, `FALSE` otherwise. A possible exception that could occur is `RWFileErr`.

```
RWBoolean                   isEmpty() const;
```
Returns `TRUE` if the `RWBTreeOnDisk` is empty, otherwise `FALSE`.

```
void                        remove(const char* ky);
```
Removes the key and value pair that has a key which matches `ky`. A possible exception that could occur is `RWFileErr`.

```
RWoffset                    rootLocation() const;
```
Returns the offset of the root node.

```
RWdiskTreeCompare           setComparison(RWdiskTreeCompare
                             fun);
```
Changes the comparison function to `fun` and returns the old function. This function must have prototype:

```
int yourFun(const char* key1, const char* key2, size_t N);
```

It should return a number less than zero, equal to zero, or greater than zero depending on whether the first argument is less than, equal to or greater than the second argument, respectively. The third argument is the key length. Possible choices (among others) are `strncmp()` (the default), or `strnicmp()` (for case-independent comparisons).

RWBufferedPageHeap

RWBufferedPageHeap
|
RWVirtualPageHeap

Synopsis

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

(*Abstract base class*)

Description

This is an abstract base class that represents an abstract page heap buffered through a set of memory buffers. It inherits from the abstract base class `RWVirtualPageHeap` which represents an abstract page heap.

`RWBufferedPageHeap` will supply and maintain a set of memory buffers. Specializing classes should supply the actual physical mechanism to swap pages in and out of these buffers by supplying definitions for the pure virtual functions `swapIn(RWHandle, void*)` and `swapOut(RWHandle, void*)`.

The specializing class should also supply appropriate definitions for the public functions `allocate()` and `deallocate(RWHandle)`.

For a sample implementation of a specializing class, see class `RWDiskPageHeap`.

Public constructor

```
RWBufferedPageHeap(unsigned pgsize, unsigned nbufs=10);
```

Constructs a buffered page heap with page size `pgsize`. The number of buffers (each of size `pgsize`) that will be allocated on the heap will be `nbufs`. If there is insufficient memory to satisfy the request, then the state of the resultant object as returned by member function `isValid()` will be `FALSE`, otherwise, `TRUE`.

Protected member functions

```
virtual RWBoolean      swapIn(RWHandle h, void* buf)  = 0;
virtual RWBoolean      swapOut(RWHandle, h void* buf) = 0;
```

It is the responsibility of the specializing class to supply definitions for these two pure virtual functions. Function `swapOut()` should copy the page with handle `h` from the buffer pointed to by buf to the swapping medium. Function `swapIn()` should copy the page with handle `h` into the buffer pointed to by `buf`.

Public member functions

```
virtual RWHandle          allocate() = 0;
```

It is the responsibility of the specializing class to supply a definition for this pure virtual function. The specializing class should allocate a page and return a unique handle for it. It should return zero if it cannot satisfy the request. The size of the page is set by the constructor.

```
virtual                   ~RWBufferedPageHeap();
```

Deallocates all internal buffers.

```
RWBoolean                 isValid();
```

Returns `TRUE` if self is in a valid state. A possible reason why the object might not be valid is insufficient memory to allocate the internal buffers.

```
virtual void              deallocate(RWHandle h);
```

Redefined from class `RWVirtualPageHeap`. It is never an error to call this function with argument zero. Even though this is not a pure virtual function, it is the responsibility of the specializing class to supply an appropriate definition for this function. All this definition does is release any buffers associated with the handle `h`. Just as the actual page allocation is done by the specializing class through virtual function `allocate()`, so must the actual deallocation be done by overriding `deallocate()`.

```
virtual void              dirty(RWHandle h);
```

Redefined from class `RWVirtualPageHeap`.

```
virtual void*             lock(RWHandle h);
```

Redefined from class `RWVirtualPageHeap`.

```
virtual void              unlock(RWHandle h);
```

Redefined from class `RWVirtualPageHeap`.

RRWCacheManager

Synopsis

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

RWFile f("file.dat");      // Construct a file
RWCacheManager(&f, 100);   // Cache 100 byte blocks to
                             file.dat
```

Description

Class `RWCacheManager` caches fixed length blocks to and from an associated `RWFile`. The block size can be of any length and is set at construction time. The number of cached blocks can also be set at construction time.

Writes to the file may be deferred. Use member function `flush()` to have any pending writes performed.

Example

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

struct Record {
  int i;
  float f;
  char str[15];
};

main()
{
  RWoffset loc;
  RWFile file("file.dat");  // Construct a file

  // Construct a cache, using 20 slots for struct Record:
  RWCacheManager cache(&file, sizeof(Record), 20);

  Record r;
  // ...
  cache.write(loc, &r);
  // ...
  cache.read(loc, &r);
}
```

Public constructor

`RWCacheManager(RWFile* file, unsigned blocksz, unsigned mxblks = 10);`
Construct a cache for the `RWFile` pointed to by `file`. The length of the fixed-size blocks is given by `blocksz`. The number of cached blocks is given by `mxblks`.

`~RWCacheManager();`
Performs any pending I/O operations (*i.e.*, calls `flush()`) and deallocates any allocated memory.

Public member functions

```
RWBoolean                  flush();
```
Perform any pending I/O operations. Returns `TRUE` if the flush was successful, `FALSE` otherwise.

```
void                       invalidate();
```
Invalidate the cache.

```
RWBoolean                  read(RWoffset locn, void* dat);
```
Return the data located at offset `locn` of the associated `RWFile`. The data is put in the buffer pointed to by `dat`. This buffer must be at least as long as the block size specified when the cache was constructed. Returns `TRUE` if the operation was successful, otherwise `FALSE`.

```
RWBoolean                  write(RWoffset locn, void* dat);
```
Write the block of data pointed to by `dat` to the offset `locn` of the associated `RWFile`. The number of bytes written is given by the block size specified when the cache was constructed. The actual write to disk may be deferred. Use member function `flush()` to perform any pending output. Returns `TRUE` if the operation was successful, otherwise `FALSE`.

22

RWCollectable

Synopsis

```
typedef RWCollectable Object;    // Smalltalk typedef
```

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

Description

Class `RWCollectable` is an abstract base class for collectable objects. This class contains virtual functions for identifying, hashing, comparing, storing and retrieving collectable objects. While these virtual functions have simple default definitions, objects that inherit this base class will typically redefine one or more of them.

Virtual functions

```
virtual                      ~RWCollectable()
```

All functions that inherit class `RWCollectable` have virtual destructors. This allows them to be deleted by such member functions as `removeAndDestroy()` without knowing their type.

```
virtual RWspace              binaryStoreSize() const;
```

Returns the number of bytes used by the virtual function `saveGuts (RWFile&)` to store an object. Typically, this involves adding up the space required to store all primitives, plus the results of calling `recursiveStoreSize()` for all objects inheriting from `RWCollectable`. See Chapter 17, "Persistence" for details.

```
virtual int                  compareTo(const RWCollectable*)
                              const;
```

The function `compareTo()` is necessary to sort the items in a collection. If `p1` and `p2` are pointers to `RWCollectable` objects, the statement

```
  p1-compareTo(p2);
```

should return:

```
             0        if *p1 "is equal to" *p2;
            >0        if *p1 is "larger" than *p2;
            <0        if *p1 is "smaller" than *p2.
```

Note – The meaning of "is equal to", "larger" and "smaller" is left to the user. The default definition provided by the base class is based on the addresses, *i.e.*,
`return this == p2 ? 0 : (this  p2 ? 1 : -1);`
and is probably not very useful.

```
virtual unsigned          hash() const;
```
Returns a hash value. This function is necessary for collection classes that use hash table look-up. The default definition provided by the base class hashes the object's address:

```
  return (unsigned)this;
```

```
virtual RWClassID         isA() const;
```
Returns a class identification number (typedef'd to be an `unsigned short`). The default definition returns `__RWCOLLECTABLE`. Identification numbers greater than or equal to 0x8000 (hex) are reserved for Tools.h++ objects. User defined classes should define `isA()` to return a number between 0 and 0x7FFF.

```
virtual RWBoolean         isEqual(const RWCollectable* t)
                           const;
```
Returns `TRUE` if collectable object "matches" object at address `t`. The default definition is:

```
  return this == t;
```

i.e., both objects have the same address (a test for *identity*). The definition may be redefined in any consistent way.

```
virtual RWCollectable*    newSpecies() const;
```
Allocates a new object off the heap of the same type as self and returns a pointer to it. You are responsible for deleting the object when done with it.

```
virtual void              restoreGuts(RWFile&);
```
Read an object's state from a binary file, using class `RWFile`, replacing the previous state.

```
virtual void              restoreGuts(RWvistream&);
```
Read an object's state from an input stream, replacing the previous state.

```
virtual void              saveGuts(RWFile&) const;
```
Write an object's state to a binary file, using class `RWFile`.

```
virtual void              saveGuts(RWvostream&) const;
```
Write an object's state to an output stream.

```
RWspace                   recursiveStoreSize() const;
```
Returns the number of bytes required to store the object using the global operator
```
                          RWFile& operator<<(RWFile&,
const RWCollectables&);
```
Recursively calls `binaryStoreSize()`, taking duplicate objects into account.

Related global operators

```
RWvostream&        operator<<(RWvostream&, const
                    RWCollectable& obj);
RWFile&            operator<<(RWFile&,      const
                    RWCollectable& obj);
```
Saves the object `obj` to a virtual stream or `RWFile`, respectively. Recursively calls the virtual function `saveGuts()`, taking duplicate objects into account. See Chapter 17, “Persistence,” and “Multiple inheritance” on page 173 in Chapter 20, “Implementation Notes.”

```
RWvistream&        operator>>(RWvistream&,
                    RWCollectable& obj);
RWFile&            operator>>(RWFile&,
                    RWCollectable& obj);
```
Restores an object inheriting from RWCollectable into `obj` from a virtual stream or `RWFile`, respectively, replacing the previous contents of `obj`. Recursively calls the virtual function `restoreGuts()`, taking duplicate objects into account. See Chapter 17, “Persistence” and “Multiple inheritance” on page 173 in Chapter 20, “Implementation Notes,” for a complete description. Various exceptions that could be thrown are `RWInternalErr` (if the `RWFactory` does not know how to make the object), and `RWExternalErr` (corrupted stream or file).

```
RWvistream&        operator>>(RWvistream&,
                    RWCollectable*& obj);
RWFile&            operator>>(RWFile&,
                    RWCollectable*& obj);
```
Looks at the next object on the input stream or `RWFile`, respectively, and either creates a new object of the proper type off the heap and returns a pointer to it, or else returns a pointer to a previously read instance. Recursively calls the virtual function `restoreGuts()`, taking duplicate objects into account. If an object is created off the heap, then you are responsible for deleting it. See Chapter 17, “Persistence” and “Multiple inheritance” on page 173 in

Chapter 20, “Implementation Notes,” for a complete description. Various exceptions that could be thrown are `RWInternalErr` (if the `RWFactory` does not know how to make the object), and `RWExternalErr` (corrupted stream or file).

RWCollectableDate

```
RWCollectableDate
    |             |
RWCollectable   RWDate
```

Synopsis

```
typedef RWCollectableDate Date;    // Smalltalk typedef

#include <rw/colldate.h>

RWCollectableDate  d;
```

Description

Collectable Dates. Inherits classes `RWDate` and `RWCollectable`. This class is useful when dates are used as keys in the "dictionary" collection classes, or if dates are stored and retrieved as `RWCollectables`. The virtual functions of the base class `RWCollectable` have been redefined.

Public constructors

```
RWCollectableDate();
RWCollectableDate(unsigned day, unsigned year);
RWCollectableDate(unsigned day, unsigned month, unsigned
                       year);
RWCollectableDate(unsigned day, const char* mon, unsigned
                       year, const RWLocale& locale =
                       RWLocale::global());
RWCollectableDate(istream& s, const RWLocale& locale =
                       RWLocale::global());
RWCollectableDate(const RWCString& str,
                       const RWLocale& locale =
                       RWLocale::global());
RWCollectableDate(const RWTime& t, const RWZone& zone
                       =RWZone::local());
RWCollectableDate(const struct tm* tmb);
RWCollectableDate(const RWDate& d);
```

Calls the corresponding constructor of the base class `RWDate`.

Public member functions

```
virtual RWspace          binaryStoreSize() const;
```

Redefined from class `RWCollectable`.

```
virtual int                 compareTo(const RWCollectable*
                             c) const;
```
Redefined from class `RWCollectable`. Returns the results of calling `RWDate::compareTo(c)`.

```
virtual unsigned            hash() const;
```
Redefined from class `RWCollectable`. Returns the results of calling `RWDate::hash()`.

```
virtual RWClassID           isA() const;
```
Redefined from class `RWCollectable` to return `__RWCOLLECTABLEDATE`.

```
virtual RWBoolean           isEqual(const RWCollectable* t)
                             const;
```
Redefined from class `RWCollectable`. Returns the results of calling `operator==()` for the base class `RWDate` by using appropriate casts.

```
virtual void                restoreGuts(RWvistream&);
virtual void                restoreGuts(RWFile&);
virtual void                saveGuts(RWvostream&) const;
virtual void                saveGuts(RWFile&) const;
```
Redefined from class `RWCollectable`.

RWCollectableInt

```
        RWCollectableInt
         |            |
RWCollectable     RWInteger
```

Synopsis

```
typedef RWCollectableInt Integer;   // Smalltalk typedef
```

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

```
RWCollectableInt  i;
```

Description

Collectable integers. Inherits classes `RWInteger` and `RWCollectable`. This class is useful when integers are used as keys in the "dictionary" collection classes, or if integers are stored and retrieved as `RWCollectables`. The virtual functions of the base class `RWCollectable` have been redefined.

Public constructors

```
RWCollectableInt();
```

Calls the appropriate base class constructor. See `RWInteger::RWInteger()`.

```
RWCollectableInt(int i);
```

Calls the appropriate base class constructor. See `RWInteger::RWInteger(int)`.

Public member functions

```
virtual RWspace          binaryStoreSize() const;
```

Redefined from class `RWCollectable`.

```
virtual int              compareTo(const RWCollectable*
                          c) const;
```

Redefined from class `RWCollectable`. Returns the difference between self and the `RWCollectableInt` pointed to by `c`.

```
virtual unsigned         hash() const;
```

Redefined from class `RWCollectable`. Returns the `RWCollectableInt`'s value as an unsigned, to be used as a hash value.

```
virtual RWClassID        isA() const;
```

Redefined from class `RWCollectable` to return `__RWCOLLECTABLEINT`.

```
virtual RWBoolean          isEqual(const RWCollectable* c)
                            const;
```
Redefined from class `RWCollectable`. Returns `TRUE` if self has the same value as the `RWCollectableInt` at address `c`.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```
Redefined from class `RWCollectable`.

RWCollectableString

```
   RWCollectableString
         |          |
RWCollectable    RWCString
```

Synopsis

```
typedef RWCollectableString String; // Smalltalk typedef
```

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

```
RWCollectableString  c;
```

Description

Collectable strings. This class is useful when strings are stored and retrieved as `RWCollectables`, or when they are used as keys in the "dictionary" collection classes. Class `RWCollectableString` inherits from both class `RWCString` and class `RWCollectable`. The virtual functions of the base class `RWCollectable` have been redefined.

Public constructors

`RWCollectableString();`
Construct an `RWCollectableString` with zero characters.

`RWCollectableString(const RWCString& s);`
Construct an `RWCollectableString` from the `RWCString s`.

`RWCollectableString(const char* c);`
Conversion from character string.

`RWCollectableString(const RWCSubString&);`
Conversion from sub-string.

`RWCollectableString(char c, size_t N);`
Construct an `RWCollectableString` with `N` characters (default blanks).

Public member functions

```
virtual RWspace          binaryStoreSize() const;
```
Redefined from class `RWCollectable`.

```
virtual int              compareTo(const RWCollectable*
                          c) const;
```
Redefined from class `RWCollectable`. Calls `RWCString::compareTo()` with `c` as the argument and returns the results. This compares strings lexicographically.

```
virtual unsigned         hash() const;
```
Redefined from class `RWCollectable`. Calls `RWCString::hash()` and returns the results.

```
virtual RWClassID        isA() const;
```
Redefined from class `RWCollectable` to return `__RWCOLLECTABLESTRING`.

```
virtual RWBoolean        isEqual(const RWCollectable* c)
                          const;
```
Redefined from class `RWCollectable`. Calls `RWCString::operator==()` (*i.e.*, the equivalence operator) with `c` as the argument and returns the results.

```
virtual void             restoreGuts(RWvistream&);
virtual void             restoreGuts(RWFile&);
virtual void             saveGuts(RWvostream&) const;
virtual void             saveGuts(RWFile&) const;
```
Redefined from class `RWCollectable`.

RWCollectableTime

```
      RWCollectableTime
          |          |
RWCollectable    RWTime
```

Synopsis

```
typedef RWCollectableTime Time;    // Smalltalk typedef
```

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

```
RWCollectableTime  t;
```

Description

Inherits classes `RWTime` and `RWCollectable`. This class is useful when times are used as keys in the "dictionary" collection classes, or if times are stored and retrieved as `RWCollectables`. The virtual functions of the base class `RWCollectable` have been redefined.

Public constructors

```
RWCollectableTime();
RWCollectableTime(unsigned long s);
RWCollectableTime(unsigned hour, unsigned minute, unsigned
                  sec = 0, const RWZone& zone =
                  RWZone::local());
RWCollectableTime(const RWDate& day, unsigned hour=0,
                  unsigned minute=0, unsigned sec = 0,
                  const RWZone& zone = RWZone::local());
RWCollectableTime(const RWDate& day, const RWCString& str,
                  const RWZone& zone = RWZone::local(),
                  const RWLocale& locale =
                  RWLocale::global());
RWCollectableTime(const struct tm* tmb, const RWZone&
                  zone = RWZone::local());
```

Calls the corresponding constructor of `RWTime`.

Public member functions

```
virtual RWspace            binaryStoreSize() const;
```

Redefined from class `RWCollectable`.

```
virtual int                compareTo(const RWCollectable*
                           c) const;
```

Redefined from class `RWCollectable`. Returns the results of calling `RWTime::compareTo(c)`.

```
virtual unsigned           hash() const;
```

Redefined from class `RWCollectable`. Returns the results of calling `RWTime::hash()`.

```
virtual RWClassID          isA() const;
```

Redefined from class `RWCollectable` to return `__RWCOLLECTABLETIME`.

```
virtual RWBoolean          isEqual(const RWCollectable* c)
                           const;
```

Redefined from class `RWCollectable`. Returns the results of calling `operator==()` for the base class `RWTime` by using appropriate casts.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```

Redefined from class `RWCollectable`.

22

RWCollection

RWCollection
```
        |
```
RWCollectable

Synopsis

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

typedef RWCollection Collection;    // Smalltalk typedef
```

Description

Class `RWCollection` is an abstract base class for the `Smalltalk`-like collection classes. The class contains virtual functions for inserting and retrieving pointers to `RWCollectable` objects into the collection classes. Virtual functions are also provided for storing and reading the collections to files and streams. Collections that inherit this base class will typically redefine one or more of these functions.

In the documentation below, pure virtual functions are indicated by "= 0" in their declaration. These functions *must be* defined in derived classes. For these functions the description is intended to be generic—all inheriting collection classes generally follow the described pattern. Exceptions are noted in the documentation for the particular class.

For many other functions, a suitable definition is provided by `RWCollection` and a deriving class may not need to redefine the function. Examples are `contains()` or `restoreGuts()`.

Public member operators

```
void                    operator+=(const RWCollection&);
void                    operator-=(const RWCollection&);
```
Adds or removes, respectively, each item in the argument to or from self.

Public member functions

```
virtual                    ~RWCollection();
```
Null definition (does nothing).

```
virtual void               apply(RWapplyCollectable ap,
                            void*) = 0;
```
This function applies the user-supplied function pointed to by ap to each member of the collection. This function should have prototype

```
  void yourApplyFunction(RWCollectable* c, void*);
```

The function *yourApplyFunction()* can perform any operation on the item at address `c` that *does not change* the ordering of the collection. Client data may be passed to this function by using the second argument.

```
RWBag                      asBag() const;
RWSet                      asSet() const;
RWOrdered                  asOrderedCollection() const;
RWBinaryTree               asSortedCollection() const
```
Allows any collection to be converted to a `RWBag`, `RWSet`, `RWOrdered`, or a `RWBinaryTree`.

```
virtual RWspace            binaryStoreSize() const;
```
Redefined from class `RWCollectable`.

```
virtual void               clear() = 0;
```
Removes all objects from the collection. Does not delete the objects themselves.

```
virtual void               clearAndDestroy();
```
Removes all objects from the collection *and deletes* them. Takes into account duplicate objects within a collection and only deletes them once. However, it does *not* take into account objects shared between different collections. Either do not use this function if you will be sharing objects between separate collections, or put all collections that could be sharing objects into one single "super-collection" and call `clearAndDestroy()` on that.

```
virtual int                compareTo(const RWCollectable*
                            a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```
Returns `TRUE` if the collection contains an item where the virtual function `find()` returns non-nil.

```
virtual size_t            entries() const = 0;
```
Returns the total number of items in the collection.

```
virtual RWCollectable*    find(const RWCollectable*
                           target) const = 0;
```
Returns a pointer to the first item in the collection which "matches" the object pointed to by `target` or nil if no item was found. For most collections, an item "matches" the target if either `isEqual()` or `compareTo()` find equivalence, whichever is appropriate for the actual collection type. However, the "identity collections" (*i.e.*, `RWIdentitySet` and `RWIdentityDictionary`) look for an item with the same address (*i.e.*, "is identical to").

```
virtual unsigned          hash() const;
```
Inherited from class RWCollectable.

```
virtual RWCollectable*    insert(RWCollectable* e) = 0;
```
Adds an item to the collection and returns a pointer to it. If the item is already in the collection, some items return the old instance, others return nil.

```
virtual RWClassID         isA() const;
```
Redefined from class `RWCollectable` to return `__RWCOLLECTION`.

```
virtual RWBoolean         isEmpty() const = 0;
```
Returns `TRUE` if the collection is empty, otherwise returns `FALSE`.

```
virtual RWBoolean         isEqual(const RWCollectable* a)
                           const;
```
Inherited from class `RWCollectable`.

```
virtual size_t            occurrencesOf(const
                           RWCollectable* t) const = 0;
```
Returns the number of items in the collection which are "matches" `t`. See function `find()` for a definition of matches.

```
virtual voidrestoreGuts(RWFile&);
```
Redefined to repeatedly call the global operator
```
    RWvistream& operator>>(RWvistream&, RWCollectable*&);
```
followed by `insert (RWCollectable*)` for each item in the collection.

```
virtual voidrestoreGuts(RWvistream&);
```
Redefined to repeatedly call the global operator
```
    RWvistream& operator>>(RWvistream&, RWCollectable*&);
```
followed by `insert(RWCollectable*)` for each item in the collection.

```
RWCollectable*              remove(const RWCollectable*
                             target) = 0;
```
Removes and returns a pointer to the first item in the collection which "matches" the object pointed to by `target`. Returns nil if no object was found. Does not delete the object.

```
virtual void                removeAndDestroy(const
                             RWCollectable* target);
```
Removes *and deletes* the first item in the collection which "matches" the object pointed to by `target`.

```
RWCollection*               select(testCollectable tst,
                             void* x) const;
```
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 allocated off the heap of the same type as self and returns a pointer to this new collection. Because the new collection is allocated *off the heap*, you are responsible for deleting it when done.

```
virtual void                saveGuts(RWFile&);
```
Redefined to call the global operator
```
  RWFiles& operator<<(RWFile&, const RWCollectables&);
```
for each object in the collection.

```
virtual void                saveGuts(RWvostream&);
  RWvostream& operator<<(RWvostream&, const RWCollectable&);
```
for each object in the collection.

22

RWCRegexp

Synopsis

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

RWCRegexp re(".*\.doc");  // Matches filename with suffix
".doc"
```

Description

Class `RWCRegexp` represents a regular expression. The constructor "compiles" the expression into a form that can be used more efficiently. The results can then be used for string searches using class `RWCString`.

The regular expression (RE) is constucted as follows:

The following rules determine one-character REs that match a *single* character:

1.1 Any character that is not a special character (to be defined) matches itself.

1.2 A backslash (\) followed by any special character matches the literal character itself (*i.e.*, this "escapes" the special character).

1.3 The "special characters" are:

+ * ? . [ ] ^ $

1.4 The period (**.**) matches any character except the newline. For example, ".umpty" matches either "Humpty" or "Dumpty".

1.5 A set of characters enclosed in brackets (**[ ]**) is a one-character RE that matches any of the characters in that set. For example, "[akm]" matches either an "a", "k", or "m". A range of characters can be indicated with a dash. E.g., "[a–z]" matches any lower-case letter. However, if the first character of the set is the caret (^), then the RE matches any character except those in the set. It does not match the empty string. Example: [^akm] matches any character except "a", "k", or "m". The caret loses its special meaning if it is not the first character of the set.

The following rules can be used to build a multicharacter RE.

2.1 A one-character RE followed by an asterisk (*) matches zero or more occurrences of the RE. Hence, [a–z]* matches zero or more lower-case characters.

2.2 A one-character RE followed by a plus (**+**) matches *one* or more occurrences of the RE. Hence, *[a–z]+* matches one or more lower-case characters.

2.3 A question mark (**?**) is an optional element. The preceeding RE can occur zero or once in the string—no more. For example, xy?z matches either *xyz* or *xz*.

2.4 The concatenation of REs is a RE that matches the corresponding concatenation of strings. For example, [A–Z][a–z]* matches any capitalized word.

Finally, the entire regular expression can be anchored to match only the beginning or end of a line:

3.1 If the caret (**^**) is at the beginning of the RE, then the matched string must be at the beginning of a line.

3.2 If the dollar sign (**$**) is at the end of the RE, then the matched string must be at the end of the line.

The following escape codes can be used to match control characters:

\b	backspace
\e	ESC (escape)
\f	formfeed
\n	newline
\r	carriage return
\t	tab
\xddd	the literal hex number 0xddd
\^C	Control code. For example \^D is "control-D"

Example

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

main()
{
  RWCString aString("Hark! Hark! the lark");

  // A regular expression matching any lower-case word
  // starting with "l":
  RWCRegexp reg("l[a-z]*");

  cout << aString(reg) << endl;// Prints "lark"
}
```

Public constructors

`RWCRegexp(RWCS);`
Construct a regular expression from the pattern given by `pat`. The status of the results can be found by using member function `status()`.

`RWCRegexp(const RWCRegexp& r);`
Copy constructor. Uses value semantics—self will be a copy of `r`.

`~RWCRegexp();`
Destructor. Releases any allocated memory.

Assignment operators

`RWCRegexp&                    operator=(const RWCRegexp&);`
Uses value semantics—sets self to a copy of `r`.

`RWCRegexp&                    operator=(const char* pat);`
Recompiles self to the pattern given by `pat`. The status of the results can be found by using member function `status()`.

Public member functions

```
size_t index(const RWCString& str, size_t* len, size_t
start=0) const;
```

Returns the index of the first instance in the string `str` that matches the regular expression compiled in self, or `RW_NPOS` if there is no such match. The search starts at index `start`. The length of the matching pattern is returned in the variable pointed to by `len`. If an invalid regular expression is used for the search, an exception of type `RWInternalErr` will be thrown.

Note – This member function is relatively clumsy to use—class `RWCString` offers a better interface to regular expression searches.

```
statVal                    status();
```

Returns the status of the regular expression:

statVal	**Meaning**
`RWCRegexp::OK`	No errors
`RWCRegexp::ILLEGAL`	Pattern was illegal
`RWCRegexp::TOOLONG`	Pattern exceeded maximum length

22

RWCString

Synopsis

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

RWCString a;
```

Description

`Class RWCString` offers very powerful and convenient facilities for manipulating strings that are just as efficient as the familiar standard C `<string.h>` functions.

Although the class is primarily intended to be used to handle single-byte character sets (SBCS; such as ASCII or ISO Latin-1), with care it can be used to handle multibyte character sets (MBCS). There are two things that must be kept in mind when working with MBCS:

- Because characters can be more than one byte long, the number of bytes in a string can, in general, be greater than the number of characters in the string. Use function `RWCString::length()` to get the number of bytes in a string, function `RWCString::mbLength()` to get the number of characters. Note that the latter is much slower because it must determine the number of bytes in every character. Hence, if the string is known to be nothing but SBCS, then `RWCString::length()` is much to be preferred.
- In general, one or more bytes of a multibyte character can be zero. Hence, MBCS cannot be counted on being null terminated. In practice, it is a rare MBCS that uses embedded nulls (Sun uses EUC encoding, which has no null). Nevertheless, for portability, you should be aware of this and program defensively.

Note – Parameters of type "`const char*`" must not be passed a value of zero. This is detected in the debug version of the library.

The class is implemented using a technique called *copy on write*. With this technique, the copy constructor and assignment operators still reference the old object and hence are very fast. An actual copy is made only when a "write" is performed, that is if the object is about to be changed. The net result is excellent performance, but with easy-to-understand copy semantics.

A separate class `RWCSubString` supports substring extraction and modification operations.

Examples

```
#include <rw/cstring.h>
#include <rw/regexp.h>
#include <rw/rstream.h>
main()
{
  RWCString a("There is no joy in Beantown.");

  RWCRegexp re("[A-Z][a-z]*town");// Any capitalized "town"
  a(re) = "Redmond";
  cout << a << endl;
}
```

Program output:

```
There is no joy in Redmond.
```

Enumerations

`enum RWCString::caseCompare { exact, ignoreCase }`
Used to specify whether comparisons, searches, and hashing functions should use case sensitive `(exact)` or case-insensitive `(ignoreCase)` semantics.

Public constructors

`RWCString();`
Creates a string of length zero (the null string).

`RWCString(const char* cs);`
Conversion from the null-terminated character string `cs`. The created string will copy the data pointed to by `cs`, up to the first terminating null.

`RWCString(const char* cs, size_t N);`
Constructs a string from the character string `cs`. The created string will copy the data pointed to by `cs`. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long.

`RWCString(RWSize_T ic);`
Creates a string of length zero (the null string). The strings *capacity* (that is, the size it can grow to without resizing) is given by the parameter `ic`.

`RWCString(const RWCString& str);`
Copy constructor. The created string will *copy* `str's` data.

`RWCString(const RWCSubString& ss);`
Conversion from sub-string. The created string will *copy* the substring represented by `ss`.

`RWCString(char c);`
Constructs a string containing the single character `c`.

`RWCString(char c, size_t N);`
Constructs a string containing the character `c` repeated `N` times.

Type conversion

`operator                const char*() const;`
Access to the `RWCString`'s data as a null terminated string. This datum is owned by the `RWCString` and may not be deleted or changed. If the `RWCString` object itself changes or goes out of scope, the pointer value previously returned may (will!) become invalid. While the string is null-terminated, note that its *length* is still given by the member function `length()`. That is, it may contain embedded nulls.

Assignment operators

`RWCString&              operator=(const char* cs);`
Assignment operator. Copies the null-terminated character string pointed to by `cs` into self. Returns a reference to self.

`RWCString&              operator=(const RWCString& str);`
Assignment operator. The string will *copy* `str's` data. Returns a reference to self.

`RWCString&              operator+=(const char* cs);`
Append the null-terminated character string pointed to by `cs` to self. Returns a reference to self.

`RWCString&              operator+=(const RWCString& str);`
Append the string `str` to self. Returns a reference to self.

Indexing operators

```
char&                   operator[](size_t i);
char                    operator[](size_t i) const;
```

Return the `i`'th character. The first variant can be used as an lvalue. The index `i` must be between 0 and the length of the string less one. Bounds checking is performed—if the index is out of range then an exception of type `RWBoundsErr` will occur.

```
char&                   operator()(size_t i);
char                    operator()(size_t i) const;
```

Return the i'th character. The first variant can be used as an lvalue. The index `i` must be between 0 and the length of the string less one. Bounds checking is performed if the pre-processor macro `RWBOUNDS_CHECK` has been defined before including `<rw/cstring.h>`. In this case, if the index is out of range, then an exception of type `RWBoundsErr` will occur.

```
RWCSubString            operator()(size_t start,
                         size_t len);
const RWCSubString      operator()(size_t start,
                         size_t len) const;
```

Substring operator. Returns a `RWCSubString` of self with length `len`, starting at index `start`. The first variant can be used as an lvalue. The sum of `start` plus `len` must be less than or equal to the string length. If the library was built using the `RWDEBUG` flag, and `start` and `len` are out of range, then an exception of type `RWBoundsErr` will occur.

```
RWCSubString            operator()(const RWCRegexp& re,
                         size_t start=0);
const RWCSubString      operator()(const RWCRegexp& re,
                         size_t start=0) const;
```

Returns the first substring starting after index `start` that matches the regular expression `re`. If there is no such substring, then the null substring is returned. The first variant can be used as an lvalue.

Public member functions

```
RWCString&              append(const char* cs);
```

Append a copy of the null-terminated character string pointed to by `cs` to self. Returns a reference to self.

```
RWCString&              append(const char* cs, size_t N);
```
Append a copy of the character string `cs` to self. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long. Returns a reference to self.

```
RWCString&              append(const RWCString& cstr);
```
Append a copy of the string `cstr` to self. Returns a reference to self.

```
RWCString&              append(const RWCString& cstr,
                         size_t N);
```
Append the first `N` characters or the length of `cstr` (whichever is less) `of` `cstr` to self. Returns a reference to self.

```
size_t                  binaryStoreSize() const;
```
Returns the number of bytes necessary to store the object using the global function

```
    RWFile& operator<<(RWFile&, const RWCString&);
```

```
size_t                  capacity() const;
```
Return the current capacity of self. This is the number of characters the string can hold without resizing.

```
size_t                  capacity(size_t capac);
```
Hint to the implementation to change the capacity of self to `capac`. Returns the actual capacity.

```
int                     collate(const RWCString& str)
                         const;
int                     collate(const char*      str)
                         const;
```
Returns an int less then, greater than, or equal to zero, according to the result of calling the standard C library function `::strcoll()` on self and the argument `str`. This supports locale-dependent collation.

```
int                     compareTo(const RWCString& str,
                         caseCompare = exact) const;
int                     compareTo(const char*      str,
                         caseCompare = exact) const;
```
Returns an int less than, greater than, or equal to zero, according to the result of calling the standard C library function `memcmp()` on self and the argument `str`. Case sensitivity is according to the caseCompare argument, and may be `RWCString::exact` or `RWCString::ignoreCase`.

```
RWBoolean                contains(const RWCString& cs,
                          caseCompare = exact) const;
RWBoolean                contains(const char* str,
                          caseCompare = exact) const;
```

Pattern matching. Returns `TRUE` if `str` occurs in self. Case sensitivity is according to the `caseCompare` argument, and may be `RWCString::exact` or `RWCString::ignoreCase`.

```
const char*              data() const;
```

Access to the `RWCString`'s data as a null terminated string. This datum is owned by the `RWCString` and may not be deleted or changed. If the `RWCString` object itself changes or goes out of scope, the pointer value previously returned may (will!) become invalid. While the string is null-terminated, note that its length is still given by the member function `length()`. That is, it may contain embedded nulls.

```
size_t                   first(char c) const;
```

Returns the index of the first occurence of the character `c` in self. Returns `RW_NPOS` if there is no such character.

```
unsigned                 hash(caseCompare = exact) const;
```

Returns a suitable hash value.

```
size_t                   index(const char* pat, size_t
                          i=0, caseCompare = exact) const;
size_t                   index(const RWCString& pat, size_t
                          i=0, caseCompare = exact) const;
```

Pattern matching. Starting with index `i`, searches for the first occurrence of `pat` in self and returns the index of the start of the match. Returns `RW_NPOS` if there is no such pattern. Case sensitivity is according to the `caseCompare` argument; it defaults to `RWCString::exact`.

```
size_t                   index(const char* pat, size_t
                          patlen,size_t i,
                          caseCompare) const;
size_t                   index(const RWCString& pat, size_t
                          patlen size_t i,
                          caseCompare) const;
```

Pattern matching. Starting with index `i`, searches for the first occurrence of the first `patlen` characters from `pat` in self and returns the index of the start of the match. Returns `RW_NPOS` if there is no such pattern. Case sensitivity is according to the `caseCompare` argument.

```
size_t                    index(const RWCRegexp& re, size_t
                           i=0) const;
```
Regular expression matching. Returns the index greater than or equal to `i` of the start of the first pattern that matches the regular expression `re`. Returns `RW_NPOS` if there is no such pattern.

```
size_t                    index(const RWCRegexp& re,size_t*
                           ext, size_t i=0) const;
```
Regular expression matching. Returns the index greater than or equal to `i` of the start of the first pattern that matches the regular expression `re`. Returns `RW_NPOS` if there is no such pattern. The length of the matching pattern is returned in the variable pointed to by `ext`.

```
RWCString&                insert(size_t pos, const char* cs);
```
Insert a copy of the null-terminated string `cs` into self at position `pos`, thus expanding the string. Returns a reference to self.

```
RWCString&                insert(size_t pos, const char* cs,
                           size_t N);
```
Insert a copy of the first `N` characters of `cs` into self at position `pos`. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long. Returns a reference to self.

Returns a reference to self.

```
RWCString&                insert(size_t pos, const RWCString&
                           str);
```
Insert a copy of the string `str` into self at position `pos`. Returns a reference to self.

```
RWCString&                insert(size_t pos, const RWCString&
                           str, size_t N);
```
Insert a copy of the first `N` characters or the length of `str` (whichever is less) of str into self at position `pos`. Returns a reference to self.

```
RWBoolean                 isAscii() const;
```
Returns `TRUE` if self contains no characters with the high bit set.

```
RWBoolean                 isNull() const;
```
Returns `TRUE` if this is a zero lengthed string (*i.e.*, the null string).

```
size_t                    last(char c) const;
```
Returns the index of the last occurrence in the string of the character `c`. Returns `RW_NPOS` if there is no such character.

`size_t                  length() const;`

Return the number of bytes in self. Note that if self contains multibyte characters, then this will not be the number of characters.

`size_t                  mbLength() const;`

Return the number of multibyte characters in self, according to the Standard C function `:: mblen()`. Returns `RW_NPOS` if a bad character is encountered. Note that, in general, `mbLength()` $\leq$ `length()`.

`RWCString&              prepend(const char* cs);`

Prepend a copy of the null-terminated character string pointed to by `cs` to self. Returns a reference to self.

`RWCString&              prepend(const char* cs, size_t N,`

Prepend a copy of the character string `cs` to self. Exactly `N` characters are copied, *including any embedded nulls.* Hence, the buffer pointed to by `cs` must be at least `N` bytes long. Returns a reference to self.

`RWCString&              prepend(const RWCString& str);`

Prepends a copy of the string `str` to self. Returns a reference to self.

```
RWCString&              prepend(const RWCString& cstr,
                         size_t N);
```

Prepend the first `N` characters or the length of `cstr` (whichever is less) of `cstr` to self. Returns a reference to self.

`istream&                readFile(istream& s);`

Reads characters from the input stream `s`, replacing the previous contents of self, until EOF is reached. Null characters are treated the same as other characters.

```
istream&                readLine(istream& s, RWBoolean
                         skipWhite = TRUE);
```

Reads characters from the input stream s, replacing the previous contents of self, until a newline (or an `EOF`) is encountered. The newline is removed from the input stream but is not stored. Null characters are treated the same as other characters. If the `skipWhite` argument is `TRUE`, then whitespace is skipped (using the iostream library manipulator `ws`) before saving characters.

`istream&                readString(istream& s);`

Reads characters from the input stream `s`, replacing the previous contents of self, until an `EOF` or null terminator is encountered.

`istream&` `readToDelim(istream& s, char delim='\n');`

Reads characters from the input stream `s`, replacing the previous contents of self, until an `EOF` or the delimiting character `delim` is encountered. The delimiter is removed from the input stream but is not stored. Null characters are treated the same as other characters.

`istream&` `readToken(istream& s);`

Whitespace is skipped before saving characters. Characters are then read from the input stream `s`, replacing previous contents of self, until trailing whitespace or an `EOF` is encountered. The whitespace is left on the input stream. Null characters are treated the same as other characters. Whitespace is identified by the standard C library function `isspace()`.

`RWCString&` `remove(size_t pos);`

Removes the characters from the position `pos` to the end of string. Returns a reference to self.

`RWCString&` `remove(size_t pos, size_t N);`

Removes `N` characters or to the end of string (whichever comes first) starting at the position `pos`. Returns a reference to self.

`RWCString&` `replace(size_t pos, size_t N, const char* cs);`

Replaces `N` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the null-terminated string `cs`. Returns a reference to self.

`RWCString&` `replace(size_t pos, size_t N1, const char* cs, size_t N2);`

Replaces `N1` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the string `cs`. Exactly `N2` characters are copied, *including any embedded nulls.* Hence, the buffer pointed to by `cs` must be at least `N2` bytes long. Returns a reference to self.

`RWCString&` `replace(size_t pos, size_t N, const RWCString& str);`

Replaces `N` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the string `str`. Returns a reference to self.

```
RWCString&               replace(size_t pos, size_t N1,
                          const RWCString& str, size_t N2);
```
Replaces `N1` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the first `N2` characters, or the length of `str` (whichever is less), from `str`. Returns a reference to self.

```
void                     resize(size_t n);
```
Changes the length of self to `n`, adding blanks or truncating as necessary.

```
RWCSubString             strip(stripType s = trailing,
                          char c = ' ');
```
Returns a substring of self where the character `c` has been stripped off the beginning, end, or both ends of the string. The enum `stripType` can take values:

stripType	**Meaning**
`leading`	Remove characters at beginning
`trailing`	Remove characters at end
`both`	Remove characters at both ends

```
RWCSubString             subString(const char* cs, size_t
                          start=0, caseCompare=exact);
const RWCSubString       subString(const char* cs, size_t
                          start=0, caseCompare=exact) const;
```
Returns a substring representing the first occurence of the null-terminated string pointed to by “`cs`”. The first variant can be used as an lvalue. Case sensitivity is according to the `caseCompare` argument; it defaults to `RWCString::exact`.

```
void                     toLower();
```
Changes all upper-case letters in self to lower-case, using the standard C library facilities declared in `<ctype.h>`.

```
void                     toUpper();
```
Changes all lower-case letters in self to upper-case, using the standard C library facilities declared in `<ctype.h>`.

Static public

member functions

```
static size_t          initialCapacity(size_t ic = 15);
```
Sets the minimum initial capacity of an `RWCString`, and returns the old value. The initial setting is 15 characters. Larger values will use more memory, but result in fewer resizes when concatenating or reading strings. Smaller values will waste less memory, but result in more resizes.

```
static size_t          maxWaste(size_t mw = 15);
```
Sets the maximum amount of unused space allowed in a string should it shrink, and returns the old value. The initial setting is 15 characters. If more than `mw` characters are wasted, then excess space will be reclaimed.

```
static size_t          resizeIncrement(size_t ri = 16);
```
Sets the resize increment when more memory is needed to grow a string. Returns the old value. The initial setting is 16 characters.

Note – It is not safe to change `initialCapacity`, `maxWaste`, or `resizeIncrement` when more than one thread is present.

Related global operators

```
RWBoolean              operator==(const RWCString&,
                        const char*      );
RWBoolean              operator==(const char*,
                        const RWCString&);
RWBoolean              operator==(const RWCString&,
                        const RWCString&);
RWBoolean              operator!=(const RWCString&,
                        const char*      );
RWBoolean              operator!=(const char*,
                        const RWCString&);
RWBoolean              operator!=(const RWCString&,
                        const RWCString&);
```
Logical equality and inequality. Case sensitivity is *exact*.

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

Comparisons are done lexicographically, byte by byte. Case sensitivity is *exact*. Use member `collate()` or `strxfrm()` for locale sensitivity.

```
RWCString                 operator+(const RWCString&,
                           const RWCString&);
RWCString                 operator+(const char*,
                           const RWCString&);
RWCString                 operator+(const RWCString&,
                           const char*      );
```

Concatenation operators.

```
ostream&                  operator<<(ostream& s,
                           const RWCString&);
```

Output a `RWCString` on ostream `s`.

```
istream&                  operator>>(istream& s,
                           RWCString& str);
```

Calls `str.readToken(s)`. That is, a token is read from the input stream `s`.

```
RWvostream&               operator<<(RWvostream&, const
                           RWCString& str);
RWFile&                   operator<<(RWFile&,      const
                           RWCString& str);
```

Saves string `str` to a virtual stream or `RWFile`, respectively.

```
RWvistream&              operator>>(RWvistream&,
                          RWCString& str);
RWFile&                  operator>>(RWFile&,
                          RWCString& str);
```

Restores a string into `str` from a virtual stream or `RWFile`, respectively, replacing the previous contents of `str`.

Related global functions

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

Returns the result of applying `::strxfrm()` to the argument string, to allow quicker collation than `RWCString::collate()`.

```
RWCString                toLower(const RWCString& str);
```

Returns a copy of `str` where all upper-case characters have been replaced with lower-case characters. Uses the standard C library function `tolower()`.

```
RWCString                toUpper(const RWCString& str);
```

Returns a copy of `str` where all lower-case characters have been replaced with upper-case characters. Uses the standard C library function `toupper()`.

RWCSubString

Synopsis

```
#include <rw/cstring.h>
RWCString s("test string");
s(6,3);                         // "tri"
```

Description

The class `RWCSubString` allows some subsection of a `RWCString` to be addressed by defining a *starting position* and an *extent*. For example the 7'th through the 11'th elements, inclusive, would have a starting position of 7 and an extent of 5. The specification of a starting position and extent can also be done in your behalf by such functions as `RWCString::strip()` or the overloaded function call operator taking a regular expression as an argument. There are no public constructors—`RWCSubStrings` are constructed by various functions of the `RWCString` class and then destroyed immediately.

A *zero lengthed* substring is one with a defined starting position and an extent of zero. It can be thought of as starting just before the indicated character, but not including it. It can be used as an lvalue. A null substring is also legal and is frequently used to indicate that a requested substring, perhaps through a search, does not exist. A null substring can be detected with member function `isNull()`. However, it cannot be used as an lvalue

Example

```
#include <rw/cstring.h>
#include <rw/rstream.h>
main() {
  RWCString s("What I tell you is true.");
  // Create a substring and use it as an lvalue:
  s(19, 0) = "three times ";
  cout << s << endl;
}
```

Program output:

```
What I tell you is three times true.
```

Assignment operators

```
void                         operator=(const RWCString&);
```

Assignment to a `RWCString`. The statements:

```
RWCString a;
RWCString b;
...
b(2, 3) = a;
```

will copy `a`'s data into the substring `b(2,3)`. The number of elements need not match: if they differ, `b` will be resized appropriately. If self is the null substring, then the statement has no effect.

```
void                         operator=(const char*);
```

Assignment from a character string. Example:

```
    RWCString a("Mary had a lamb");
    char dat[] = "Perrier"; a(11,4) = dat;
    // "Mary had a Perrier"
```

Note – the number of characters selected need not match: if they differ, a will be resized appropriately. If self is the null substring, then the statement has no effect.

Indexing operators

```
char                         operator[](size_t i);
char&                        operator[](size_t i) const;
```

Returns the i'th character of the substring. The first variant can be used as an lvalue, the second cannot. The index i must be between zero and the length of the substring, less one. Bounds checking is performed: if the index is out of range, then an exception of type `RWBoundsErr` will occur.

```
char                         operator()(size_t i);
char&                        operator()(size_t i) const;
```

Returns the i'th character of the substring. The first variant can be used as an lvalue, the second cannot. The index i must be between zero and the length of the substring, less one. Bounds checking is enabled by defining the pre-processor macro `RWBOUNDS_CHECK` before including `<rw/cstring.h>`. In this case, if the index is out of range, then an exception of type `RWBoundsErr` will occur.

Public member functions

```
RWBoolean                    isNull() const;
```

Returns `TRUE` if this is a null substring.

```
size_t                  length() const;
```
Returns the extent (*i.e.*, length) of the `RWCSubString`.

```
RWBoolean               operator!() const;
```
Returns `TRUE` if this is a null substring.

```
size_t                  start() const;
```
Returns the starting element of the `RWCSubString`.

```
void                    toLower();
```
Changes all upper-case letters in self to lower-case. Uses the standard C library function `tolower()`.

```
void                    toUpper();
```
Changes all lower-case letters in self to upper-case. Uses the standard C library function `toupper()`.

Global logical operators

```
RWBoolean               operator==(const RWCSubString&,
                         const RWCSubString&);
RWBoolean               operator==(const RWCString&,
                         const RWCSubString&);
RWBoolean               operator==(const RWCSubString&,
                         const RWCString&    );
RWBoolean               operator==(const char*,
                         const RWCSubString&);
RWBoolean               operator==(const RWCSubString&,
                         const char*         );
```
Returns `TRUE` if the substring is lexicographically equal to the character string or `RWCString` argument. Case sensitivity is *exact*.

```
RWBoolean               operator!=(const RWCString&,
                         const RWCString&    );
RWBoolean               operator!=(const RWCString&,
                         const RWCSubString&);
RWBoolean               operator!=(const RWCSubString&,
                         const RWCString&    );
RWBoolean               operator!=(const char*,
                         const RWCString&    );
RWBoolean               operator!=(const RWCString&,
                         const char*         );
```
Returns the negation of the respective `operator==()`.

RWCTokenizer

Synopsis

```
#include <rw/ctoken.h>
RWCString str("a string of tokens");
RWCTokenizer(str);          // Lex the above string
```

Description

Class `RWCTokenizer` is designed to break a string up into separate tokens, delimited by an arbitrary "white space". It can be thought of as an iterator for strings and as an alternative to the ANSI C function `strtok()` which has the unfortunate side effect of changing the string being tokenized.

Example

```
#include <rw/ctoken.h>
#include <rw/rstream.h>
main()
{
  RWCString a("Something is rotten in the state of Denmark");

  RWCTokenizer next(a);   // Tokenize the string a

  RWCString token;   // Will receive each token

  // Advance until the null string is returned:
  while (!(token=next()).isNull())
    cout << token << "\n";
}
```

Program output:

```
Something
is
rotten
in
the
state
of
Denmark
```

Public constructor

```
RWCTokenizer(const RWCString& s);
```

Construct a tokenizer to lex the string `s`.

Public member function

```
RWCSubString          operator()(const char* s ="
                        \t\n");
```

Advance to the next token and return it as a substring. The token are considered to be deliminated by any of the characters in `s`.

RWDate

Synopsis

```
#include <rw/rwdate.h>
RWDate a;                // Construct today's date
```

Description

Class `RWDate` represents a date, stored as a Julian day number. The member function `isValid()` can be used to determine whether an `RWDate` is a valid date. For example, `isValid()` would return `FALSE` for the date 29 February 1991 because 1991 is not a leap year.

`RWDate`'s can be converted to and from `RWTime`'s, and to and from the Standard C library type `struct tm` defined in `<time.h>`.

Note that because the default constructor for this class creates an instance holding the current 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.

```
class FastDate : public RWDate
{
public:
FastDate() : RWDate(0) {;}
 //Constructs an “invalid” date by default
};
```

Example

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

 main()
{
  // Today's date
  RWDate d;

  // Last Sunday's date:
  RWDate lastSunday = d.previous("Sunday");

  cout << d << endl << lastSunday << endl;
}
```

Program output:

```
03/22/91
03/17/91
```

Public constructors

`RWDate();`
Default constructor. Constructs an `RWDate` with the present date.

`RWDate(const RWDate&);`
Copy constructor.

`RWDate(unsigned day, unsigned year);`
Constructs an `RWDate` with a given day of the year and a given year. The member function `isValid()` can be used to test whether the results are a valid date.

`RWDate(unsigned day, unsigned month, unsigned year);`
Constructs an `RWDate` with the given day of the month, month of the year, and year. Days should be 1-31, months should be 1–12, and the year may be specified as (for example) 1990, or 90. The member function `isValid()` can be used to test whether the results are a valid date.

```
RWDate(unsigned day, const char* mon, unsigned year,
             const RWLocale& locale = RWLocale::global());
```
Constructs an `RWDate` with the given day of the month, month and year. The locale argument is used to convert the month name. Days should be 1-31, months may be specified as (for example): January, JAN, or Jan, and the year may be specified as (for example) 1990, or 90. Leading blanks and case in `mon` are ignored. The member function `isValid()` can be used to test whether the results are a valid date.

```
RWDate(istream& s, const RWLocale& locale =
             RWLocale::global());
```
A full line is read, and converted to a date by the locale argument. The member function `isValid()` must be used to test whether the results are a valid date. Because `RWLocale` cannot rigorously check date input, dates created in this way should also be reconfirmed by the user.

```
RWDate(const RWCString& str, const RWLocale& locale =
             RWLocale::global());
```
The string `str` is converted to a date. The member function `isValid()` must

be used to test whether the results are a valid date. Because `RWLocale` cannot rigorously check date input, dates created in this way should also be reconfirmed by the user.

```
RWDate(const RWTime& t, const RWZone& zone =
             RWZone::local());
```
Constructs an `RWDate` from an `RWTime`. The time zone used defaults to local. The member function `isValid()` must be used to test whether the results are a valid date.

```
RWDate(const struct tm*);
```
Constructs an `RWDate` from the contents of the `struct tm` argument members `tm_year, tm_mon,` and `tm_mday`. Note that the numbering of months and years used in `struct tm` differs from that used for `RWDate` and `RWTime` operations. `struct tm` is declared in the standard include file `<time.h>`.

```
RWDate(unsigned long jd);
```
Construct a date from the Julian Day number `jd`.

Public member operators

```
RWDate&                    operator=(const RWDate&);
```
Assignment operator, generated by the compiler.

```
RWDate                     operator++();
```
Prefix increment operator. Add one day to self, then return the results.

```
RWDate                     operator--();
```
Prefix decrement operator. Subtract one day from self, then return the results.

```
RWDate                     operator++(int);
```
Postfix increment operator. Add one day to self, returning the initial value.

```
RWDate                     operator--(int);
```
Postfix decrement operator. Subtract one day from self, returning the initial value.

```
RWDate&                    operator+=(int s);
```
Add `s` days to self, returning self.

```
RWDate&                    operator-=(int s);
```
Substract `s` days from self, returning self.

Public member functions

```
RWCString                    asString(char format = 'x',
                              const RWLocale& =
                              RWLocale::global()) const;
```

Returns the date as a string, formatted by the `RWLocale` argument. Formats are as defined the standard C library function `strftime()`.

```
RWBoolean                    between(const RWDate& a, const
                              RWDate& b) const;
```

Returns `TRUE` if this `RWDate` is between `a` and `b`, inclusive.

```
size_t                       binaryStoreSize() const;
```

Returns the number of bytes necessary to store the object using the global function

```
    RWFile& operator<<(RWFile&, const RWDate&);
```

```
int                          compareTo(const RWDate* d)
                              const;
```

Compares self to the `RWDate` pointed to by `d` and returns:

0	if self == *d;
1	if self > *d;
–1	if self < *d.

```
unsigned                     day() const;
```

Returns the day of the year (1-366) for this date.

```
unsigned                     dayOfMonth() const;
```

Returns the day of the month (1-31) for this date.

```
void                         extract(struct tm*) const;
```

Returns with the `struct tm` argument filled out completely, with the time members set to 0 and `tm_isdst` set to -1.

Note – the encoding for months and days of the week used in `struct tm` differs from that used elsewhere in `RWDate`. If the date is invalid, all fields are set to -1.

```
unsigned                     firstDayOfMonth() const;
```

Returns the day of the year (1-336) corresponding to the first day of this `RWDate`'s month and year.

```
unsigned                      firstDayOfMonth(unsigned month)
                               const;
```
Returns the day of the year (1-336) corresponding to the first day of the month month (1–12) in this `RWDate`'s year.

```
unsigned                      hash() const;
```
Returns a suitable hashing value.

```
RWBoolean                     isValid() const;
```
Returns `TRUE` if this is a valid date, `FALSE` otherwise.

```
RWBoolean                     leap() const;
```
Returns `TRUE` if the year of this `RWDate` is a leap year.

```
RWDate                        max(const RWDate& t) const;
```
Returns the later date of self or `t`.

```
RWDate                        min(const RWDate& t) const;
```
Returns the earlier date of self or `t`.

```
unsigned                      month() const;
```
Returns the month (1–12) for this date.

```
RWCString                     monthName(const RWLocale& =
                               RWLocale::global()) const;
```
Returns the name of the month for this date, according to the optional `RWLocale` argument.

```
static RWDate                 now();
```
Returns today's date.

```
RWDate                        previous(unsigned dayNum) const;
```
Returns the date of the previous numbered day of the week, where *Monday* = 1, ..., *Sunday* = 7. The variable `dayNum` must be between 1 and 7, inclusive.

```
RWDate                        previous(const char* dayName,
                               const RWLocale& =
                               RWLocale::global()) const;
```
Returns the date of the previous `dayName` (for example, the date of the previous Monday) The weekday name is interpreted according to the `RWLocale` argument.

```
RWCString                   weekDayName(const RWLocale& =
                             RWLocale::global());
```

Returns the name of the day of the week for this date, according to the optional `RWLocale` argument.

```
unsigned                    weekDay() const;
```

Returns the number of the day of the week for this date, where *Monday* = 1, ..., *Sunday* = 7.

```
unsigned                    year() const;
```

Returns the year of this date.

Static member functions

```
static unsigned             dayOfWeek(const char* dayName,
                             const RWLocale& =
                             RWLocale::global());
```

Returns the number of the day of the week corresponding to the given `dayName`. "*Monday*" = 1, ..., "*Sunday*" = 7. Names are interpreted by the `RWLocale` argument. Returns 0 if no match is found.

```
static unsigned             daysInYear(unsigned year);
```

Returns the number of days in a given year.

```
static RWBoolean            dayWithinMonth(unsigned
                             monthNum, unsigned dayNum,
                             unsigned year);
```

Returns `TRUE` if a day (1-31) is within a given month in a given year.

```
static unsigned             indexOfMonth(const char*
                             monthName, const RWLocale& =
                             RWLocale::global());
```

Returns the number of the month (1–12) corresponding to the given `monthName`. Returns 0 for no match.

```
static unsigned long        jday(unsigned mon, unsigned day,
                             unsigned year);
```

Returns the Julian day corresponding to the given month (1–12), day (1-31) and year. Returns zero (0) if the date is invalid.

```
static RWCString          nameOfMonth(unsigned monNum,
                           const RWLocale& =
                           RWLocale::global());
```
Returns the name of month `monNum` (*January* = 1, ..., `December` = 12), formatted for the given locale.

```
static RWBoolean          leapYear(unsigned year);
```
Returns `TRUE` if a given year is a leap year.

```
static RWCString          weekDayName(unsigned dayNum,
                           const RWLocale& =
                           RWLocale::global());
```
Returns the name of the day of the week `dayNum` (*Monday* = 1, ..., *Sunday* = 7), formatted for the given locale

Related global operators

```
RWBoolean                 operator<(const RWDate& d1,
                           const RWDate& d2);
```
Returns `TRUE` if the date `d1` is before `d2`.

```
RWBoolean                 operator<=(const RWDate& d1,
                           const RWDate& d2);
```
Returns `TRUE` if the date `d1` is before or the same as `d2`.

```
RWBoolean                 operator>(const RWDate& d1,
                           const RWDate& d2);
```
Returns `TRUE` if the date `d1` is after `d2`.

```
RWBoolean                 operator>=(const RWDate& d1,
                           const RWDate& d2);
```
Returns `TRUE` if the date `d1` is after or the same as `d2`.

```
RWBoolean                 operator==(const RWDate& d1,
                           const RWDate& d2);
```
Returns `TRUE` if the date `d1` is the same as `t2`.

```
RWBoolean                 operator!=(const RWDate& d1,
                           const RWDate& d2);
```
Returns `TRUE` if the date `d1` is not the same as `d2`.

```
RWDate                    operator+(const RWDate& d,
                           int s);
RWDate                    operator+(int s, const
                           RWDate& d);
```
Returns the date s days in the future from the date `d`.

```
unsigned long             operator-(const RWDate& d1,
                           const RWDate& d2);
```
Returns the number of days between `d1` and `d2`.

```
RWDate                    operator-(const RWDate& d,
                           int s);
```
Returns the date s days in the past from `d`.

```
ostream&                  operator<<(ostream& s, const
                           RWDate& d);
```
Outputs the date `d` on ostream `s`, according to the locale imbued in the stream (see class `RWLocale`), or by `RWLocale::global()` if none.

```
istream&                  operator>>(istream& s,
                           RWDate& t);
```
Reads `t` from `istream s`. One full line is read, and the string contained is converted according to the locale imbued in the stream (see class `RWLocale`), or by `RWLocale::global()` if none. The function `RWDate::isValid()` must be used to test whether the results are a valid date.

```
RWvostream&               operator<<(RWvostream&, const
                           RWDate& date);
RWFile&                   operator<<(RWFile&,     const
                           RWDate& date);
```
Saves the date `date` to a virtual stream or `RWFile`, respectively.

```
RWvistream&               operator>>(RWvistream&,
                           RWDate& date);
RWFile&                   operator>>(RWFile&,     RWDate&
                           date);
```
Restores the date into `date` from a virtual stream or `RWFile`, respectively, replacing the previous contents of `date`.

RWDiskPageHeap

RWDiskPageHeap
        |
RWBufferedPageHeap
        |
RWVirtualPageHeap

Synopsis

```
#include <rw/diskpage.h>
unsigned nbufs;
unsigned pagesize;
RWDiskPageHeap heap("filename", nbufs, pagesize);
```

Description

Class `RWDiskPageHeap` is a specializing type of buffered page heap. It swaps its pages to disk as necessary.

Example

In this example, 100 nodes of a linked list are created and strung together. The list is then walked, confirming that it contains 100 nodes. Each node is a single page. The "pointer" to the next node is actually the handle for the next page.

Code Example 22-2

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

struct Node {
  int   key;
  RWHandlenext;
};

RWHandle head = 0;

const int N = 100;     // Exercise 100 Nodes

main() {

  // Construct a disk-based page heap with page size equal
  // to the size of Node and with 10 buffers:
  RWDiskPageHeap heap(0, 10, sizeof(Node));

  // Build the linked list:
  for (int i=0; i; i++){
```

Code Example 22-2 (Continued)

```
      RWHandle h = heap.allocate();
      Node* newNode = (Node*)heap.lock(h);
      newNode->key  = i;
      newNode->next = head;
      head = h;
      heap.dirty(h);
      heap.unlock(h);
    }


    // Now walk the list:
    unsigned count = 0;
    RWHandle nodeHandle = head;
    while(nodeHandle){
      Node* node = (Node*)heap.lock(nodeHandle);

      RWHandle nextHandle = node->next;
      heap.unlock(nodeHandle);
      heap.deallocate(nodeHandle);
      nodeHandle = nextHandle;
      count++;
    }

    cout << "List with " << count << " nodes walked.\n";
    return 0;
  }
```

Program output:

```
List with 100 nodes walked.
```

Public constructor

```
RWDiskPageHeap(const char* filename=0,
                unsigned nbufs=10,
                unsigned pgsize=512);
```

Constructs a new disk-based page heap. The heap will use a file with filename `filename`, otherwise it will negotiate with the operating system for a temporary filename. The number of buffers, each the size of the page size, will be `nbufs`. No more than this many pages can be locked at any one time. The size of each page is given by `pgsize`. To see whether a valid `RWDiskPageHeap` has been constructed, call member function `isValid()`.

```
virtual                  ~RWDiskPageHeap();
```

Returns any resources used by the disk page heap back to the operating system. All pages should have been deallocated before the destructor is called.

Public member functions

```
virtual RWHandle         allocate();
```

Redefined from class `RWVirtualPageHeap`. Allocates a page off the disk page heap and returns a handle for it. If there is no more space (for example, the disk is full) then returns zero.

```
virtual void             deallocate(RWHandle h);
```

Redefined from class `RWBufferedPageHeap`. Deallocate the page associated with handle `h`. It is not an error to deallocate a zero handle.

```
virtual void             dirty(RWHandle h);
```

Inherited from `RWBufferedPageHeap`.

```
RWBoolean                isValid() const;
```

Returns `TRUE` if this is a valid `RWDiskPageHeap`.

```
virtual void*            lock(RWHandle h);
```

Inherited from `RWBufferedPageHeap`.

```
virtual void             unlock(RWHandle h);
```

Inherited from `RWBufferedPageHeap`.

RWDlistCollectables

```
     RWDlistCollectables
         |              |
RWSequenceable     RWDlist
         |              |
RWCollection       RWSlist
         |
RWCollectable
```

Synopsis

```
#include <rw/dlistcol.h>
RWDlistCollectables a;
```

Description

Class `RWDlistCollectables` represents a group of ordered items, not accessible by an external key. Duplicates are allowed. The ordering of elements is determined externally, generally by the order of insertion and removal. An object stored by `RWDlistCollectables` must inherit abstract base class `RWCollectable`.

Class `RWDlistCollectables` is implemented as a doubly-linked list, which allows for efficient insertion and removal, as well as for movement in either direction.

Public constructors

```
RWDlistCollectables();
```
Constructs an empty doubly-linked list.

```
RWDlistCollectables (const RWCollectable* a);
```
Constructs a linked-list with a single item `a`.

Public member operators

```
RWBoolean                    operator==(const
                               RWDlistCollectables& d) const;
```
Returns `TRUE` if self and `d` have the same number of items and if for every item in self, the corresponding item in the same position in `d` isEqual to it.

Public member functions

```
virtual Collectable*       append(RWCollectable*);
```
Redefined from `RWSequenceable`. Inserts the item at the end of the collection and returns it. Returns nil if the insertion was unsuccesful.

```
virtual void               apply(RWapplyCollectable ap,
                            void*)
```
Redefined from class `RWCollection` to apply the user-supplied function pointed to by `ap` to each member of the collection, in order, from first to last.

```
virtual RWCollectable*&     at(size_t i);
virtual const RWCollectable*at(size_t i) const;
```
Redefined from class `RWSequenceable`. The index must be between zero and the number of items in the collection less one, or an exception of type `RWBoundsErr` will occur.

Note – For a linked-list, these functions must traverse all the links, making them not particularly efficient.

```
virtual RWspace            binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void               clear();
```
Redefined from class `RWCollection`.

```
virtual void               clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                compareTo(const RWCollectable*
                            a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```
Inherited from class `RWCollection`.

```
RWBoolean                  containsReference(const
                            RWCollectable* e) const;
```
Returns true if the list contains an item that *is identical to* the item pointed to by `e` (that is, that has the address `e`).

```
virtual size_t             entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```
Redefined from class `RWCollection`. The first item that *isEqual to* the item pointed to by `target` is returned, or nil if no item is found.

```
RWCollectable*             findReference(const
                            RWCollectable* e) const;
```
Returns the first item that *is identical to* the item pointed to by `e` (that is, that has the address `e`), or nil if none is found.

```
virtual RWCollectable*     first() const;
```
Redefined from class `RWSequenceable`. Returns the item at the beginning of the list.

```
RWCollectable*             get();
```
Returns and *removes* the item at the beginning of the list.

```
virtual unsigned           hash() const;
```
Inherited from class `RWCollectable`.

```
virtual size_t             index(const RWCollectable* c)
                            const;
```
Redefined from class `RWSequenceable`. Returns the index of the first item that *isEqual to* the item pointed to by `c`, or `RW_NPOS` if there is no such index.

```
virtual RWCollectable*     insert(RWCollectable* c);
```
Redefined from class `RWCollection`. Adds the item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
void                       insertAt(size_t indx,
                            RWCollectable* e);
```
Redefined from class `RWSequenceable`. Adds a new item to the collection at position `indx`. The item previously at position `i` is moved to `i+1`, *etc*. The index `indx` must be between 0 and the number of items in the collection, or an exception of type `RWBoundsErr` will occur.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWDLISTCOLLECTABLES`.

```
virtual RWBoolean          isEmpty() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*     last() const;
```
Redefined from class `RWSequenceable`. Returns the item at the end of the list.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```
Redefined from class `RWCollection`. Returns the number of items that *isEqual to* the item pointed to by `target`.

```
size_t                     occurrencesOfReference(const
                            RWCollectable* e) const;
```
Returns the number of items that *are identical to* the item pointed to by `e` (that is, that have the address `e`).

```
virtual RWCollectable*     prepend(RWCollectable*);
```
Redefined from class `RWSequenceable`. Adds the item to the beginning of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
virtual RWCollectable*     remove(const RWCollectable*
                            target);
```
Redefined from class `RWCollection`. Removes and returns the first item that *isEqual to* the item pointed to by `target`. Returns nil if there is no such item.

```
virtual void               removeAndDestroy(const
                            RWCollectable* target);
```
Inherited from class `RWCollection`.

```
RWCollectable*             removeReference(const
                            RWCollectable* e);
```
Removes and returns the first item that *is identical to* the item pointed to by `e` (that is, that has the address `e`). Returns nil if there is no such item.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```
Inherited from class `RWCollection`.

RWDlistCollectablesIterator

```
RWDlistCollectablesIterator
     |                    |
RWIterator       RWDlistIterator
                          |
                 RWSlistIterator
```

Synopsis

```
#include <rw/dlistcol.h>
RWDlistCollectables d;
RWDlistCollectablesIterator it(d);
```

Description

Iterator for class `RWDlistCollectables`. Traverses the linked-list from the first (head) to the last (tail) item. Functions are provided for moving in *either* direction.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWDlistCollectablesIterator (RWDlistCollectables& d);`
Construct a `RWDlistCollectablesIterator` from a `RWDlistCollectables`. Immediately after construction, the position of the iterator is undefined.

Public member operators

`virtual RWCollectable*     operator()();`
Redefined from class `RWIterator`. Advances the iterator to the next item and returns it. Returns nil when the end of the list is reached.

`void                       operator++();`
Advances the iterator one item.

`void                       operator--();`
Moves the iterator back one item.

`void                       operator+=(size_t n);`
Advances the iterator `n` items.

`void                    operator-=(size_t n);`

Moves the iterator back `n` items.

Public member functions

`RWBoolean               atFirst() const;`

Returns `TRUE` if the iterator is at the beginning of the list, otherwise `FALSE`;

`RWBoolean               atLast() const;`

Returns `TRUE` if the iterator is at the end of the list, otherwise `FALSE`;

```
virtual RWCollectable*  findNext(const RWCollectable*
                         target);
```

Redefined from class `RWIterator`. Moves iterator to the next item which *isEqual to* the item pointed to by `target` and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

```
RWCollectable*          findNextReference(const
                         RWCollectable* e);
```

Moves iterator to the next item which *is identical to* the item pointed to by `e` (that is, that has address `e`) and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

```
RWCollectable*          insertAfterPoint(RWCollectable*
                         a);
```

Insert item `a` after the current cursor position and return the item. The cursor's position will be unchanged.

`virtual RWCollectable*  key() const;`

Redefined from class `RWIterator`. Returns the item at the current iterator position.

`RWCollectable*          remove();`

Removes and returns the item at the current cursor position. Afterwards, the iterator will be positioned at the previous item in the list.

```
RWCollectable*              removeNext(const RWCollectable*
                             target);
```

Moves iterator to the next item in the list which *isEqual to* the item pointed to by target, removes it from the list and returns it. Afterwards, the iterator will be positioned at the previous item in the list. If no item is found, returns nil and the position of the iterator will be undefined.

```
RWCollectable*              removeNextReference(const
                             RWCollectable* e);
```

Moves iterator to the next item in the list which *is identical to* the item pointed to by `e` (that is, that has address `e`), removes it from the list and returns it. Afterwards, the iterator will be positioned at the previous item in the list. If no item is found, returns nil and the position of the iterator will be undefined.

```
virtual void                reset();
```

Redefined from class `RWIterator`. Resets the iterator. Afterwards, the position of the iterator will be undefined.

```
void                        toFirst();
```

Moves the iterator to the beginning of the list.

```
void                        toLast();
```

Moves the iterator to the end of the list.

RWFactory

Synopsis

```
typedef unsigned short   RWClassID;
typedef RWCollectable*   (*sRWuserCreator)();
#include <rw/factory.h>

RWFactory* theFactory;
```

Description

Class `RWFactory` can create an instance of a `RWCollectable` object, given a class ID. It does this by maintaining a table of class ID's and associated "creator function". A creator function has prototype:

`RWCollectable*` *aCreatorFunction*`();`

This function should create an instance of a particular class. For a given `RWClassID` tag, the appropriate function is selected, invoked and the resultant pointer returned. Because any object created this way is created off the heap, you are responsible for deleting it when done.

There is a one-of-a-kind global `RWFactory` pointed to by the pointer `theFactory`. It is guaranteed to have creator functions in it for all of the classes referenced by your program. See “An aside: the RWFactory” on page 142 in Chapter 18, “Designing an RWCollectable Class,” for more information.

Example

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

main()
{
  // Create a new RWBag off the heap, using the Class ID __RWBAG.
  // "theFactory" points to the predefined global factory:

  RWBag* b = (RWBag*)theFactory-create(__RWBAG);

  b->insert( new (CollectableDate );// Insert today’s date
  // ...
  b->clearAndDestroy();// Cleanup: first delete members,
  delete b;// then the bag itself
}
```

Public constructors

```
RWFactory();
```

Construct an `RWFactory`.

Public member functions

```
void                          addFunction(RWuserCreator uc,
                               RWClassID id);
```

Adds to the `RWFactory` the global function pointed to by `uc`, which creates an instance of an object with `RWClassID` `id`.

```
RWCollectable*                create(RWClassID id) const;
```

Allocates a new instance of the class with `RWClassID` `id` off the heap and returns a pointer to it. Returns nil if `id` does not exist. Because this instance is allocated *off the heap*, you are responsible for deleting it when done.

```
RWuserCreator                 getFunction(RWClassID id) const;
```

Returns from the `RWFactory` a pointer to the global function associated with `RWClassID` `id`. Returns nil if `id` does not exist.

```
void                          removeFunction(RWClassID id);
```

Removes from the `RWFactory` the global function associated with `RWClassID` `id`. If `id` does not exist in the factory, no action is taken.

22

RWFile

Synopsis

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

RWFile f("filename");
```

Description

Class `RWFile` encapsulates binary file operations using the Standard C stream library (functions `fopen(), fread(), fwrite(),` etc.). This class is based on class `PFile` of the *Interviews Class Library* (1987, Stanford University). The member function names begin with upper case letters in order to maintain compatibility with class `PFile`.

Because this class is intended to encapsulate *binary* operations, it is important that it be opened using a binary mode. This is particularly important under MS-DOS—otherwise bytes that happen to match a newline will be expanded to (carriage return, line feed).

Public constructors

`RWFile(const char* filename, const char* mode = 0);`
Construct an `RWFile` to be used with the file of name `filename` and with mode `mode`. The mode is as given by the Standard C library function `fopen()`. If `mode` is zero (the default) then the constructor will attempt to open an existing file with the given filename for update (mode "`rb+`"). If this is not possible, then it will attempt to create a new file with the given filename (mode "`wb+`"). The resultant object should be checked for validity using function `isValid()`.

`~RWFile();`
Performs any pending I/O operations and closes the file.

Public member functions

`long                    CurOffset();`
Returns the current position, in bytes from the start of the file, of the file pointer.

`RWBoolean               Eof();`
Returns `TRUE` if an end-of-file has been encountered.

`RWBoolean               Erase();`
Erases the contents but does not close the file. Returns `TRUE` if the operation was successful.

```
RWBoolean                  Error();
```
Returns TRUE if a file I/O error has occurred.

```
RWBoolean                  Exists();
```
Returns TRUE if the file exists and has read/write permission.

```
RWBoolean                  Flush();
```
Perform any pending I/O operations. Returns TRUE if successful.

```
const char*                GetName();
```
Returns the file name.

```
RWBoolean                  IsEmpty();
```
Returns TRUE if the file contains no data, FALSE otherwise.

```
RWBoolean                  isValid() const
```
Returns TRUE if the file was successfully opened, FALSE otherwise.

```
RWBoolean                  Read(char& c);
RWBoolean                  Read(wchar_t& wc);
RWBoolean                  Read(short& i);
RWBoolean                  Read(int& i);
RWBoolean                  Read(long& i);
RWBoolean                  Read(unsigned char& c);
RWBoolean                  Read(unsigned short& i);
RWBoolean                  Read(unsigned int& i);
RWBoolean                  Read(unsigned long& i);
RWBoolean                  Read(float& f);
RWBoolean                  Read(double& d);
```
Reads the indicated built-in type. Returns TRUE if the read is successful.

```
RWBoolean                  Read(char* i, size_t count);
RWBoolean                  Read(wchar_t* i, size_t count);
RWBoolean                  Read(short* i, size_t count);
RWBoolean                  Read(int* i, size_t count);
RWBoolean                  Read(long* i, size_t count);
RWBoolean                  Read(unsigned char* i, size_t
                            count);
RWBoolean                  Read(unsigned int* i, size_t
                            count);
RWBoolean                  Read(float* i, size_t count);
RWBoolean                    Read(double* i, size_t count);
```
Reads count instances of the indicated built-in type into a block pointed to by `i`. Returns `TRUE` if the read is successful.

Note – You are responsible for declaring `i` and for allocating the necessary storage before calling this function.

```
RWBoolean                    Read(char* string);
```
Reads a character string, including the terminating null character, into a block pointed to by `string`. Returns `TRUE` if the read is successful.

Note – You are responsible for declaring string and for allocating the necessary storage before calling this function.

Caution – Beware of overflow when using this function.

```
RWBoolean                    SeekTo(long offset);
```
Repositions the file pointer to `offset` bytes from the start of the file. Returns `TRUE` if the operation is successful.

```
RWBoolean                    SeekToBegin();
```
Repositions the file pointer to the start of the file. Returns `TRUE` if the operation is successful.

```
RWBoolean                    SeekToEnd();
```
Repositions the file pointer to the end of the file. Returns `TRUE` if the operation is successful.

```
RWBoolean                    Write(char i);
RWBoolean                    Write(wchar_t i);
RWBoolean                    Write(short i);
RWBoolean                    Write(int i);
RWBoolean                    Write(long i);
RWBoolean                    Write(unsigned char i);
RWBoolean                    Write(unsigned short i);
RWBoolean                    Write(unsigned int i);
RWBoolean                    Write(unsigned long i);
RWBoolean                   Write(float f);
RWBoolean                   Write(double d);
```
Writes the appropriate built-in type. Returns `TRUE` if the write is successful.

```
RWBoolean                   Write(const char* i, size_t
                             count);
RWBoolean                   Write(const wchar_t* i, size_t
                             count);
RWBoolean                   Write(const short* i, size_t
                             count);
RWBoolean                   Write(const int* i, size_t
                             count);
RWBoolean                   Write(const long* i, size_t
                             count);
RWBoolean                   Write(const unsigned char* i,
                             size_t count);
RWBoolean                   Write(const unsigned int* i,
                             size_t  count);
RWBoolean                   Write(const float* i, size_t
                             count);
RWBoolean                   Write(const double* i, size_t
                             count);
```
Writes `count` instances of the indicated built-in type from a block pointed to by `i`. Returns `TRUE` if the write is successful.

```
RWBoolean                   Write(const char* string);
```
Writes a character string, *including the terminating null character*, from a block pointed to by `string`. Returns `TRUE` if the write is successful.

Caution – Beware of non-terminated strings when using this function.

Static public member functions

```
static RWBoolean            Exists(const char* filename);
```
Returns `TRUE` if an `RWFile` with name `filename` exists, with read / write permission.

RWFileManager

RWFileManager
|
RWFile

Synopsis

```
typedef long              RWoffset;
typedef unsigned long     RWspace;   // (typically)

#include <rw/filemgr.h>
RWFileManager f("file.dat");
```

Description

Class `RWFileManager` allocates and deallocates storage in a disk file, much like a “freestore” manager. It does this by maintaining a linked list of free space within the file.

Note – Class `RWFileManager` inherits class `RWFile` as a public base class, hence all the public member functions of `RWFile` are visible to `RWFileManager`. They are not listed here.

If a file is managed by an `RWFileManager` then the results are undefined to read or write to an unallocated space in the file.

Public constructor

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

Construct a `RWFileManager` for the file with path name `filename`. The `RWFileManager` can be constructed with an old file or it can create a new file. If it is constructed with an old file, then the file must have been originally created by a `RWFileManager`. A possible exception that could occur is `RWFileErr`. The resulting object should be checked for validity using function `isValid()`.

Public member functions

```
RWoffset                 allocate(RWspace s);
```

Allocates `s` bytes of storage in the file. Returns the offset to the start of the storage location. The very first allocation for the file is considered "special" and can be returned at any later time by the function `start()`. A possible exception that could occur is `RWFileErr`.

```
void                            deallocate(RWoffset t);
```

Deallocates (frees) the storage space starting at offset `t`. This space must have been previously allocated by a call to `allocate()`. The very first allocation ever made in the file is considered "special" and cannot be deallocated. A possible exception that could occur is `RWFileErr`.

```
RWoffset                        endData();
```

Returns the offset of the last space allocated on this file. If no space has every been allocated, returns `RWNIL`.

```
RWoffset                        start();
```

Returns the offset of the first space ever allocated for this file. If no space has every been allocated, returns `RWNIL`. This is typically used to "get started" and find the rest of the data in the file.

RWHashDictionary

RWHashDictionary
|
RWSet
|
RWHashTable
|
RWCollection
|
RWCollectable

Synopsis

```
typedef RWHashDictionary Dictionary;  // Smalltalk typedef.
```

```
#include <rw/hashdict.h>
RWHashDictionary  a;
```

Description

A `RWHashDictionary` represents a group of unordered values, accessible by external keys. Duplicate keys are not allowed. Class `RWHashDictionary` is implemented as a hash table of associations of keys and values. Both the key and the value must inherit from abstract the base class `RWCollectable`, with a suitable definition of the virtual function `hash()` and `isEqual()` for the key.

This class corresponds to the Smalltalk class `Dictionary`.

Public constructors

`RWHashDictionary(size_t n = RWDEFAULT_CAPACITY);`
Construct an empty hashed dictionary using `n` hashing buckets.

`RWHashDictionary(const RWHashDictionary& hd);`
Copy constructor. A shallow copy of the collection `hd` is made.

Public member operators

`void                  operator=(const RWHashDictionary& hd);`

Assignment operator. A shallow copy of the collection `hd` is made.

`RWBoolean             operator<=(const RWHashDictionary& hd) const;`

Returns `TRUE` if for every key-value pair in self, there is a corresponding key in `hd` that `isEqual`. Their corresponding values must also be equal.

`RWBoolean             operator==(const RWHashDictionary& hd) const;`

Returns `TRUE` if self and `hd` have the same number of entries and if for every key-value pair in self, there is a corresponding key in `hd` that `isEqual`. Their corresponding values must also be equal.

`void                  applyToKeyAndValue(RWapplyKeyAndValue ap, void*)`

Applies the user-supplied function pointed to by `ap` to each key-value pair of the collection. Items are not visited in any particular order.

`virtual RWspace       binaryStoreSize() const;`

Inherited from class `RWCollection`.

`virtual void          clear();`

Redefined from class `RWCollection`. Removes all key-value pairs in the collection.

`virtual void          clearAndDestroy();`

Redefined from class `RWCollection`. Removes all key-value pairs in the collection, and deletes the key *and* the value.

`virtual int           compareTo(const RWCollectable* a) const;`

Inherited from class `RWCollectable`.

`virtual RWBoolean     contains(const RWCollectable* target) const;`

Inherited from class `RWCollection`.

`virtual size_t        entries() const;`

Inherited from class `RWSet`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```
Redefined from class `RWCollection`. Returns the *key* which *isEqual to* the object pointed to by `target`, or nil if no key was found.

```
RWCollectable*             findKeyAndValue(const
                            RWCollectable* target,
                             RWCollectable*& v) const;
```
Returns the key which *isEqual to* the item pointed to by `target`, or nil if no key was found. The value is put in `v`. You are responsible for defining `v` before calling this function.

```
RWCollectable*             findValue(const RWCollectable*
                            target) const;
```
Returns the *value* associated with the key which *isEqual to* the item pointed to by `target`, or nil if no key was found.

```
RWCollectable*             findValue(const RWCollectable*
                            target, RWCollectable*
                             newValue);
```
Returns the *value* associated with the key which *isEqual to* the item pointed to by `target`, or nil if no key was found. Replaces the value with `newValue` (if a key was found).

```
virtual unsigned           hash() const;
```
Inherited from class `RWCollectable`.

```
RWCollectable*            insertKeyAndValue(RWCollectable*
                           key,RWCollectable* value);
```
Adds a key-value pair to the collection and returns the key if successful, nil if the key is already in the collection.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWHASHDICTIONARY`.

```
virtual RWBoolean          isEmpty() const;
```
Inherited from class `RWSet`.

```
virtual RWBoolean          isEqual(const RWCollectable* a)
                            const;
```
Redefined to return TRUE is the object pointed to by `a` is of the same type as self, and self == `t`.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```
Inherited from class `RWSet`. Returns the number of keys which *isEqual to* the item pointed to by `target`. Because duplicates are not allowed, this function can only return 0 or 1.

```
virtual RWCollectable*     remove(const RWCollectable*
                            target);
```
Redefined from class `RWCollection`. Removes the key and value pair where the key *isEqual* to the item pointed to by `target`. Returns the key, or nil if no match was found.

```
virtual void               removeAndDestroy(const
                            RWCollectable* target);
```
Redefined from class `RWCollection`. Removes *and* deletes the key and value pair where the key *isEqual to* the item pointed to by `target`.

Note – Both the key and the value are deleted. Does nothing if the key is not found.

```
RWCollectable*             removeKeyAndValue(const
                            RWCollectable* target,
                             RWCollectable*& v);
```
Removes the key and value pair where the key *isEqual to* the item pointed to by `target`. Returns the key, or nil if no match was found. The value is put in `v`. You are responsible for defining `v` before calling this function.

```
void                       resize(size_t n = 0);
```
Inherited from class `RWSet`.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```
Inherited from class `RWCollection`.

RWHashDictionaryIterator

```
RWHashDictionaryIterator
    |
RWSetIterator
    |
RWIterator
```

Synopsis

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

RWHashDictionary          hd;
RWHashDictionaryIterator     iter(hd);
```

Description

Iterator for class `RWHashDictionary`, allowing sequential access to all the elements of `RWHashDictionary`. Since `RWHashDictionary` is unordered, elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWHashDictionaryIterator(RWHashDictionary&);`
Construct an iterator for a `RWHashDictionary` collection. Immediately after construction, the position of the iterator is undefined until positioned.

Public member operator

`virtual RWCollectable*     operator()();`
Redefined from class `RWIterator`. Advances the iterator to the next key-value pair and returns the key. Returns nil if the cursor is at the end of the collection. Use member function `value()` to recover the value.

Public member functions

```
virtual RWCollectable*     findNext(const RWCollectable*
                            target);
```
Redefined from class `RWIterator`. Moves the iterator to the next key-value pair where the key *isEqual to* the object pointed to by `target`. Returns the key or nil if no key was found.

```
virtual RWCollectable*     key() const;
```
Redefined from class `RWIterator`. Returns the key at the current iterator position.

```
RWCollectable*             remove();
```
Removes the key-value pair at the current iterator position. Returns the key, or nil if there was no key-value pair.

```
RWCollectable*             removeNext(const RWCollectable*
                            target);
```
Moves the iterator to the next key-value pair where the key isEqual to the object pointed to by `target`. Removes the key-value pair, returning the key or nil if there was no match.

```
virtual void               reset();
```
Redefined from class `RWIterator`. Inherited from class `RWSetIterator`. Resets the iterator to its initial state.

```
RWCollectable*             value() const;
```
Returns the value at the current iterator position.

```
RWCollectable*             value(RWCollectable* newValue)
                            const;
```
Replaces the value at the current iterator position and returns the old value.

RWHashTable

RWHashTable
|
RWCollection
|
RWCollectable

Synopsis

```
#include <rw/hashtab.h>
RWHashTable h;
```

Description

This class is a simple hash table for objects inheriting from `RWCollectable`. It uses chaining (as implemented by class `RWSlistCollectables`) to resolve hash collisions. Duplicate objects are allowed.

An object stored by `RWHashTable` must inherit from the abstract base class `RWCollectable`, with suitable definition for virtual functions `hash()` and `isEqual()` (see class `RWCollectable`).

To find an object that matches a key, the key's virtual function `hash()` is first called to determine in which bucket the object occurs. The bucket is then searched linearly by calling the virtual function `isEqual()` for each candidate, with the key as the argument. The first object to return `TRUE` is the returned object.

The initial number of buckets in the table is set by the constructor. There is a default value. If the number of items in the collection greatly exceeds the number of buckets then efficiency will sag because each bucket must be searched linearly. The number of buckets can be changed by calling member function `resize()`. This will require that all objects be rehashed.

The iterator for this class is `RWHashTableIterator`.

Example

```
#include <rw/hashtab.h>
#include <rw/colldate.h>
#include <rw/rstream.h>

main()
{
  RWHashTable table;
  table.insert(new Date(7, "July", 1990));
  table.insert(new Date(1, "May", 1977));
  table.insert(new Date(22, "Feb", 1983));
  table.insert(new Date(2, "Aug", 1966));

  cout << "Table contains " << table.entries() << " entries.\n";
  Date key(22, "Feb", 1983);
  cout << "It does ";
  if (!table.contains(&key)) cout << "not ";
  cout << "contain the key " << key << endl;
  return 0;
}
```

Program output:

```
Table contains 4 entries.
It does contain the key February 22, 1983
```

Public constructors

`RWHashTable(size_t N = RWCollection::DEFAULT_CAPACITY);`
Construct an empty hash table with `N` buckets.

`RWHashTable(const RWHashTable& t);`
Copy constructor. Create a new hash table as a shallow copy of the table `t`. The new table will have the same number of buckets as the old table. Hence, the members need not be and will not be rehashed.

Public operators

`void` `operator=(const RWHashTable& t);`
Assignment operator. Sets self as a shallow copy of `t`. Afterwards, the two tables will have the same number of buckets. Hence, the members need not be and will not be rehashed.

`RWBoolean` `operator==(const RWHashTable& t) const;`
Returns `TRUE` if self and `t` have the same number of elements and if for every key in self there is a corresponding key in `t` which isEqual.

`RWBoolean` `operator<=(const RWHashTable& t) const;`
Returns `TRUE` if self is a subset of `t`, that is, every element of self has a counterpart in `t` which isEqual.

`RWBoolean` `operator!=(const RWHashTable&) const;`
Returns the negation of `operator==()`, above.

Member functions

`virtual void` `apply(RWapplyCollectable ap, void*);`
Redefined from RWCollection. The function pointed to by `ap` will be called for each member in the collection. Because of the nature of hashing collections, this will not be done in any particular order. The function should not do anything that could change the hash value or equality properties of the objects.

`virtual RWspace` `binaryStoreSize() const;`
Inherited from `RWCollection`.

`virtual void` `clear();`
Redefined from `RWCollection`.

`virtual void` `clearAndDestroy();`
Inherited from `RWCollection`.

`virtual int` `compareTo(const RWCollectable*) const;`
Inherited from `RWCollection`.

`virtual RWBoolean` `contains(const RWCollectable*) const;`
Inherited from `RWCollection`.

```
virtual size_t              entries() const
```
Redefined from `RWCollection`.

```
virtual RWCollectable*      find(const RWCollectable*)
                            const;
```
Redefined from `RWCollection`.

```
virtual unsigned            hash() const;
```
Inherited from `RWCollection`.

```
virtual RWCollectable*      insert(RWCollectable* a);
```
Redefined from `RWCollection`. Returns `a` if successful, nil otherwise.

```
virtual RWClassID           isA() const;
```
Redefined from `RWCollection` to return `__RWHASHTABLE`.

```
virtual RWBoolean           isEmpty() const;
```
Redefined from `RWCollection`.

```
virtual RWBoolean           isEqual(const RWCollectable*)
                             const;
```
Redefined from `RWCollection`.

```
virtual RWCollectable*      newSpecies() const;
```
Redefined from `RWCollection`.

```
virtual size_t              occurrencesOf(const
                             RWCollectable*) const;
```
Redefined from `RWCollection`.

```
virtual RWCollectable*      remove(const RWCollectable*);
```
Redefined from `RWCollection`.

```
virtual void                removeAndDestroy(const
                             RWCollectable*);
```
Inherited from `RWCollection`.

```
virtual void                resize(size_t n = 0);
```
Resizes the internal hash table to have `n` buckets. This will require rehashing all the members of the collection. If `n` is zero, then an appropriate size will be picked automatically.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```

Inherited from class `RWCollection`.

RWHashTableIterator

RWHashTableIterator
|
RWIterator

Synopsis

```
#include <rw/hashtab.h>
RWHashTable h;
RWHashTableIterator it(h);
```

Description

Iterator for class `RWHashTable`, which allows sequential access to all the elements of `RWHashTable`.

Note – Because a `RWHashTable` is unordered, elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

```
RWHashTableIterator(RWHashTable&);
```
Construct an iterator for an `RWHashTable`. After construction, the position of the iterator is undefined.

Public member operator

```
virtual RWCollectable*     operator()();
```
Redefined from class `RWIterator`. Advances the iterator to the next item and returns it. Returns nil when the end of the collection is reached.

Public member functions

```
virtual RWCollectable*     findNext(const RWCollectable*
                           target);
```

Redefined from class `RWIterator`. Moves iterator to the next item which *isEqual to* the item pointed to by `target` and returns it.

```
virtual RWCollectable*     key() const;
```

Redefined from class `RWIterator`. Returns the item at the current iterator position.

```
RWCollectable*             remove();
```

Remove the item at the current iterator position from the collection.

```
RWCollectable*             removeNext(const
                            RWCollectable*);
```

Moves the iterator to the next item which *isEqual to* the item pointed to by `target`, removes it from the collection and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

```
virtual void               reset();
```

Redefined from class `RWIterator`. Resets the iterator to its starting state.

RWIdentityDictionary

```
RWIdentityDictionary
    |
RWHashDictionary
    |
RWSet
    |
RWCollection
    |
RWCollectable
```

Synopsis

```
#include <rw/idendict.h>
typedef RWIdentityDictionary IdentityDictionary;
 // Smalltalk typedef

RWIdentityDictionary a;
```

Description

The class `RWIdentityDictionary` is implemented as a hash table, for the storage and retrieval of key-value pairs. Class `RWIdentityDictionary` is similar to class `RWHashDictionary` except that items are found by requiring that they be *identical* (*i.e.*, have the same address) as the key, rather than being equal (*i.e.*, test true for `isEqual()`).

Both keys and values must inherit from the abstract base class `RWCollectable`.

The iterator for this class is `RWHashDictionaryIterator`.

Public constructor

`RWIdentityDictionary(size_t n = RWDEFAULT_CAPACITY);`
Construct an empty identity dictionary with n hashing buckets.

Public member functions

The user interface to this class is identical to class `RWHashDictionary` and is not reproduced here. The only difference between the classes is that keys are found on the basis of *identity* rather than *equality*, and that the virtual function `isA()` returns `__RWIDENTITYDICTIONARY`, the `ClassId` for `RWIdentityDictionary`.

RWIdentitySet

RWIdentitySet
    |
RWSet
    |
RWCollection
    |
RWCollectable

Synopsis

```
#include <rw/idenset.h>
typedef RWIdentitySet IdentitySet; // Smalltalk typedef
RWIdentitySet a;
```

Description

The class `RWIdentitySet` is similar to class `RWSet` except that items are found by requiring that they be *identical* (*i.e.*, have the same address) as the key, rather than being equal (*i.e.*, test true for `isEqual()`).

The iterator for this class is `RWSetIterator`.

Public constructor

```
RWIdentitySet(size_t n = RWDEFAULT_CAPACITY);
```
Construct an empty identity set with n hashing buckets.

Public member functions

The user interface to this class is identical to class `RWSet` and is not reproduced here. The only difference between the classes is that keys are found on the basis of *identity* rather than *equality*, and that the virtual function `isA()` returns `__RWIDENTITYSET`, the `ClassId` for `RWIdentitySet`.

RWInteger

Synopsis

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

RWInteger i;
```

Description

Integer class. This class is useful as a base class for classes that use integers as keys in dictionaries, *etc*.

Public constructors

`RWInteger();`
Construct a `RWInteger` with value zero (0).

`RWInteger(int i);`
Construct a `RWInteger` with value `i`. Serves as a type conversion from int.

Type conversion

```
operator                    int();
```
Type conversion to int.

Public member functions

```
RWspace                     binaryStoreSize() const;
```
Returns the number of bytes necessary to store the object using the global function
```
     RWFiles& operator<<(RWFile&, const RWInteger&)
```

```
int                         value() const;
```
Returns the value of the `RWInteger`.

```
int                         value(int newval);
```
Changes the value of the `RWInteger` to `newval` and returns the old value.

Related Global Operators

```
ostream&                    operator<<(ostream& o, const
                             RWInteger& x);
```
Output `x` to ostream `o`.

```
istream&                    operator>>(istream& i,
                             RWInteger& x);
```
Input `x` from istream `i`.

```
RWvostream&                  operator<<(RWvostream&, const
                              RWInteger& x);
RWFile&                      operator<<(RWFile&,      const
                              RWInteger& x);
```

Saves the `RWInteger x` to a virtual stream or `RWFile`, respectively.

```
RWvistream&                  operator>>(RWvistream&,
                              RWInteger& x);
RWFile&                      operator>>(RWFile&,
                              RWInteger& x);
```

Restores an `RWInteger` into `x` from a virtual stream or `RWFile`, respectively, replacing the previous contents of `x`.

RWIterator

Synopsis

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

typedef RWIterator Iterator;// "Smalltalk" typedef
```

Description

Class `RWIterator` is an abstract base class for iterators used by the Smalltalk-like collection classes. The class contains virtual functions for positioning and resetting the iterator. They are all *pure virtual* functions, meaning that deriving classes must supply a definition. The descriptions below are intended to be generic—all inheriting iterators generally follow the described pattern.

Public virtual functions

```
virtual RWCollectable*     findNext(const RWCollectable*
                            target) = 0;
```

Moves the iterator forward to the next item which "matches" the object pointed to by `target` and returns it or nil if no item was found. For most collections, an item "matches" the target if either `isEqual()` or `compareTo()` find equivalence, whichever is appropriate for the actual collection type. However, when an iterator is used with an "identity collection" (*i.e.*, `RWIdentitySet` and `RWIdentityDictionary`), it looks for an item with the same address (*i.e.*, "is identical to").

```
virtual RWCollectable*     key() const = 0;
```

Returns the item at the current iterator position.

```
virtual RWCollectable*     operator()() = 0;
```

Advances the iterator and returns the next item, or nil if the end of the collection has been reached.

```
virtual void               reset() = 0;
```

Resets the iterator to the state it had immediately after construction.

22

RWLocale

Synopsis

```
#include <locale.h>
```

#include <rw/locale.h>

(Abstract base class)

Description

`RWLocale` is an abstract base class. It defines an interface for formatting dates (including day and month names), times, numbers (including digit grouping), and currency, to and from strings.

Note that because it is an *abstract* base class, there is no way to actually enforce these goals—the description here is merely the model of how a class derived from `RWLocale` should act.

There are three ways to use an `RWLocale` object:

- By passing it to functions which expect one, such as `RWDate::asString()`.
- By specifying a "global" locale by using the static member function `RWLocale::global()`. This locale is passed as the default argument to functions that use a locale.
- By "imbuing" a stream with one so that when an `RWDate` or `RWTime` is written to a stream using `operator<<()`, the appropriate formatting will be used automatically.

Two implementations of `RWLocale` are provided with the library: RWLocaleSnapshot and `RWLocaleDefault`.

- Class `RWLocaleSnapshot` encapsulates the Standard C library locale facility, with two additional advantages: more than one locale can be active at the same time; and it supports conversions *from* strings to other types.
- Class `RWLocaleDefault` implements only the "C" locale. It is small and cheap to construct: one is constructed automatically at program startup to be used as the default value of `RWLocale::global()`.

Enumeration

```
enum CurrSymbol { NONE, LOCAL, INTL };
```

Controls whether no currency symbol, the local currency symbol, or the international currency symbol should be used to format currency.

Public member functions

```
virtual RWCString   asString(         long) const = 0;
virtual RWCString   asString(unsigned long) const = 0;
```

Converts the number to a string (*e.g.*, `"3,456"`).

```
virtual RWCString   asString(double f, int precision = 6,
                           RWBoolean showpoint = 0) const = 0;
```

Converts the double f to a string. The variable `precision` is the number of digits to place after the decimal separator. If `showpoint` is `TRUE`, the decimal separator will appear regardless of the precision.

```
virtual RWCString   asString(struct tm* tmbuf,
                    char format, const RWZone& zone) const = 0;
```

Converts components of the struct tm object to a string, according to the format character. The meanings assigned to the format character are identical to those used in the Standard C Library function `strftime()`. The members of struct tm are assumed to be set consistently. See Table 22-2 for a summary of `strftime()` formatting characters.

```
virtual int         monthIndex(const RWCString&) const = 0;
```

Interprets the `RWCString` as a month, and returns an integer in the range of 1 to 12. Returns 0 if there is an error.

```
virtual int         weekdayIndex(const RWCString&) const=0;
```

Interprets the `RWCString` as a weekday, and returns an integer in the range of 1 to 7, with Monday corresponding to 1. Returns 0 if there is an error.

```
virtual RWCString   moneyAsString(double value,
                           enum CurrSymbol = LOCAL) const = 0;
```

Returns a string containing the `value` argument formatted according to monetary conventions for the locale. The `value` argument is assumed to contain an integer representing the number of units of currency (*e.g.*, `moneyAsString(1000., RWLocale::LOCAL)` in a US locale would yield `"$10.00"`). The `CurrSymbol` argument determines whether the local (*e.g.*, `"$"`) or international (*e.g.*, `"USD "`) currency symbol is applied, or none.

```
virtual RWBoolean stringToNum(const RWCString&, double* fp)
const = 0;
```

Interprets the `RWCString` argument as a floating point number. Spaces are allowed before and after the (optional) sign, and at the end. Digit group separators are allowed in the integer portion. Returns `TRUE` for a valid number, `FALSE` for an error. If it returns `FALSE`, the `double*` argument is untouched. All valid numeric strings are accepted; all others are rejected. The following are examples of valid numeric strings in an English-speaking locale:

```
"1"              " -02. "        ".3"
"1234.56"        "1e10"          "+ 19,876.2E+20"
```

`virtual RWBooleanstringToNum(const RWCString&, long* ip) const = 0;`

Interprets the `RWCString` argument as an integer. Spaces are allowed before and after the (optional) sign, and at the end. Digit group separators are allowed. Returns `TRUE` for a valid integer, `FALSE` for an error. If it returns `FALSE`, the `long*` argument is untouched. All valid numeric strings are accepted; all others are rejected. The following are examples of valid integral strings in an English-speaking locale:

```
“1”              “ -02. “        “+ 1,234”
“1234545”        “1,234,567”
```

Table 22-2 lists formatting characters used by `strftime()`. Examples are given (in parenthesis). For those formats that do not use all members of the struct tm, only those members that are actually used are noted [in brackets].

Table 22-2

Format character	Meaning	Example
`a`	Abbreviated weekday name [from `tm::tm_wday`]	`Sun`
`A`	Full weekday name [from tm::`tm_wday`]	`Sunday`
`b`	Abbreviated month name	`Feb`
`B`	Full month name	`February`
`c`	Date and time [may use all members]	`Feb 29 14:34:56 1984`
`C`	Long date and time. Available only in implementations where “%C” is permitted as a `strftime()` format argument.	`Sunday, February 29 14:34:56 1984`
`d`	Day of the month	`29`
`H`	Hour of the 24-hour day	`14`
`I`	Hour of the 12-hour day	`02`
`j`	Day of the year, from 001 [from `tm::tm_yday`]	`60`
`m`	Month of the year, from 01	`02`
`M`	Minutes after the hour	`34`

Table 22-2 (Continued)

Format character	Meaning	Example
`p`	AM/PM indicator, if any	`AM`
`S`	Seconds after the minute	`56`
`U`	Sunday week of the year, from 00 [from `tm::tm_yday` and `tm::tm_wday`]	
`w`	Day of the week, with 0 for Sunday	`0`
`W`	Monday week of the year, from 00 [from `tm::tm_yday` and `tm::tm_wday`]	
`x`	Date [uses `tm::tm_yday` in some locales]	`Feb 29 1984`
`X`	Time	`14:34:56`
`y`	Year of the century, from 00	`84`
`Y`	Year	`1984`
`Z`	Time zone name [from `tm::tm_isdst`]	`PST` or `PDT`

`virtual RWBoolean stringToDate(const RWCString&, struct tm*) const = 0;`
Interprets the `RWCString` as a date, and extracts the month, day, and year components to the `tm` argument. It returns `TRUE` for a valid date, `FALSE` otherwise. If it returns `FALSE`, the `struct tm` argument is untouched; otherwise it sets the `tm_mday`, `tm_mon`, and `tm_year` members. If the date is entered as three numbers, the order expected is the same as that produced by `strftime()`.

Note – This function cannot reject all invalid date strings.

The following are examples of valid date strings in an English-speaking locale:

```
      "Jan 9,        62" "1/9/62"   "January 9 1962"
      "09Jan62"      "010962"
```

`virtual RWBooleanstringToTime(const RWCString&, struct tm*) const = 0;`
Interprets the `RWCString` argument as a time, with hour, minute, and optional

second. If the hour is in the range [1..12], the local equivalent of “AM” or “PM” is allowed. Returns TRUE for a valid time string, FALSE for an error. If it returns FALSE, the tm argument is untouched; otherwise it sets the tm_hour, tm_min, and tm_sec members. Note that this function cannot reject all invalid time strings. The following are examples of valid time strings in an English-speaking locale:

```
      "1:10 AM"      "13:45:30"     "12.30.45pm"
      "PM 3:15"      "1430"
```

```
virtual RWBoolean    stringToMoney(const RWCString&,
                      double*, RWLocale::CurrSymbol=LOCAL)
                      const = 0;
```

Interprets the `RWCString` argument as a monetary value. The currency symbol, if any, is ignored. Negative values may be specified by the negation symbol or by enclosing parentheses. Digit group separators are optional; if present they are checked. Returns TRUE for a valid monetary value, FALSE for an error. If it returns FALSE, the `double*` argument is untouched; otherwise it is set to the integral number of monetary units entered (e.g. cents, in a U.S. locale).

```
const RWLocale*      imbue(ios& stream) const;
```

Installs self in the stream argument, for later use by the operators << and >> (e.g. in `RWDate` or `RWTime`). The pointer may be retrieved from the stream with the static member `RWLocale::of()`. In this way a locale may be passed transparently through many levels of control to be available where needed, without intruding elsewhere.

Static member functions

```
static const RWLocale& of(ios&);
```

Returns the locale installed in the stream argument by a previous call to `RWLocale::imbue()` or, if no locale was installed, the result from `RWLocale::global()`.

```
static const RWLocale* global(const RWLocale* loc);
```

Sets the global “default” locale object to `loc`, returning the old object. This object is used by `RWDate` and `RWTime` string conversion functions as a default locale. It is set initially to refer to an instance of `RWLocaleDefault`.

```
static const RWLocale& global();
```

Returns a reference to the present global “default” locale.

RWLocaleSnapshot

RWLocaleSnapshot
    |
RWLocale

Synopsis

```
#include <locale.h>

#include /locale.h

RWLocaleSnapshot ourLocale("");  // encapsulate user's
formats
```

Description The class `RWLocaleSnapshot` implements the `RWLocale` interface using Standard C library facilities. To use it, the program creates an `RWLocaleSnapshot` instance. The constructor of the instance queries the program's environment (using standard C library functions such as `localeconv(), strftime()`, and, if available , vendor specific library functions) to learn everything it can about formatting conventions in effect at the moment of instantiation. When done, the locale can then be switched and another instance of `RWLocaleSnapshot` created. By creating multiple instances of `RWLocaleSnapshot`, your program can have more than one locale active at the same time, something that is difficult to do with the Standard C library facilities.

Note – `RWLocaleSnapshot` does not encapsulate character set, collation, or message information.

Class `RWLocaleSnapshot` has a set of public data members initialized by its constructor with information extracted from its execution environment.

Example

Try this program with the environmental variable `LANG` set to various locales:

Code Example 22-3

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

main()
{
    RWLocale::global(new RWLocaleSnapshot(""));

    // Print a number using the global locale:
```

Code Example 22-3 (Continued)

```
    cout << locale.asString(1234567.6543) << endl;

    // Now get and print a date:
    cout << "enter a date: " << flush;
    RWDate date;
    cin >> date;
    if (date.isValid())
       cout << date << endl;
    else
       cout << "bad date" << endl;
    return 0; }
}
```

Enumerations

`enum RWDateOrder { DMY, MDY, YDM, YMD };`

Public constructor

`RWLocaleSnapshot(const char* localeName = 0);`

Constructs an `RWLocale` object by extracting formats from the global locale environment. It uses the Standard C Library function `setlocale()` to set the named locale, and then restores the previous global locale after formats have been extracted. If `localeName` is 0, it simply uses the current locale. The most useful locale name is the empty string, "", which is a synonym for the user's chosen locale (usually specified by the environment variable `LANG`).

Public member functions

```
virtual RWCString    asString(           long) const;
virtual RWCString    asString(unsigned long) const;
virtual RWCString    asString(double f, int precision = 6,
                        RWBoolean showpoint = 0) const;
virtual RWCString    asString(struct tm* tmbuf,
                        char format,
                           const RWZone& zone) const;
virtual RWCString    moneyAsString(double value,
                        enum CurrSymbol = LOCAL) const;

virtual RWBoolean    stringToNum  (const RWCString&,
                                   double* fp) const;
virtual RWBoolean    stringToNum  (const RWCString&,
                                   long* ip   ) const;
virtual RWBoolean    stringToDate (const RWCString&,
                                   struct tm*) const;
virtual RWBoolean    stringToTime (const RWCString&, struct
                                   tm*) const;
```

```
virtual RWBoolean    stringToMoney(const RWCString&,
                      double*       ,
                      RWLocale::CurrSymbol=LOCAL) const;
```

Redefined from class `RWLocale`. These virtual functions follow the interface described under class `RWLocale`. They generally work by converting values to and from strings using the rules specified by the `struct lconv` values (see `<locale.h>`) encapsulated in self.

Public data members

```
RWCString      decimal_point_;
RWCString      thousands_sep_;
RWCString      grouping_;
RWCString      int_curr_symbol_;
RWCString      currency_symbol_;
RWCString      mon_decimal_point_;
RWCString      mon_thousands_sep_;
RWCString      mon_grouping_;
RWCString      positive_sign_;
RWCString      negative_sign_;
char           int_frac_digits_;
char           frac_digits_;
char           p_cs_precedes_;
char           p_sep_by_space_;
char           n_cs_precedes_;
char           n_sep_by_space_;
char           p_sign_posn_;
char           n_sign_posn_;
```

These are defined identically as the correspondingly-named members of the standard C library type `lconv`, from `<locale.h>`.

RWModel

Synopsis

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

(abstract base class)

Description

This abstract base class has been designed to implement the “Model” leg of a Model-View-Controller architecture. A companion class, `RWModelClient`, supplies the “View” leg.

It maintains a list of dependent RWModelClient objects. When member function `changed(void*)` is called, the list of dependents will be traversed, calling `updateFrom(RWModel*, void*)` for each one, with itself as the first argument. Classes subclassing off `RWModelClient` should be prepared to accept such a call.

Example

This is an incomplete and somewhat contrived example in that it does not completely define the classes involved. "Dial" is assumed to be a graphical representation of the internal settings of "Thermostat". The essential point is that there is a dependency relationship between the "Thermostat" and the "Dial": when the setting of the thermostat is changed, the dial must be notified so that it can update itself to reflect the new setting of the thermostat.

Code Example 22-4

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

class Dial : public RWModelClient {
public:
  virtual voidupdateFrom(RWModel* m, void d);
};


class Thermostat : public RWModel {
  doublesetting;
public:
  Thermostat( Dial* d )
    { addDependent(d); }
  doubletemperature() const
    { return setting; }
```

Code Example 22-4 (Continued)

```
  voidsetTemperature(double t)
    { setting = t; changed(); }
};

void Dial::updateFrom(RWModel* m, void*){
  Thermostat* t = (Thermostat*)m;
  double temp = t-temperature();
  // Redraw graphic.
}
```

Public constructor

`RWModel();`
When called by the specializing class, sets up the internal ordered list of dependents.

Public member functions

`void addDependent(RWModelClient* m);`
Adds the object pointed to by `m` to the list of dependents of self.

`void removeDependent(RWModelClient* m);`
Removes the object pointed to by `m` from the list of dependents of self.

`virtual void changed(void* d);`
Traverse the internal list of dependents, calling member function `updateFrom(RWModel*, void*)` for each one, with self as the first argument and `d` as the second argument.

RWModelClient

Synopsis #include <rw/model.h>

(abstract base class)

Description This abstract base class has been designed to implement the "View" leg of a Model-View-Controller architecture. Class `RWModel`, supplies the "Model" leg. See class `RWModel` for details.

Public member function

```
virtual void         updateFrom(RWModel* p, void* d) = 0;
```

Deriving classes should supply an appropriate definition for this pure virtual function. The overall semantics of the definition should be to update self from the data presented by the object pointed to by `p`. That is, self is considered a dependent of the object pointed to by `p`. The pointer `d` is available to pass client data.

RWOrdered

```
RWOrdered
     |
RWSequenceable
     |
RWCollection
     |
RWCollectable
```

Synopsis

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

RWOrdered a;
```

Description

Class `RWOrdered` represents a group of ordered items, accessible by an index number, but not accessible by an external key. Duplicates are allowed. The ordering of elements is determined externally, generally by the order of insertion and removal. An object stored by `RWOrdered` must inherit from the abstract base class `RWCollectable`.

Class `RWOrdered` is implemented as a vector of pointers, allowing for more efficient traversing of the collection than the linked list classes `RWSlistCollectables` and `RWDlistCollectables`, but slower insertion in the center of the collection.

Public constructors

```
RWOrdered(size_t size = RWDEFAULT_CAPACITY);
```

Construct an `RWOrdered` with an initial capacity of size.

Public member operators

```
RWBoolean                   operator==(const RWOrdered& od)
                             const;
```

Returns `TRUE` if for every item in self, the corresponding item in `od` at the same index isEqual. The two collections must also have the same number of members.

```
RWCollectable*&             operator[](size_t i);
```

Returns the `i`'th element in the collection. If `i` is out of range, an exception of type `RWBoundsErr` will occur. The results of this function can be used as an lvalue.

```
RWCollectable*&            operator()(size_t i);
```
Returns the `i`'th element in the collection. Bounds checking is enabled by defining the preprocessor directive `RWBOUNDS_CHECK` before including the header file `ordcltn.h`. In this case, if `i` is out of range, an exception of type `RWBoundsErr` will occur. The results of this function can be used as an lvalue.

Public member functions

```
virtual RWCollectable*     append(RWCollectable*);
```
Redefined from class `RWSequenceable`. Adds the item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
virtual void               apply(RWapplyCollectable ap,
                            void* x);
```
Redefined from class `RWCollection`. This function has been redefined to apply the user-supplied function pointed to by `ap` to each member of the collection, in order, from first to last.

```
virtual RWCollectable*&    at(size_t i);
virtual const RWCollectable*  at(size_t i) const;
```
Redefined from class `RWSequenceable`.

```
virtual RWspace            binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void               clear();
```
Redefined from class `RWCollection`.

```
virtual void               clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                compareTo(const RWCollectable*
                            a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```
Inherited from class `RWCollection`.

```
virtual size_t             entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```

Redefined from class `RWCollection`. Returns the first item that *isEqual to* the item pointed to by `target`, or nil if no item was found.

```
virtual RWCollectable*     first() const;
```

Redefined from class `RWSequenceable`. Returns the first item in the collection.

```
virtual unsigned           hash() const;
```

Inherited from class `RWCollectable`.

```
virtual size_t             index(const RWCollectable*)
                            const;
```

Refined from class `RWSequenceable`.

```
virtual RWCollectable*     insert(RWCollectable* c);
```

Redefined from class `RWCollection`. Adds the item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
void                       insertAt(size_t indx,
                            RWCollectable* e);
```

Redefined from class `RWSequenceable`. Adds a new item to the collection at position `indx`. The item previously at positition `i` is moved to `i+1`, *etc.* The index `indx` must be between `0` and the number of items in the collection, or an exception of type `RWBoundsErr` will be thrown.

```
virtual RWClassID          isA() const;
```

Redefined from class `RWCollectable` to return `__RWORDERED`.

```
virtual RWBoolean          isEmpty() const;
```

Redefined from class `RWCollection`.

```
virtual RWBoolean          isEqual(const RWCollectable* a)
                            const;
```

Inherited from class `RWCollectable`.

```
virtual RWCollectable*     last() const;
```

Refined from class `RWSequenceable`. Returns the last item in the collection.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```

Redefined from class `RWCollection`. Returns the number of items that compare *isEqual to* the item pointed to by `target`.

```
RWCollectable*              prepend(RWCollectable*);
```
Redefined from class `RWSequenceable`. Adds the item to the beginning of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
void                        push(RWCollectable* c);
```
This is an alternative implementation of a stack to class `RWSlistCollectablesStack`. The item pointed to by `c` is put at the end of the collection.

```
RWCollectable*              pop();
```
This is an alternative implementation of a stack to class `RWSlistCollectablesStack`. The last item in the collection is removed and returned. If there are no items in the collection, nil is returned.

```
virtual RWCollectable*      remove(const RWCollectable*
                             target);
```
Redefined from class `RWCollection`. Removes the first item that *isEqual to* the item pointed to by `target` and returns it. Returns nil if no item was found.

```
virtual void                removeAndDestroy(const
                             RWCollectable* target);
```
Inherited from class `RWCollection`.

```
RWCollectable*              top() const;
```
This is an alternative implementation of a stack to class `RWSlistCollectablesStack`. The last item in the collection is returned. If there are no items in the collection, nil is returned.

RWOrderedIterator

RWOrderedIterator
|
RWIterator

Synopsis

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

RWOrdered a;
RWOrderedIterator iter(a);
```

Description

Iterator for class `RWOrdered`. Traverses the collection from the first to the last item.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructors

```
RWOrderedIterator(const RWOrdered& a);
```

Construct a `RWOrderedIterator` from a `RWOrdered`. Immediately after construction the position of the iterator is undefined.

Public member operator

```
virtual RWCollectable*     operator()();
```

Redefined from class `RWIterator`. Advances the iterator to the next item and returns it. Returns nil when the end of the collection is reached.

Public member functions

```
virtual RWCollectable*     findNext(const RWCollectable*);
```

Redefined from class `RWIterator`. Moves iterator to the next item which *isEqual to* the item pointed to by `target` and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

```
virtual RWCollectable*     key() const;
```

Redefined from class `RWIterator`. Returns the item at the current iterator position.

```
virtual void               reset();
```

Redefined from class `RWIterator`. Resets the iterator to its starting state.

22

RWpistream

```
RWpistream
    |              |
RWvistream       istream
    |
   RWvios
```

Synopsis

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

RWpistream pstr(cin);   // Construct a RWpistream, using
cin's streambuf
```

Description

Class `RWpistream` specializes the abstract base class `RWvistream` to restore variables stored in a portable ASCII format by `RWpostream`.

You can think of `RWpistream` and `RWpostream` as an ASCII veneer over an associated `streambuf` which are responsibile for formatting variables and escaping characters such that the results can be interchanged between any machines. As such, they are slower than their binary counterparts `RWbistream` and `RWbostream` which are more machine dependent. Because `RWpistream` and `RWpostream` retain no information about the state of their associated `streambufs`, their use can be freely exchanged with other users of the streambuf (such as `istream` or `ifstream`).

`RWpistream` can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, *etc.*

Example

See `RWpostream` for an example of how to create an input stream for this program.

```
#include <rw/pstream.h>
main()
{
  // Construct a RWpistream to use standard input
  RWpistream pstr(cin);
  int i;
  float f;
  double d;
  char string[80];
  pstr >> i;// Restore an int that was stored in binary
  pstr >> f >> d;   // Restore a float & double
  pstr.getString(string, 80);  // Restore a character string
}
```

Public constructors

`RWpistream(streambuf* s);`
Initialize a `RWpistream` from the `streambuf s`.

`RWpistream(istream& str);`
Initialize a `RWpistream` using the `streambuf` associated with the `istream str`.

Public member functions

`virtual int                 get();`
Redefined from class `RWvistream`. Get and return the next character from the input stream. Returns EOF if end of file is encountered.

`virtual RWvistream&         get(char& c);`
Redefined from class `RWvistream`. Get the next char and store it in `c`.

`virtual RWvistream&         get(wchar_t& wc);`
Redefined from class `RWvistream`. Get the next wide char and store it in `wc`.

`virtual RWvistream&         get(unsigned char& c);`
Redefined from class `RWvistream`. Get the next unsigned char and store it in `c`.

`virtual RWvistream&        get(char* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of char's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

Note – The vector is treated as a vector of numbers, not characters. If you wish to restore a character string, use function `getString(char*, size_t)`.

`virtual RWvistream&        get(wchar_t* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of wide `char`'s and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

Note – The vector is treated as a vector of numbers, not characters. If you wish to restore a character string, use function `getString(wchar_t*, size_t)`.

`virtual RWvistream&        get(double* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of double's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(float* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of float's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(int* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of int's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(long* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of long's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(short* v, size_t N);`
Redefined from class `RWvistream`. Get a vector of short's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(unsigned char* v, size_t N);`

Redefined from class `RWvistream`. Get a vector of unsigned char's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

Note – The vector is treated as a vector of numbers, not characters. If you wish to restore a character string, use function `getString(char*, size_t)`.

`virtual RWvistream&        get(unsigned short* v, size_t N);`

Redefined from class `RWvistream`. Get a vector of unsigned short's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(unsigned int* v, size_t N);`

Redefined from class `RWvistream`. Get a vector of unsigned int's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        get(unsigned long* v, size_t N);`

Redefined from class `RWvistream`. Get a vector of unsigned long's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

`virtual RWvistream&        getString(char* s, size_t N);`

Redefined from class `RWvistream`. Restores a character string from the input stream and stores it in the array beginning at `s`. The function stops reading at the end of the string or after `N–1` characters, whichever comes first. If the latter, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte. If the input stream has been corrupted, then an exception of type `RWExternalErr` will be thrown.

`virtual RWvistream&        getString(wchar_t* ws, size_t N);`

Redefined from class `RWvistream`. Restores a character string from the input stream and stores it in the array beginning at `ws`. The function stops reading at the end of the string or after `N-1` characters, whichever comes first. If the latter, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte. If the input stream has been corrupted, then an exception of type `RWExternalErr` will be thrown.

```
virtual RWvistream&       operator>>(char& c);
```
Redefined from class `RWvistream`. Get the next character from the input stream and store it in `c`.

```
virtual RWvistream&       operator>>(double& d);
```
Redefined from class `RWvistream`. Get the next double from the input stream and store it in `d`.

```
virtual RWvistream&       operator>>(float& f);
```
Redefined from class `RWvistream`. Get the next float from the input stream and store it in `f`.

```
virtual RWvistream&       operator>>(int& i);
```
Redefined from class `RWvistream`. Get the next int from the input stream and store it in `i`.

```
virtual RWvistream&       operator>>(long& l);
```
Redefined from class `RWvistream`. Get the next long from the input stream and store it in `l`.

```
virtual RWvistream&       operator>>(short& s);
```
Redefined from class `RWvistream`. Get the next short from the input stream and store it in `s`.

```
virtual RWvistream&       operator>>(unsigned char& c);
```
Redefined from class `RWvistream`. Get the next unsigned char from the input stream and store it in `c`.

```
virtual RWvistream&       operator>>(unsigned short& s);
```
Redefined from class `RWvistream`. Get the next unsigned short from the input stream and store it in `s`.

```
virtual RWvistream&       operator>>(unsigned int& i);
```
Redefined from class `RWvistream`. Get the next unsigned int from the input stream and store it in `i`.

```
virtual RWvistream&       operator>>(unsigned long& l);
```
Redefined from class `RWvistream`. Get the next unsigned long from the input stream and store it in `l`.

RWpostream

```
RWpostream
    |              |
RWvostream      ostream
    |
  RWvios
```

Synopsis

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

```
// Construct a RWpostream, using cout's streambuf:
RWBostream pstr(cout);
```

Description

Class `RWpostream` specializes the abstract base class `RWvostream` to store variables in a portable (printable) ASCII format. The results can be restored by using its counterpart `RWpistream`.

You can think of `RWpistream` and `RWpostream` as an ASCII veneer over an associated `streambuf` which are responsbile for formatting variables and escaping characters such that the results can be interchanged between any machines. As such, they are slower than their binary counterparts `RWbistream` and `RWbostream` which are more machine dependent. Because `RWpistream` and `RWpostream` retain no information about the state of their associated `streambufs`, their use can be freely exchanged with other users of the `streambuf` (such as `istream` or `ifstream`).

The goal of class `RWpostream` and `RWpistream` is to store variables using nothing but printable ASCII characters. Hence, nonprintable characters must be converted into an external representation where they can be recognized. Furthermore, other characters may be merely bit values (a bit image, for example), having nothing to do with characters as symbols. For example,

```
    RWpostream pstrm(cout);
    char c = '\n';

    pstr << c;      // Stores "newline"
    pstr.put(c);    // Stores the number 10.
```

The expression "`pstr << c`" treats `c` as a symbol for a newline, an unprintable character. The expression "`pstr.put(c)`" treats `c` as the literal number “10”.

Note – Variables should not be separated with whitespace. Such whitespace would be interpreted literally and would have to be read back in as a character string.

`RWpostream` can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, *etc.*

Example

See RWpistream for an example of how to read back in the results of this program. The symbol "º" is intended to represent a control-G, or bell.

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

main()
{
  // Construct a RWpostream to use standard output:
  RWpostream pstr(cout);

  int i = 5;
  float f = 22.1;
  double d = -0.05;
  char string[]
          = "A string with\ttabs,\nnewlines and a º bell.";

  pstr << i;// Store an int in binary
  pstr << f << d;// Store a float & double
  pstr << string;// Store a string
}
```

Program output:

```
5
22.1
-0.05
"A string with\ttabs,\nnewlines and a \x07 bell."
```

Public constructors

`RWpostream(streambuf* s);`
Initialize a `RWpostream` from the `streambuf s`.

`RWpostream(ostream& str);`
Initialize a `RWpostream` from the `streambuf` associated with the output stream `str`.

Public member functions

`virtual RWvostream&        operator<<(const char* s);`
Redefined from class `RWvostream`. Store the character string starting at `s` to the output stream using a portable format. The character string is expected to be null terminated.

`virtual RWvostream&        operator<<(const wchar_t* ws);`
Redefined from class `RWvostream`. Store the wide character string starting at `ws` to the output stream using a portable format. The character string is expected to be null terminated.

`virtual RWvostream&        operator<<(char c);`
Redefined from class `RWvostream`. Store the char `c` to the output stream using a portable format.

Note – `c` is treated as a character, not a number.

`virtual RWvostream&        operator<<(wchar_t wc);`
Redefined from class `RWvostream`. Store the wide char `wc` to the output stream using a portable format.

Note – `wc` is treated as a character, not a number.

`virtual RWvostream&        operator<<(unsigned char c);`
Redefined from class `RWvostream`. Store the unsigned char `c` to the output stream using a portable format.

Note – `c` is treated as a character, not a number.

`virtual RWvostream&        operator<<(double d);`
Redefined from class `RWvostream`. Store the double `d` to the output stream using a portable format.

`virtual RWvostream&        operator<<(float f);`
Redefined from class `RWvostream`. Store the float `f` to the output stream using a portable format.

`virtual RWvostream&        operator<<(int i);`
Redefined from class `RWvostream`. Store the int `i` to the output stream using a portable format.

`virtual RWvostream&       operator<<(unsigned int i);`
Redefined from class `RWvostream`. Store the unsigned int `i` to the output stream using a portable format.

`virtual RWvostream&       operator<<(long l);`
Redefined from class `RWvostream`. Store the long `l` to the output stream using a portable format.

`virtual RWvostream&       operator<<(unsigned long l);`
Redefined from class `RWvostream`. Store the unsigned long `l` to the output stream using a portable format.

`virtual RWvostream&       operator<<(short s);`
Redefined from class `RWvostream`. Store the short `s` to the output stream using a portable format.

`virtual RWvostream&       operator<<(unsigned short s);`
Redefined from class `RWvostream`. Store the unsigned short `s` to the output stream using a portable format.

`virtual RWvostream&       put(char c);`
Redefined from class `RWvostream`. Store the `char c` to the output stream, preserving its value using a portable format.

`virtual RWvostream&       put(unsigned char c);`
Redefined from class `RWvostream`. Store the `unsigned char c` to the output stream, preserving its value using a portable format.

`virtual RWvostream&       put(const char* p, size_t N);`
Redefined from class `RWvostream`. Store the vector of `chars` starting at `p` to the output stream, preserving their values using a portable format.

Note – the characters will be treated as literal numbers (*i.e.*, not as a character string).

`virtual RWvostream&       put(const wchar_t* p, size_t N);`
Redefined from class `RWvostream`. Store the vector of wide `chars` starting at `p` to the output stream, preserving their values using a portable format.

Note – the characters will be treated as literal numbers (i.e., not as a character string).

```
virtual RWvostream&       put(const unsigned char* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of unsigned chars starting at `p` to the output stream using a portable format. The characters should be treated as literal numbers (*i.e.*, not as a character string).

```
virtual RWvostream&       put(const short* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of shorts starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const unsigned short* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of unsigned shorts starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const int* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of ints starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const unsigned int* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of unsigned ints starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const long* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of longs starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const unsigned long* p,
                           size_t N);
```
Redefined from class `RWvostream`. Store the vector of unsigned longs starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const float* p, size_t N);
```
Redefined from class `RWvostream`. Store the vector of floats starting at `p` to the output stream using a portable format.

```
virtual RWvostream&       put(const double* p, size_t
                           N);
```
Redefined from class `RWvostream`. Store the vector of doubles starting at `p` to the output stream using a portable format.

22

RWSequenceable

RWSequenceable
|
RWCollection
|
RWCollectable

Synopsis

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

typedef RWSequenceable SequenceableCollection; // Smalltalk
typedef
```

Description

Class `RWSequenceable` is an abstract base class for collections that can be accessed via an index. It inherits class `RWCollection` as a public base class and adds a few extra virtual functions. This documentation only describes these extra functions.

Public member functions

```
RWCollectable*              append(RWCollectable*) = 0;
```

Adds the item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
virtual RWCollectable*&    at(size_t i);
virtual const RWCollectable*    at(size_t i) const;
```

Allows access to the `i`'th element of the collection. The first variant can be used as an lvalue, the second cannot. The index i must be between zero and the number of items in the collection less one, or an exception of type `RWBoundsErr` will be thrown.

```
virtual RWCollectable*     first() const = 0;
```

Returns the first item in the collection.

```
virtual size_t             index(const RWCollectable* c)
                            const = 0;
```

Returns the index number of the first item that "matches" the item pointed to by `c`. If there is no such item, returns `RW_NPOS`. For most collections, an item "matches" the target if either `isEqual()` or `compareTo()` find equivalence, whichever is appropriate for the actual collection type.

```
virtual void               insertAt(size_t indx,
                            RWCollectable* e);
```
Adds a new item to the collection at position `indx`. The item previously at position `i` is moved to `i+1`, *etc*. The index `indx` must be between 0 and the number of items in the collection, or an exception of type `RWBoundsErr` will be thrown.

```
virtual RWCollectable*     last() const = 0;
```
Returns the last item in the collection.

```
RWCollectable*             prepend(RWCollectable*) = 0;
```
Adds the item to the beginning of the collection and returns it. Returns nil if the insertion was unsuccessful.

RWSet

RWSet
|
RWHashTable
|
RWCollection
|
RWCollectable

Synopsis

```
typedef RWSet Set;        // Smalltalk typedef.
#include <rw/rwset.h>

RWSet h;
```

Description

Class `RWSet` represents a group of unordered elements, not accessible by an external key, where duplicates are not allowed. It corresponds to the `Smalltalk` class `Set`.

An object stored by `RWSet` must inherit abstract base class `RWCollectable`, with suitable definition for virtual functions `hash()` and `isEqual()` (see class `RWCollectable`). The function `hash()` is used to find objects with the same hash value, then `isEqual()` is used to confirm the match.

An item `c` is considered to be "already in the collection" if there is a member of the collection for which `isEqual(c)` returns `TRUE`. In this case, the message `insert(c)` will not add it, thus insuring that there are no duplicates.

The iterator for this class is `RWSetIterator`.

Public constructors

`RWSet (size_t n = RWDEFAULT_CAPACITY);`
Constructs an empty set with `n` hashing buckets.

`RWSet (const RWSet & h);`
Copy constructor. Makes a shallow copy of the collection `h`.

`virtual                    ~RWSet();`
Calls `clear()`.

Public member operators

`void                          operator=(const RWSet& h);`

Assignment operator. Makes a shallow copy of the collection `h`.

`RWBoolean                     operator==(const RWSet& h);`

Returns `TRUE` if self and `h` have the same number of elements and if for every key in self there is a corresponding key in `h` which isEqual.

`RWBoolean                     operator!=(const RWSet& h);`

Returns the negation of `operator==()`, above.

`RWBoolean                     operator<=(const RWSet& h);`

Returns `TRUE` if self is a subset of `h`, that is, every element of self has a counterpart in `h` which isEqual.

Public member functions

```
virtual voidapply(RWapplyCollectable ap,
 void*)
```

Redefined from class `RWCollection` to apply the user-supplied function pointed to by `ap` to each member of the collection in a (generally) unpredictable order. This supplied function should not do anything to the items that could change the ordering of the collection.

`virtual RWspace               binaryStoreSize() const;`

Inherited from class `RWCollection`.

`virtual void                  clear();`

Redefined from class `RWCollection`.

`virtual void                  clearAndDestroy();`

Inherited from class `RWCollection`.

```
virtual int                   compareTo(const RWCollectable*
                               a) const;
```

Inherited from class `RWCollectable`.

```
virtual RWBoolean             contains(const RWCollectable*
                               target) const;
```

Inherited from class `RWCollection`.

`virtual size_t                entries() const;`

Redefined from class `RWCollection`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```
Redefined from class `RWCollection`. Returns the item in self which *isEqual to* the item pointed to by `target` or nil if no item is found. Hashing is used to narrow the search.

```
virtual unsigned           hash() const;
```
Inherited from class `RWCollectable`.

```
virtual RWCollectable*     insert(RWCollectable* c);
```
Redefined from class `RWCollection`. Adds `c` to the collection and returns it. If an item is already in the collection which *isEqual to* `c`, then the old item is returned and the new item is not inserted.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWSET`.

```
virtual RWBoolean          isEmpty() const;
```
Redefined from class `RWCollection`.

```
virtual RWBoolean          isEqual(const RWCollectable* a)
                            const;
```
Redefined to return TRUE is the object pointed to by `a` is of the same type as self, and self == `t`.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```
Redefined from class `RWCollection`. Returns the number of entries that *isEqual to* the item pointed to by `target`. Because duplicates are not allowed for this collection, only 0 or 1 can be returned.

```
virtual RWCollectable*     remove(const RWCollectable*
                            target);
```
Redefined from class `RWCollection`. Returns and removes the item that *isEqual to* the item pointed to by `target`, or nil if there is no item.

```
virtual void               removeAndDestroy(const
                            RWCollectable* target);
```
Inherited from class `RWCollection`.

```
void                       resize(size_t n = 0);
```
Resizes the internal hashing table to the next highest prime number that is greater than or equal to `n`. If `n==0`, then the next highest prime number greater than or equal to the present size.

```
virtual void                restoreGuts(RWvistream&);
virtual void                restoreGuts(RWFile&);
virtual void                saveGuts(RWvostream&) const;
virtual void                saveGuts(RWFile&) const;
```

Inherited from class `RWCollection`.

RWSetIterator

```
RWSetIterator
      |
RWHashTableIterator
      |
RWIterator
```

Synopsis

```
#include <rw/rwset.h>
RWSet h;
RWSetIterator it(h);
```

Description

Iterator for class `RWSet`, which allows sequential access to all the elements of `RWSet`.

Note – Because a `RWSet` is unordered, elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWSetIterator(RWSet&);`
Construct an iterator for an `RWSet`. After construction, the position of the iterator will be undefined.

Public member operator

`virtual RWCollectable*     operator()();`
Inherited from `RWHashTableIterator`.

Public member functions

```
virtual RWCollectable*     findNext(const RWCollectable*
                            target);
```
Inherited from `RWHashTableIterator`.

```
virtual RWCollectable*     key() const;
```
Inherited from `RWHashTableIterator`.

```
RWCollectable*             remove();
```
Inherited from `RWHashTableIterator`.

```
RWCollectable*             removeNext(const
                            RWCollectable*);
```
Inherited from `RWHashTableIterator`.

```
virtual void               reset();
```
Inherited from `RWHashTableIterator`.

RWSlistCollectables

```
     RWSlistCollectables
          |              |
RWSequenceable     RWSlist
          |
RWCollection
          |
RWCollectable
```

Synopsis

```
// Smalltalk typedef:
typedef RWSlistCollectables LinkedList;

#include <rw/slistcol.h>
RWSlistCollectables a;
```

Description

Class `RWSlistCollectables` represents a group of ordered elements, without keyed access. Duplicates are allowed. The ordering of elements is determined externally, by the order of insertion and removal. An object stored by `RWSlistCollectables` must inherit abstract base class `RWCollectable`.

The virtual function `isEqual()` (see class `RWCollectable`) is required to find a match between a target and an item in the collection

Class `RWSlistCollectables` is implemented as a singly-linked list, which allows for efficient insertion and removal, but efficient movement in only one direction. This class corresponds to the Smalltalk class `LinkedList`.

Public constructors

```
RWSlistCollectables();
```
Constructs an empty linked list.

```
RWSlistCollectables(const RWCollectable* a);
```
Constructs an ordered collection with single item `a`.

Public member operators

```
RWBoolean                    operator==(const
                               RWSlistCollectables& s) const;
```
Returns `TRUE` if self and `s` have the same number of members and if for every item in self, the corresponding item at the same index in `s` isEqual to it.

Public member functions

```
virtual RWCollectable*     append(RWCollectable*);
```
Redefined from `RWSequenceable`. Inserts the item at the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
virtual void               apply(RWapplyCollectable ap,
                            void*)
```
Redefined from class `RWCollection`. This function has been redefined to apply the user-defined function pointed to by `ap` to each member of the collection, in order, from first to last.

```
virtual RWCollectable*&    at(size_t i);
virtual const RWCollectable*   at(size_t i) const;
```
Redefined from class `RWSequenceable`. The index `i` must be between 0 and the number of items in the collection less one, or an exception of type `RWBoundsErr` will be thrown.

Note – For a linked-list, these functions must traverse all the links, making them not particularly efficient.

```
virtual RWspace            binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void               clear();
```
Redefined from class `RWCollection`.

```
virtual void               clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                compareTo(const RWCollectable*
                            a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```
Inherited from class `RWCollection`.

```
RWBoolean                  containsReference(const
                            RWCollectable* e) const;
```
Returns true if the list contains an item that *is identical to* the item pointed to by `e` (that is, that has the address `e`).

```
virtual size_t             entries() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```
Redefined from class `RWCollection`. The first item that matches `target` is returned, or nil if no item was found.

```
RWCollectable*             findReference(const
                            RWCollectable* e) const;
```
Returns the first item that *is identical to* the item pointed to by `e` (that is, that has the address `e`), or nil if none is found.

```
virtual RWCollectable*     first() const;
```
Redefined from class `RWSequenceable`. Returns the item at the beginning of the list.

```
RWCollectable*             get();
```
Returns and *removes* the item at the beginning of the list.

```
virtual unsigned           hash() const;
```
Inherited from class `RWCollectable`.

```
virtual size_t             index(const RWCollectable* c)
                            const = 0;
```
Redefined from class `RWSequenceable`. Returns the index of the first item that *isEqual to* the item pointed to by `c`.

```
virtual RWCollectable*     insert (RWCollectable* c);
```
Redefined from class `RWCollection`. Adds the item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
void                       insertAt(size_t indx,
                            RWCollectable* e);
```
Redefined from class `RWSequenceable`. Adds a new item to the collection at position `indx`. The item previously at positition `i` is moved to `i+1`, *etc.* The index `indx` must be between `0` and the number of items in the collection, or an exception of type `RWBoundsErr` will be thrown.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWSLISTCOLLECTABLES`.

```
virtual RWBoolean          isEmpty() const;
```
Redefined from class `RWCollection`.

```
virtual RWCollectable*     last() const;
```
Redefined from class `RWSequenceable`. Returns the value at the end of the collection.

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
```
Redefined from class `RWCollection`. Returns the number of items that *isEqual to* the item pointed to by `target`.

```
size_t                     occurrencesOfReference(const
                            RWCollectable* e) const;
```
Returns the number of items that *are identical to* the item pointed to by `e` (that is, that have the address `e`).

```
virtual RWCollectable*     prepend(RWCollectable*);
```
Redefined from class `RWSequenceable`. Adds the item to the beginning of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
virtual RWCollectable*     remove(const RWCollectable*
                            target);
```
Redefined from class `RWCollection`. Removes and returns the first item that *isEqual to* the item pointed to by `target`. Returns nil if there is no such item.

```
virtual void               removeAndDestroy(const
                            RWCollectable* target);
```
Inherited from class `RWCollection`.

```
RWCollectable*             removeReference(const
                            RWCollectable* e);
```
Removes and returns the first item that *is identical to* the item pointed to by `e` (that is, that has the address `e`). Returns nil if there is no such item.

```
virtual void               restoreGuts(RWvistream&);
virtual void               restoreGuts(RWFile&);
virtual void               saveGuts(RWvostream&) const;
virtual void               saveGuts(RWFile&) const;
```
Inherited from class `RWCollection`.

RWSlistCollectablesIterator

```
RWSlistCollectablesIterator
    |              |
RWIterator    RWSlistIterator
```

Synopsis

```
// Smalltalk typedef.
typedef RWSlistCollectablesIterator     LinkedListIterator;

#include <rw/slistcol.h>
RWSlistCollectables sc;
RWSlistCollectablesIterator sci(sc);
```

Description

Iterator for class `RWSlistCollectables`. Traverses the linked-list from the first to last item.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWSlistCollectablesIterator (RWSlistCollectables&);`
Constructs an iterator from a singly-linked list. Immediately after construction, the position of the iterator will be undefined.

Public member operators

`virtual RWCollectable*     operator()();`
Redefined from class `RWIterator`. Advances the iterator to the next element and returns it. Returns nil when the end of the collection is reached.

`void                       operator++();`
Advances the iterator one item.

`void                       operator+=(size_t n);`
Advances the iterator `n` items.

Public member functions

`RWBoolean                  atFirst() const;`
Returns `TRUE` if the iterator is at the beginning of the list, otherwise `FALSE`;

`RWBoolean                  atLast() const;`

Returns `TRUE` if the iterator is at the end of the list, otherwise `FALSE`;

`virtual RWCollectable*     findNext(const RWCollectable* target);`

Redefined from class `RWIterator`. Moves iterator to the next item which *isEqual to* the item pointed to by `target` and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

`RWCollectable*             findNextReference(const RWCollectable* e);`

Moves iterator to the next item which *is identical to* the item pointed to by `e` (that is, that has address `e`) and returns it. If no item is found, returns nil and the position of the iterator will be undefined.

`RWCollectable*             insertAfterPoint(RWCollectable* a);`

Insert item `a` after the current cursor position and return the item. The cursor's position will be unchanged.

`virtual RWCollectable*     key() const;`

Redefined from class `RWIterator`. Returns the item at the current iterator position.

`RWCollectable*             remove();`

Removes and returns the item at the current cursor position. Afterwards, the iterator will be positioned at the previous item in the list.

Note – This function is not very efficient in a singly-linked list.

`RWCollectable*             removeNext(const RWCollectable* target);`

Moves iterator to the next item in the list which *isEqual to* the item pointed to by `target`, removes it from the list and returns it. Afterwards, the iterator will be positioned at the previous item in the list. If no item is found, returns nil and the position of the iterator will be undefined.

`RWCollectable*             removeNextReference(const RWCollectable* e);`

Moves iterator to the next item in the list which *is identical to* the item pointed to by `e` (that is, that has address `e`), removes it from the list and returns it. Afterwards, the iterator will be positioned at the previous item in the list. If no item is found, returns nil and the position of the iterator will be undefined.

```
virtual void               reset();
```
Redefined from class `RWIterator`. Resets the iterator. Afterwards, the position of the iterator will be undefined.

```
void                       toFirst();
```
Moves the iterator to the beginning of the list.

```
void                       toLast();
```
Moves the iterator to the end of the list.

RWSlistCollectablesQueue

```
RWSlistCollectablesQueue
        |
  RWSlistCollectables
        |              |
RWSequenceable   RWSlist
        |
RWCollection
        |
RWCollectable
```

Synopsis

```
// Smalltalk typedef:
typedef RWSlistCollectablesQueue Queue;
```

```
#include <rw/queuecol.h>
RWSlistCollectablesQueue a;
```

Description

Class `RWSlistCollectablesQueue` represents a restricted interface to class `RWSlistCollectables` to implement a first in first out (FIFO) queue. A Queue is a sequential list for which all insertions are made at one end (the "tail"), but all removals are made at the other end (the "head"). Hence, the ordering is determined externally by the ordering of the insertions. Duplicates are allowed.

An object stored by `RWSlistCollectablesQueue` must inherit abstract base class `RWCollectable`. The virtual function `isEqual()` (see class `RWCollectable`) is required to find a match between a target and an item in the queue.

This class corresponds to the `Smalltalk` class `Queue`.

Public constructors

`RWSlistCollectablesQueue();`
Construct an empty queue.

`RWSlistCollectablesQueue(RWCollectable* a);`
Construct an queue with single item `a`.

`RWSlistCollectablesQueue(const RWSlistCollectablesQueue & q);`
Copy constructor. A shallow copy of the queue `q` is made.

Public member operators

```
void                         operator=(const
                              RWSlistCollectablesQueue & q);
```
Assignment operator. A shallow copy of the queue `q` is made.

Public member functions

```
virtual void                 apply(RWapplyCollectable ap,
                              void*)
```
Inherited from class `RWSlistCollectables`.

```
virtual RWCollectable*     append(RWCollectable*);
```
Inherited from class `RWSlistCollectables`. Adds an element to the end of the queue.

```
virtual RWspace            binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void               clear();
```
Inherited from class `RWSlistCollectables`.

```
virtual void               clearAndDestroy();
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```
Inherited from class `RWCollection`.

```
RWBoolean                  containsReference(const
                            RWCollectable* e) const;
virtual size_t             entries() const;
```
Inherited from class `RWSlistCollectables`.

```
virtual RWCollectable*     first() const;
```
Inherited from class `RWSlistCollectables`. Returns the item at the beginning of the queue (*i.e.*, the least recently inserted item). Returns nil if the queue is empty.

```
RWCollectable*             get();
```
Inherited from class `RWSlistCollectables`. Returns and *removes* the item at the beginning of the queue (*i.e.*, the least recently inserted item). Returns nil if the queue is empty.

```
virtual RWCollectable*     insert(RWCollectable* c);
```
Redefined from class `RWSlistCollectables` to call `append()`.

```
virtual RWClassID          isA() const;
```
Redefined from class `RWCollectable` to return `__RWSLISTCOLLECTABLESQUEUE`.

```
virtual RWBoolean          isEmpty() const;
```
Inherited from class `RWSlistCollectables`.

```
virtual RWCollectable*     last() const;
```
Inherited from class `RWSlistCollectables`. Returns the last item in the queue (the most recently inserted item).

```
virtual size_t             occurrencesOf(const
                            RWCollectable* target) const;
size_t                     occurrencesOfReference(const
                            RWCollectable* e) const;
```
Inherited from class `RWSlistCollectables`.

```
virtual RWCollectable*     remove(const RWCollectable*);
```
Redefined from class `RWSlistCollectables`. Calls `get()`. The argument is ignored.

RWSlistCollectablesStack

```
RWSlistCollectablesStack
              |
   RWSlistCollectables
         |               |
RWSequenceable     RWSlist
         |
RWCollection
         |
RWCollectable
```

Synopsis

```
// Smalltalk typedef:
typedef RWSlistCollectablesStackStack;

#include <rw/stackcol.h>
RWSlistCollectablesStack a;
```

Description

Class `RWSlistCollectablesStack` represents a restricted interface to class `RWSlistCollectables` to implement a last in first out (LIFO) stack. A Stack is a sequential list for which all insertions and deletions are made at one end (the beginning of the list). Hence, the ordering is determined externally by the ordering of the insertions. Duplicates are allowed.

An object stored by `RWSlistCollectablesStack` must inherit abstract base class `RWCollectable`. The virtual function `isEqual()` (see class `RWCollectable`) is required to find a match between a target and an item in the stack.

This class corresponds to the `Smalltalk` class `Stack`.

Public constructors

```
RWSlistCollectablesStack();
```
Construct an empty stack.

```
RWSlistCollectablesStack(RWCollectable* a);
```
Construct a stack with one entry `a`.

```
RWSlistCollectablesStack(const RWSlistCollectablesStack&
s);
```
Copy constructor. A shallow copy of the stack `s` is made.

Assignment operator

```
void                          operator=(const
                               RWSlistCollectablesStack& s);
```

Assignment operator. A shallow copy of the stack s is made.

Public member functions

```
virtual void                  apply(RWapplyCollectable ap,
void*)
virtual RWspace               binaryStoreSize() const;
virtual void                  clear();
```

Inherited from class `RWSlistCollectables`.

```
virtual void                  clearAndDestroy();
virtual RWBoolean             contains(const RWCollectable*
                               target) const;
```

Inherited from class `RWCollection`.

```
RWBoolean                     containsReference(const
                               RWCollectable* e) const;
virtual size_t                entries() const;
```

Inherited from class `RWSlistCollectables`.

```
virtual RWCollectable*        first() const;
```

Inherited from class `RWSlistCollectables`. Same as `top()`.

```
virtual RWCollectable*        insert(RWCollectable* c);
```

Inherited from class `RWSlistCollectables`. Same as `push()`.

```
virtual RWClassID             isA() const;
```

Redefined from class `RWCollectable` to return `__RWSLISTCOLLECTABLESSTACK`.

```
virtual RWBoolean             isEmpty()const;
```

Inherited from class `RWSlistCollectables`.

```
virtual RWCollectable*        last() const;
```

Inherited from class `RWSlistCollectables`. Returns the item at the bottom of the stack.

```
virtual size_t                occurrencesOf(const
                               RWCollectable* target) const;
size_t                        occurrencesOfReference(const
                               RWCollectable* e) const;
```

Inherited from class `RWSlistCollectables`.

`virtual RWCollectable*     remove(const RWCollectable*);`
Redefined from class `RWSlistCollectables`. Calls `pop()`. The argument is ignored.

`RWCollectable*             pop();`
Removes and returns the item at the top of the stack, or returns nil if the stack is empty.

`void                       push(RWCollectable*);`
Adds an item to the top of the stack.

`RWCollectable*             top() const;`
Returns the item at the top of the stack or nil if the stack is empty.

RWSortedVector

RWSortedVector
|
RWOrdered
|
RWSequenceable
|
RWCollection
|
RWCollectable

Synopsis

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

RWSortedVector a;
```

Description

Class `RWSortedVector` represents a group of ordered items, internally sorted by the `compareTo()` function and accessible by an index number. Duplicates are allowed. An object stored by `RWSortedVector` must inherit from the abstract base class `RWCollectable`. An insertion sort is used to maintain the vector in sorted order.

Because class `RWSortedVector` is implemented as a vector of pointers, traversing the collection is more efficient than with class `RWBinaryTree`. However, insertions are slower in the center of the collection.

Example

```
#include   <rw/sortvec.h>
#include   <rw/collstr.h>
#include   <rw/rstream.h>
main()
{
  RWSortedVector sv;
  sv.insert(new RWCollectableString("dog"));
  sv.insert(new RWCollectableString("cat"));
  sv.insert(new RWCollectableString("fish"));
  RWSortedVectorIterator next(sv);
  RWCollectableString* item;
  while( item = (RWCollectableString*)next() )
    cout << *item << endl;
  sv.clearAndDestroy();
}
```

Program output:

```
cat
dog
fish
```

Public constructors

`RWSortedVector(size_t size = RWDEFAULT_CAPACITY);`
Construct an empty `RWSortedVector` that has an initial capacity of size items. The capacity will be increased automatically if excess items are inserted into the collection.

Public member operators

```
RWBoolean                   operator==(const
                             RWSortedVector& sv) const;
```
Returns `TRUE` if for every item in self, the corresponding item in `sv` at the same index *is equal*. The two collections must also have the same number of members.

```
const RWCollectable*        operator[](size_t i);
```
Returns the `i`'th element in the collection. If `i` is out of range, an exception of type `RWBoundsErr` will be thrown. This function cannot be used as an lvalue.

```
const RWCollectable*       operator()(size_t i);
```
Returns the `i`'th element in the collection. Bounds checking is enabled by defining the preprocessor directive `RWBOUNDS_CHECK` before including the header file `sortvec.h`. In this case, if `i` is out of range, an exception of type `RWBoundsErr` will be thrown. This function cannot be used as an lvalue.

Public member functions

```
virtual void               apply(RWapplyCollectable ap,
                            void* x);
```
Inherited from class `RWOrdered`.

```
virtual const RWCollectable*    at(size_t i) const;
```
Inherited from class `RWOrdered`.

```
virtual RWspace            binaryStoreSize() const;
```
Inherited from class `RWCollection`.

```
virtual void               clear();
```
Inherited from class `RWOrdered`.

```
virtual void               clearAndDestroy();
```
Inherited from class `RWCollection`.

```
virtual int                compareTo(const RWCollectable*
                            a) const;
```
Inherited from class `RWCollectable`.

```
virtual RWBoolean          contains(const RWCollectable*
                            target) const;
```
Inherited from class `RWCollection`.

```
virtual size_t             entries() const;
```
Inherited from class `RWOrdered`.

```
virtual RWCollectable*     find(const RWCollectable*
                            target) const;
```
Inherited from class `RWOrdered`.

Note – `RWOrdered::find()` uses the virtual function `index()` to perform its search. Hence, a binary search will be used.

```
virtual RWCollectable*     first() const;
```
Inherited from class `RWOrdered`.

```
virtual unsigned          hash() const;
```
Inherited from class RWCollectable.

```
virtual size_t            index(const RWCollectable*)
                           const;
```
Redefined from class RWOrdered. Performs a binary search to return the index of the first item that *compares equal* to the target item, or RW_NPOS if no such item can be found.

```
virtual RWCollectable*    insert(RWCollectable* c);
```
Redefined from class RWOrdered. Performs a binary search to insert the item pointed to by c after all items that compare less than or equal to it, but before all items that compare greater than it. Returns nil if the insertion was unsuccessful, c otherwise.

```
virtual RWClassID         isA() const;
```
Redefined from class RWCollectable to return __RWSORTEDVECTOR.

```
virtual RWBoolean         isEmpty() const;
```
Inherited from class RWOrdered.

```
virtual RWBoolean         isEqual(const RWCollectable* a)
                           const;
```
Redefined to return TRUE is the object pointed to by a is of the same type as self, and self == t.

```
virtual RWCollectable*    last() const;
```
Inherited from class RWOrdered.

```
virtual size_t            occurrencesOf(const
                           RWCollectable* target) const;
```
Redefined from class RWOrdered. Returns the number of items that *compare equal* to the item pointed to by target.

```
virtual RWCollectable*    remove(const RWCollectable*
                           target);
```
Inherited from class RWOrdered.

Note – RWOrdered::remove() uses the virtual function index() to perform its search. Hence, a binary search will be used.

```
virtual void                removeAndDestroy(const
                              RWCollectable* target);
```

Inherited from class `RWCollection`.

RWTime

Synopsis

```
#include <rw/rwtime.h>
RWTime a;              // Construct with current time
```

Description

Class `RWTime` represents a time, stored as the number of seconds since 00:00:00 January 1901 UTC. See "Setting the time zone" on page 59 in Chapter 8, "Using Class RWTime," for how to set the time zone.

Caution – Failure to do this may result in UTC (GMT) times being wrong.

Output formatting is now done using an `RWLocale` object. The default locale formats according to U.S. conventions.

Example

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

main()
{
  RWTime t; // Current time
  RWTime d(RWTime::beginDST(1990, RWZone::local()));
  cout << "Current time:       " << RWDate(t) << " " << t << endl;
  cout << "Start of DST, 1990:  " << RWDate(d) << " " << d << endl;
}
```

Program output

```
Current time:           03/22/91 15:01:40
Start of DST, 1990:     05/01/90 02:00:00
```

Public constructors

`RWTime();`
Default constructor. Constructs a time with the present time.

`RWTime(const RWTime&);`
Copy constructor, generated by the compiler.

`RWTime(unsigned long s);`
Constructs a time with s seconds since 00:00:00 January 1, 1901 UTC.

```
RWTime(unsigned hour, unsigned minute, unsigned second = 0,
                       const RWZone& zone =
                       RWZone::local());
```
Constructs a time with today's date, and the specified hour, minute, and second, relative to the time zone `zone`, which defaults to local time.

```
RWTime(const RWDate& date, unsigned hour = 0, unsigned
                       minute = 0, unsigned second = 0,
                       const RWZone& =
                       RWZone::local());
```
Constructs a time for a given date, hour, minute, and second, relative to the time zone `zone`, which defaults to local time.

```
RWTime(const struct tm*, const RWZone& = RWZone::local());
```
Constructs a time from the `tm_year, tm_mon, tm_mday, tm_hour, tm_min,` and `tm_sec` components of the `struct tm` argument. These components are understood to be relative to the time zone `zone`, which defaults to local time.

Note – the numbering of months and years in a struct tm differs from that used in RWTime arguments.

```
RWTime(const RWDate& date, const RWCString& str, const
                       RWZone& zone = RWZone::local(),
                       const RWLocale& locale =
                       RWLocale::global());
```
Constructs a time for the given date, extracting the time from the string `str`. The time is understood to be relative to the time zone `zone`, which defaults to local time. The specified locale is used for formatting information . Use function `isValid()` to check the results.

Note – not all time string errors can be detected by this function.

Public member operators

```
RWTime&                operator=(const RWTime&);
```
Assignment operator, generated by the compiler.

```
RWTime                 operator++();
```
Prefix increment operator. Add one second to self, then return the results.

```
RWTime                     operator--();
```
Prefix decrement operator. Subtract one second from self, then return the results.

```
RWTime                     operator++(int);
```
Postfix increment operator. Add one second to self, returning the initial value.

```
RWTime                     operator--(int);
```
Postfix decrement operator. Subtract one second from self, returning the initial value.

```
RWTime&                    operator+=(long s);
```
Add `s` seconds to self, returning self.

```
RWTime&                    operator-=(long s);
```
Subtract `s` seconds from self, returning self.

Public member functions

```
RWCString                  asString(char format = '\0',
                            const RWZone&    =
                            RWZone::local(), const
                            RWLocale& = RWLocale::global())
                            const;
```
Returns self as a string, formatted by the `RWLocale` argument, with the time zone adjusted according to the `RWZone` argument. Formats are as defined by the standard C library function `strftime()`. The default format is the date followed by the time: `"%x %X"`.

```
RWBooleanbetween(const RWTime& a, const RWTime& b) const;
```
Returns `TRUE` if `RWTime` is between `a` and `b`, inclusive.

```
size_t                     binaryStoreSize() const;
```
Returns the number of bytes necessary to store the object using the global function

```
    RWFile& operator<<(RWFile&, const RWTime&);
```

```
int                        compareTo(const RWTime* t)
                            const;
```
Comparison function, useful for sorting times. Compares self to the `RWTime` pointed to by `t` and returns:

0	if self == *`t`;
1	if self > *`t`;
–1	if self < *`t`.

```
void                           extract(struct tm*, const
                                RWZone& = RWZone::local())
                                const;
```
Fills all members of the `struct tm` argument, adjusted to the time zone specified by the `RWZone` argument. If the time is invalid, the `struct tm` members are all set to -1.

Note – the encoding of struct tm members is different from that used in RWTime and RWDate functions.

```
unsigned                       hash() const;
```
Returns a suitable hashing value.

```
unsigned                       hour(const RWZone& zone =
                                RWZone::local()) const;
```
Returns the hour, adjusted to the time zone specified.

```
unsigned                       hourGMT() const;
```
Returns the hour in UTC (GMT).

```
RWBoolean                      isDST(const RWZone& zone =
                                RWZone::local()) const;
```
Returns `TRUE` if self is during Daylight Savings Time in the time zone given by `zone, FALSE` otherwise.

```
RWBoolean                      isValid() const;
```
Returns `TRUE` if this is a valid time, `FALSE` otherwise.

```
RWTime                         max(const RWTime& t) const;
```
Returns the later time of self or `t`.

```
RWTime                         min(const RWTime& t) const;
```
Returns the earlier time of self or `t`.

```
unsigned                       minute(const RWZone& zone =
                                RWZone::local()) const;
```
Returns the minute, adjusted to the time zone specified.

```
unsigned                       minuteGMT() const;
```
Returns the minute in UTC (GMT).

```
static RWTime                  now();
```
Returns the current time.

```
unsigned                      second() const;
```
Returns the second; local time or UTC (GMT).

```
unsigned long                 seconds() const;
```
Returns the number of seconds since 00:00:00 January 1, 1901 UTC.

Static public member functions

```
static RWTime                 beginDST(unsigned year,
                               const RWZone& zone =
                               RWZone::local());
```
Return the start of Daylight Savings Time (DST) for the given `year`, in the given time zone. Returns an "invalid time" if DST is not observed in that year and zone.

```
static RWTime                 endDST(unsigned year, const
                               RWZone& = RWZone::local());
```
Return the end of Daylight Savings Time for the given `year`, in the given time zone. Returns an "invalid time" if DST is not observed in that year and zone.

Related global operators

```
RWTime                        operator+(const RWTime& t,
                               long s          );
RWTime                        operator+(long s,          const
                               RWTime& t);
```
Returns a `RWTime` `s` seconds greater than `t`.

```
RWTime                        operator-(const RWTime& t,
                               long s          );
```
Returns a `RWTime` `s` seconds less than `t`.

```
RWBoolean                     operator<(const RWTime& t1,
                               const RWTime& t2);
```
Returns `TRUE` if `t1` is less than `t2`.

```
RWBoolean                     operator<=(const RWTime& t1,
                               const RWTime& t2);
```
Returns `TRUE` if `t1` is less than or equal to `t2`.

```
RWBoolean                    operator>(const RWTime& t1,
                              const RWTime& t2);
```
Returns TRUE if t1 is greater than t2.

```
RWBoolean                    operator>=(const RWTime& t1,
                              const RWTime& t2);
```
Returns TRUE if t1 is greater than or equal to t2.

```
RWBoolean                    operator==(const RWTime& t1,
                              const RWTime& t2);
```
Returns TRUE if t1 is equal to t2.

```
RWBoolean                    operator!=(const RWTime& t1,
                              const RWTime& t2);
```
Returns TRUE if t1 is not equal to t2.

```
ostream&                     operator<<(ostream& s, const
                              RWTime& t);
```
Outputs the time t on ostream s, according to the locale imbued in the stream (see class RWLocale), or by RWLocale::global() if none.

```
RWvostream&                  operator<<(RWvostream&, const
                              RWTime& t);
RWFile&                      operator<<(RWFile&,      const
                              RWTime& t);
```
Saves RWTime t to a virtual stream or RWFile, respectively.

```
RWvistream&                  operator>>(RWvistream&,
                              RWTime& t);
RWFile&                      operator>>(RWFile&,
                              RWTime& t);
```
Restores an RWTime into t from a virtual stream or RWFile, respectively, replacing the previous contents of t.

RWTimer

Synopsis

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

RWTimer  timer;
```

Description

This class can measure elapsed CPU (user) time. The timer has two states: running and stopped. The timer measures the total amount of time spent in the "running" state since it was either constructed or reset.

The timer is put into the "running" state by calling member function `start()`. It is put into the "stopped" state by calling `stop()`.

Example

This example prints out the amount of CPU time used while looping for 5 seconds (as measured using class RWTime).

```
#include <rw/timer.h>
#include <rw/rwtime.h>
#include <rw/rstream.h>

main()
{
  RWTimer t;
  t.start();// Start the timer

  RWTime start;
  start.now()// Record starting time

  // Loop for 5 seconds;
  for (RWTime current; current.seconds() - start.seconds() < 5;
                                          current.now())
    {;}

  t.stop()Stop the timer

  cout << t.elapsedTime() << endl;
  return 0;
}
```

Program output:

```
5.054945
```

Public constructor

```
RWTimer();
```
Constructs a new timer. The timer will not start running until `start()` is called.

```
double                    elapsedTime() const;
```
Returns the amount of (CPU) time that has accumulated while the timer was in the running state.

```
void                      reset();
```
Resets (and stops) the timer.

```
void                      start();
```
Puts the timer in the "running" state. Time accumulates while in this state.

```
void                      stop();
```
Puts the timer in the "stopped" state. Time will not accumulate while in this state.

22

RWVirtualPageHeap

Synopsis

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

(*Abstract base class*)

Description

This is an abstract base class representing an abstract page heap of fixed sized pages. The following describes the model by which specializing classes of this class are expected to work.

You allocate a page off the abstract heap by calling member function `allocate()` which will return a memory "handle", an object of type `RWHandle`. This handle logically represents the page.

In order to use the page it must first be "locked" by calling member function `lock()` with the handle as an argument. It is the job of the specializing class of `RWVirtualPageHeap` to make whatever arrangements are necessary to swap in the page associated with the handle and bring it into physical memory. The actual swapping medium could be disk, expanded or extended memory, or a machine someplace on a network. Upon return, `lock()` returns a pointer to the page, now residing in memory.

Once a page is in memory, you are free to do anything you want with it although if you change the contents, you must call member function `dirty()` before unlocking the page.

Locked pages use up memory. In fact, some specializing classes may have only a fixed number of buffers in which to do their swapping. If you are not using the page, you should call `unlock()`. After calling `unlock()` the original address returned by `lock()` is no longer valid—to use the page again, it must be locked again with `lock()`.

When you are completely done with the page then call `deallocate()` to return it to the abstract heap.

In practice, managing this locking and unlocking and the inevitable type casts can be difficult. It is usually easier to design a class than can work with an abstract heap to bring things in and out of memory automatically. Indeed, this is what has been done with class `RWTValVirtualArray<T>`, which represents a virtual array of elements of type T. Elements are automatically swapped in as necessary as they are addressed.

Example

This example illustrates adding N nodes to a linked list. In this linked list, a "pointer" to the next node is actually a handle.

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

struct Node {
  int  key;
  RWHandlenext;
};

RWHandle head = 0;

void addNodes(RWVirtualPageHeap& heap, unsigned N) {
  for (unsigned i=0; i<N; i++){
    RWHandle h = heap.allocate();
    Node* newNode = (Node*)heap.lock(h);
    newNode->key  = i;
    newNode->next = head;
    head = h;
    heap.dirty(h);
    heap.unlock(h);
  }
}
```

Public constructor

`RWVirtualPageHeap(unsigned pgsize);`
Sets the size of a page.

Public destructor

`virtual                    ~RWVirtualPageHeap();`
The destructor has been made virtual to give specializing classes a chance to deallocate any resources that they may have allocated.

Public member functions

`unsigned                   pageSize() const;`
Returns the page size for this abstract page heap.

Public pure virtual functions

```
virtual RWHandle           allocate() = 0
```
Allocates a page off the abstract heap and returns a handle for it. If the specializing class is unable to honor the request, then it should return a zero handle.

```
virtual void               deallocate(RWHandle h) = 0;
```
Dallocate the page associated with handle `h`. It is not an error to deallocate a zero handle.

```
virtual void               dirty(RWHandle h) = 0;
```
Declare the page associated with handle `h` to be "dirty". That is, it has changed since it was last locked. The page must be locked before calling this function.

```
virtual void*              lock(RWHandle h) = 0;
```
Lock the page, swapping it into physical memory, and return an address for it. A nil pointer will be returned if the specializing class is unable to honor the lock. The returned pointer should be regarded as pointing to a buffer of the page size.

```
virtual void               unlock(RWHandle h) = 0;
```
Unlock a page. A page must be locked before calling this function. After calling this function the address returned by `lock()` is no longer valid.

RWvistream

RWvistream
|
RWvios

Synopsis

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

Description

Class `RWvistream` is an abstract base class. It provides an interface for format-independent retrieval of primitives and arrays of primitives. Its counterpart, `RWvostream`, provides a complementary interface for the storage of these variables.

Because the interface of `RWvistream` and `RWvostream` is independent of formatting, the user of these classes need not be concerned with how variables will actually be stored or restored. That will be up to the derived class to decide. It might be done using an operating-system independent ASCII format (classes `RWpistream` and `RWpostream`), a binary format (classes `RWbistream` and `RWbostream`), or the user could define his or her own format (*e.g.*, an interface to a network). Note that because it is an abstract base class, there is no way to actually enforce these goals—the description here is merely the model of how a class derived from `RWvistream` and `RWvostream` should act.

See class `RWvostream` for additional explanations and examples of format-independent stream storage.

Example

```
#include <rw/vstream.h>
void restoreStuff( RWvistream& str)
{
  int i;
  double d;
  char string[80];
  str >> i;   // Restore an int
  str >> d;   // Restore a double
  // Restore a character string, up to 80 characters long:
  str.getString(string, sizeof(string));

  if(str.fail()) cerr << "Oh, oh, bad news.\n";
}
```

Public member functions

`virtual int                get() = 0;`
Get and return the next char from the input stream, returning its value. Returns EOF if end of file is encountered.

`virtual RWvistream&        get(char& c) = 0;`
Get the next char from the input stream, returning its value in `c`.

`virtual RWvistream&        get(wchar_t& wc) = 0;`
Get the next wide char from the input stream, returning its value in `wc`.

`virtual RWvistream&        get(unsigned char& c) = 0;`
Get the next char from the input stream, returning its value in `c`.

`virtual RWvistream&        get(char* v, size_t N) = 0;`
Get a vector of char's and store then in the array beginning at `v`. If the restore is stopped prematurely, get stores whatever it can in `v`, and sets the failbit.

Note – The vector is treated as a vector of numbers, not characters. If you wish to restore a character string, use function `getString(char*, size_t)`.

`virtual RWvistream&        get(wchar_t* v, size_t N) = 0;`
Get a vector of wide char's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

Note – The vector is treated as a vector of numbers, not characters. If you wish to restore a character string, use function `getString(wchar_t*, size_t)`.

```
virtual RWvistream&       get(double* v, size_t N) = 0;
```
Get a vector of double's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(float* v, size_t N) = 0;
```
Get a vector of float's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(int* v, size_t N) = 0;
```
Get a vector of int's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(long* v, size_t N) = 0;
```
Get a vector of long's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(short* v, size_t N) = 0;
```
Get a vector of short's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned char* v, size_t
                           N) = 0;
```
Get a vector of unsigned char's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

Note – The vector is treated as a vector of numbers, not characters. If you wish to restore a character string, use function `getString(char*, size_t)`.

```
virtual RWvistream&       get(unsigned short* v, size_t
                           N) = 0;
```
Get a vector of unsigned short's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned int* v, size_t N)
                           = 0;
```
Get a vector of unsigned int's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       get(unsigned long* v, size_t
                           N) = 0;
```
Get a vector of unsigned long's and store then in the array beginning at `v`. If the restore is stopped prematurely, `get` stores whatever it can in `v`, and sets the failbit.

```
virtual RWvistream&       getString(char* s, size_t N)
                           = 0;
```
Restores a character string from the input stream and stores it in the array beginning at `s`. The function stops reading at the end of the string or after N–1 characters, whichever comes first. If the latter, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte.

```
virtual RWvistream&       getString(wchar_t* ws, size_t N)
                           = 0;
```
Restores a wide character string from the input stream and stores it in the array beginning at `ws`. The function stops reading at the end of the string or after `N-1` characters, whichever comes first. If the latter, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte.

```
virtual RWvistream&       operator>>(char& c) = 0;
```
Get the next character from the input stream and store it in `c`.

```
virtual RWvistream&       operator>>(wchar_t& wc) = 0;
```
Get the next wide character from the input stream and store it in `wc`.

```
virtual RWvistream&       operator>>(double& d) = 0;
```
Get the next double from the input stream and store it in `d`.

```
virtual RWvistream&       operator>>(float& f) = 0;
```
Get the next float from the input stream and store it in `f`.

```
virtual RWvistream&       operator>>(int& i) = 0;
```
Get the next int from the input stream and store it in `i`.

```
virtual RWvistream&       operator>>(long& l) = 0;
```
Get the next long from the input stream and store it in `l`.

`virtual RWvistream&       operator>>(short& s) = 0;`
Get the next short from the input stream and store it in `s`.

`virtual RWvistream&      operator>>(unsigned char& c) = 0;`
Get the next unsigned char from the input stream and store it in `c`.

`virtual RWvistream&      operator>>(unsigned short& s) = 0;`
Get the next unsigned short from the input stream and store it in `s`.

`virtual RWvistream&      operator>>(unsigned int& i) = 0;`
Get the next unsigned int from the input stream and store it in `i`.

`virtual RWvistream&      operator>>(unsigned long& l) = 0;`
Get the next unsigned long from the input stream and store it in `l`.

22

RWvostream

```
RWvostream
    |
  RWv ios
```

Synopsis `#include <rw/vstream.h>`

Description

Class `RWvostream` is an abstract base class. It provides an interface for format-independent storage of primitives and arrays of primitives. Its counterpart, `RWvistream`, provides a complementary interface for the retrieval of these variables.

Because the interface of `RWvistream` and `RWvostream` is independent of formatting, the user of these classes need not be concerned with how variables will actually be stored or restored. That will be up to the derived class to decide. It might be done using an operating-system independent ASCII format (classes `RWpistream` and `RWpostream`), a binary format (classes `RWbistream` and `RWbostream`), or the user could define his or her own format (*e.g.*, an interface to a network).

Note – Because it is an *abstract* base class, there is no way to actually enforce these goals—the description here is merely the model of how a class derived from `RWvistream` and `RWvostream` should act.

Note – There is no need to separate variables with whitespace. It is the responsibility of the derived class to delineate variables with whitespace, packet breaks, or whatever might be appropriate for the final output sink.

The model is one where variables are inserted into the output stream, either individually or as homogeneous vectors, to be restored in the same order using `RWvistream`.

Storage and retrieval of characters requires some explanation. Characters can be thought of as either representing some alphanumeric or control character, or as the literal number. Generally, the overloaded lshift (<<) and rshift (>>) operators seek to store and restore characters preserving their symbolic meaning. *I.e.*, storage of a newline should be restored as a newline, regardless of its representation on the target machine. By contrast, member functions `get()` and `put()` should treat the character as a literal number, whose value is to be preserved. See also class `RWpostream`.

Example

```
#include <rw/vstream.h>
void storeStuff( RWvostream& str)
{
  int i = 5;
  double d = 22.5;
  char string[] = "A string with \t tabs and a newline\n";
  str << i;// Store an int
  str << d;// Store a double
  str << string;// Store a string

  if(str.fail()) cerr << "Oh, oh, bad news.\n";
}
```

Public operators

`virtual RWvostream&        operator<<(const char* s) = 0;`
Store the character string starting at `s` to the output stream. The character string is expected to be null terminated.

`virtual RWvostream&        operator<<(const wchar_t* ws) = 0;`
Store the wide character string starting at `ws` to the output stream. The character string is expected to be null terminated.

`virtual RWvostream&        operator<<(char c) = 0;`
Store the char `c` to the output stream.

Note – `c` is treated as a character, not a number.

`virtual RWvostream&        operator<<(wchar_t wc) = 0;`
Store the wide char `wc` to the output stream.

Note – `wc` is treated as a character, not a number.

`virtual RWvostream&       operator<<(unsigned char c) = 0;`
Store the unsigned char `c` to the output stream.

Note – `c` is treated as a character, not a number.

`virtual RWvostream&       operator<<(double d) = 0;`
Store the double `d` to the output stream.

`virtual RWvostream&       operator<<(float f) = 0;`
Store the float `f` to the output stream.

`virtual RWvostream&       operator<<(int i) = 0;`
Store the int `i` to the output stream.

`virtual RWvostream&       operator<<(unsigned int i) = 0;`
Store the unsigned int `i` to the output stream.

`virtual RWvostream&       operator<<(long l) = 0;`
Store the long `l` to the output stream.

`virtual RWvostream&       operator<<(unsigned long l) = 0;`
Store the unsigned long `l` to the output stream.

`virtual RWvostream&       operator<<(short s) = 0;`
Store the short `s` to the output stream.

```
virtual RWvostream&       operator<<(unsigned short s)
                           = 0;
```
Store the unsigned short `s` to the output stream.

Public member functions

`virtual RWvostream&put(char c) = 0;`
Store the char `c` to the output stream, preserving its value.

`virtual RWvostream&       put(wchar_t wc) = 0;`
Store the wide char `wc` to the output stream, preserving its value.

`virtual RWvostream&       put(unsigned char c) = 0;`
Store the char `c` to the output stream, preserving its value.

```
virtual RWvostream&       put(const char* p, size_t N)
                           = 0;
```
Store the vector of chars starting at `p` to the output stream. The chars should be treated as literal numbers (*i.e.*, not as a character string).

```
virtual RWvostream&       put(const wchar_t* p, size_t N)
                           = 0;
```
Store the vector of wide chars starting at `p` to the output stream. The chars should be treated as literal numbers (*i.e.*, not as a character string).

```
virtual RWvostream&       put(const unsigned char* p,
                           size_t N) = 0;
```
Store the vector of unsigned chars starting at `p` to the output stream. The chars should be treated as literal numbers (*i.e.*, not as a character string).

```
virtual RWvostream&       put(const short* p, size_t N)
                           = 0;
```
Store the vector of shorts starting at `p` to the output stream.

```
virtual RWvostream&       put(const unsigned short* p,
                           size_t N) = 0;
```
Store the vector of unsigned shorts starting at `p` to the output stream.

```
virtual RWvostream&       put(const int* p, size_t N)
                           = 0;
```
Store the vector of ints starting at `p` to the output stream.

```
virtual RWvostream&       put(const unsigned int* p,
                           size_t N) = 0;
```
Store the vector of unsigned ints starting at `p` to the output stream.

```
virtual RWvostream&       put(const long* p, size_t N)
                           = 0;
```
Store the vector of longs starting at `p` to the output stream.

```
virtual RWvostream&       put(const unsigned long* p,
                           size_t N) = 0;
```
Store the vector of unsigned longs starting at `p` to the output stream.

```
virtual RWvostream&       put(const float* p, size_t N)
                           = 0;
```
Store the vector of floats starting at `p` to the output stream.

```
virtual RWvostream&       put(const double* p, size_t N)
                           = 0;
```
Store the vector of doubles starting at `p` to the output stream.

22

RWWString

Note **Solaris 2.*x* systems only.**

Synopsis

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

RWWString a;
```

Description

Class `RWWString` offers very powerful and convenient facilities for manipulating wide character strings.

This string class manipulates *wide characters* of the Standard C type `wchar_t`. These characters are generally two or four bytes, and can be used to encode richer code sets than the classic `"char"` type. Because `wchar_t` characters are all the same size, indexing is fast.

Conversion to and from multibyte and ASCII forms are provided by the `RWWString` constructors, and by the `RWWString` member functions `isAscii(), toAscii(), and toMultiByte()`.

Stream operations implicitly translate to and from the multibyte stream representation. That is, on output, wide character strings are converted into multibyte strings, while on input they are converted back into wide character strings. Hence, the external representation of wide character strings is usually as multibyte character strings, saving storage space and making interfaces with devices (which usually expect multibyte strings) easier.

Class `RWWString` tolerates embedded nulls.

Note – Parameters of type `const wchar_t*` must not be passed a value of zero. This is detected in the debug version of the library.

The class is implemented using a technique called *copy on write*. With this technique, the copy constructor and assignment operators still reference the old object and hence are very fast. An actual copy is made only when a “write” is performed, that is if the object is about to be changed. The net result is excellent performance, but with easy-to-understand copy semantics.

A separate `RWWSubString` class supports substring extraction and modification operations.

Example

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

main()
{
  RWWString a(L"There is no joy in Beantown.");
  a.subString(L"Beantown") = L"Redmond";
  cout << a << endl;
  return 0;
}
```

Program output:

```
There is no joy in Redmond.
```

Enumerations

`enum RWWString::caseCompare { exact, ignoreCase };`
Used to specify whether comparisons, searches, and hashing functions should use case sensitive (`exact`) or case-insensitive (`ignoreCase`) semantics.

`enum RWWString::multiByte_ { multiByte };`
Allows conversion from multibyte character strings to wide character strings. See constructor below.

`enum RWWString::ascii_ {ascii };`
Allows conversion from Ascii character strings to wide character strings. See constructor below.

Public constructors

`RWWString();`
Creates a string of length zero (the null string).

`RWWString(const wchar_t* cs);`
Conversion from the null-terminated character string `cs`. The created string will copy the data pointed to by `cs`, up to the first terminating null.

`RWWString(const wchar_t* cs, size_t N);`
Constructs a string from the character string `cs`. The created string will copy the data pointed to by `cs`. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long.

`RWWString(RWSize_T ic)`
Creates a string of length zero (the null string). The strings *capacity* (that is, the size it can grow to without resizing) is given by the parameter `ic`.

`RWWString(const RWWString& str);`
Copy constructor. The created string will *copy* `str`'s data.

`RWWString(const RWWSubString& ss);`
Conversion from sub-string. The created string will *copy* the substring represented by `ss`.

`RWWString(wchar_c c);`
Constructs a string containing the single character `c`.

`RWWString(wchar_c c, size_t N);`
Constructs a string containing the character `c` repeated `N` times.

`RWWString(const char* mbcs, multiByte_ mb);`
Construct a wide character string from the multibyte character string contained in `mbcs`. The conversion is done using the Standard C library function `::mbstowcs()`. This constructor can be used as follows:

```
    RWWString a("\306\374\315\313\306\374", multiByte);
```

`RWWString(const char* acs, ascii_ asc);`
Construct a wide character string from the ASCII character string contained in `acs`. The conversion is done with the assumption that the given string contains only ASCII characters, so it is much faster than the more general constructor given above. For this conversion to be successful, you must be certain that the string contains only ASCII characters. This can be confirmed (if necessary) using `RWCString::isAscii()`. This constructor can be used as follows:

```
    RWWString a("An ASCII charater string", ascii);
```

`RWWString(const char* cs, size_t N, multiByte_ mb);`
`RWWString(const char* cs, size_t N, ascii__ asc);`
These two constructors are similar to the two constructors immediately above except that they copy exactly `N` characters, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long.

`RWWString(const RWCString& cs, multiByte_ mb);`
Construct a wide character string from the multibyte string contained in `cs`.

`RWWString(const RWCString& cs, ascii_ asc);`
Construct a wide character string from the ASCII string contained in `cs`. The conversion is done with the assumption that the given string contains only ASCII characters, so it is much faster than the more general constructor above.

Type conversion

`operator const wchar_t*() const;`
Access to the `RWWString`'s data as a null terminated string. This datum is owned by the `RWWString` and may not be deleted or changed. If the `RWWString` object itself changes or goes out of scope, the pointer value previously returned may (will!) become invalid. While the string is null-terminated, note that its *length* is still given by the member function `length()`. That is, it may contain embedded nulls.

Assignment operators

`RWWString& operator=(const wchar_t* cs);`
Assignment operator. Copies the null-terminated character string pointed to by `cs` into self. Returns a reference to self.

`RWWString& operator=(const RWWString& str);`
Assignment operator. The string will *copy* `str`'s data. Returns a reference to self.

`RWWString& operator+=(const wchar_t* cs);`
Append the null-terminated character string pointed to by `cs` to self. Returns a reference to self.

`RWWString& operator+=(const RWWString& str);`
Append the string `str` to self. Returns a reference to self.

Indexing operators

`wchar_t& operator[](size_t i);`
`wchar_t operator[](size_t i) const;`
Return the `i`'th character. The first variant can be used as an lvalue. The index

`i` must be between 0 and the length of the string less one. Bounds checking is performed—if the index is out of range then an exception of type `RWBoundsErr` will be thrown.

```
wchar_t&            operator()(size_t i);
wchar_t             operator()(size_t i) const;
```
Return the i'th character. The first variant can be used as an lvalue. The index `i` must be between 0 and the length of the string less one. Bounds checking is performed if the pre-processor macro `RWBOUNDS_CHECK` has been defined before including `<rw/wstring.h>`. In this case, if the index is out of range, then an exception of type `RWBoundsErr` will be thrown.

```
RWWSubString        operator()(size_t start, size_t len);
const RWWSubString  operator()(size_t start, size_t len)
                     const;
```
Substring operator. Returns an `RWWSubString` of self with length `len`, starting at index `start`. The first variant can be used as an lvalue. The sum of `start` plus `len` must be less than or equal to the string length. If the library was built using the `RWDEBUG` flag, and `start` and `len` are out of range, then an exception of type `RWBoundsErr` will be thrown.

Public member functions

```
RWWString&          append(const wchar_t* cs);
```
Append a copy of the null-terminated character string pointed to by `cs` to self. Returns a reference to self.

```
RWWString&          append(const wchar_t* cs, size_t N);
```
Append a copy of the character string `cs` to self. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long. Returns a reference to self.

```
RWWString&          append(const RWWString& cstr);
```
Append a copy of the string cstr to self. Returns a reference to self.

```
RWWString&          append(const RWWString& cstr, size_t
                     N);
```
Append the first `N` characters or the length of `cstr` (whichever is less) of `cstr` to self. Returns a reference to self.

```
size_t              binaryStoreSize() const;
```
Returns the number of bytes necessary to store the object using the global function
```
    RWFile& operator<<(RWFile&, const RWWString&);
```

```
size_t                capacity() const;
```
Return the current capacity of self. This is the number of characters the string can hold without resizing.

```
size_t                capacity(size_t capac);
```
Hint to the implementation to change the capacity of self to `capac`. Returns the actual capacity.

```
int                   collate(const RWWString& str) const;
int                   collate(const wchar_t*   str) const;
```
Returns an int less then, greater than, or equal to zero, according to the result of calling the `POSIX` function `::wscoll()` on self and the argument `str`. This supports locale-dependent collation.

```
int                   compareTo(const RWWString& str,
                         caseCompare = exact) const;
int                   compareTo(const wchar_t*   str,
                         caseCompare = exact) const;
```
Returns an int less than, greater than, or equal to zero, according to the result of calling the Standard C library function `::memcmp()` on self and the argument `str`. Case sensitivity is according to the `caseCompare` argument, and may be `RWWString::exact or RWCString::ignoreCase`.

```
RWBoolean             contains(const RWWString& cs,
                         caseCompare = exact) const;
RWBoolean             contains(const wchar_t* str,
                         caseCompare = exact) const;
```
Pattern matching. Returns `TRUE` if `cs` occurs in self. Case sensitivity is according to the `caseCompare` argument, and may be `RWWString::exact or RWWString::ignoreCase`.

```
const wchar_t*        data() const;
```
Access to the `RWWString`'s data as a null terminated string. This datum is owned by the `RWWString` and may not be deleted or changed. If the `RWWString` object itself changes or goes out of scope, the pointer value previously returned may (will!) become invalid. While the string is null-terminated, note that its length is still given by the member function `length()`. That is, it may contain embedded nulls. .

```
size_t                first(wchar_t c) const;
```
Returns the index of the first occurence of the character c in self. Returns `RW_NPOS` if there is no such character.

```
unsigned             hash(caseCompare = exact) const;
```
Returns a suitable hash value.

```
size_t               index(const wchar_t* pat, size_t
                        i=0, caseCompare = exact) const;
size_t               index(const RWWString& pat, size_t
                        i=0, caseCompare = exact) const;
```
Pattern matching. Starting with index i, searches for the first occurrence of `pat` in self and returns the index of the start of the match. Returns `RW_NPOS` if there is no such pattern. Case sensitivity is according to the `caseCompare` argument; it defaults to `RWWString::exact`.

```
size_t               index(const wchar_t* pat, size_t
                        patlen, size_t i, caseCompare)
                        const;
size_t               index(const RWWString& pat, size_t
                        patlen, size_t i, caseCompare)
                        const;
```
Pattern matching. Starting with index `i`, searches for the first occurrence of the first `patlen` characters from `pat` in self and returns the index of the start of the match. Returns `RW_NPOS` if there is no such pattern. Case sensitivity is according to the `caseCompare` argument.

```
RWWString&           insert(size_t pos, const wchar_t* cs);
```
Insert a copy of the null-terminated string `cs` into self at position `pos`. Returns a reference to self.

```
RWWString&           insert(size_t pos, const wchar_t* cs,
                        size_t N);
```
Insert a copy of the first `N` characters of `cs` into self at position `pos`. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long. Returns a reference to self.

```
RWWString&           insert(size_t pos, const RWWString&
                        str);
```
Insert a copy of the string `str` into self at position `pos`. Returns a reference to self.

```
RWWString&           insert(size_t pos, const RWWString&
                        str, size_t N);
```
Insert a copy of the first `N` characters or the length of `str` (whichever is less) of str into self at position `pos`. Returns a reference to self.

```
RWBoolean             isAscii() const;
```
Returns `TRUE` if it is safe to perform the conversion `toAscii()` (that is, if all characters of self are ASCII characters).

```
RWBoolean             isNull() const;
```
Returns `TRUE` if this is a zero lengthed string (*i.e.*, the null string).

```
size_t                last(wchar_t c) const;
```
Returns the index of the last occurrence in the string of the character `c`. Returns `RW_NPOS` if there is no such character.

```
size_t                length() const;
```
Return the number of characters in self.

```
RWWString&            prepend(const wchar_t* cs);
```
Prepend a copy of the null-terminated character string pointed to by `cs` to self. Returns a reference to self.

```
RWWString&            prepend(const wchar_t* cs, size_t N,);
```
Prepend a copy of the character string `cs` to self. Exactly `N` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N` bytes long. Returns a reference to self.

```
RWWString&            prepend(const RWWString& str);
```
Prepends a copy of the string `str` to self. Returns a reference to self.

```
RWWString&            prepend(const RWWString& cstr, size_t
                         N);
```
Prepend the first `N` characters or the length of `cstr` (whichever is less) of `cstr` to self. Returns a reference to self.

```
istream&              readFile(istream& s);
```
Reads characters from the input stream `s`, replacing the previous contents of self, until EOF is reached. The input stream is treated as a sequence of multibyte characters, each of which is converted to a wide character (using the Standard C library function `mbtowc()`) before storing. Null characters are treated the same as other characters.

```
istream&              readLine(istream& s, RWBoolean
                         skipWhite = TRUE);
```
Reads characters from the input stream `s`, replacing the previous contents of self, until a newline (or an EOF) is encountered. The newline is removed from the input stream but is not stored. The input stream is treated as a sequence of multibyte characters, each of which is converted to a wide character (using the

Standard C library function `mbtowc()`) before storing. Null characters are treated the same as other characters. If the `skipWhite` argument is `TRUE`, then whitespace is skipped (using the iostream library manipulator ws) before saving characters.

```
istream&             readString(istream& s);
```
Reads characters from the input stream `s`, replacing the previous contents of self, until an EOF or null terminator is encountered. The input stream is treated as a sequence of multibyte characters, each of which is converted to a wide character (using the Standard C library function `mbtowc()`) before storing.

```
istream&             readToDelim(istream&, wchar_t
                        delim=(wchar_t)'\n');
```
Reads characters from the input stream `s`, replacing the previous contents of self, until an `EOF` or the delimiting character `delim` is encountered. The delimiter is removed from the input stream but is not stored. The input stream is treated as a sequence of multibyte characters, each of which is converted to a wide character (using the Standard C library function `mbtowc()`) before storing. Null characters are treated the same as other characters.

```
istream&             readToken(istream& s);
```
Whitespace is skipped before saving characters. Characters are then read from the input stream `s`, replacing previous contents of self, until trailing whitespace or an EOF is encountered. The whitespace is left on the input stream. Only ASCII whitespace characters are recognized, as defined by the standard C library function `isspace()`. The input stream is treated as a sequence of multibyte characters, each of which is converted to a wide character (using the Standard C library function `mbtowc()`) before storing.

```
RWWString&           remove(size_t pos);
```
Removes the characters from the position `pos` to the end of string. Returns a reference to self.

```
RWWString&           remove(size_t pos, size_t N);
```
Removes `N` characters or to the end of string (whichever comes first) starting at the position `pos`. Returns a reference to self.

```
RWWString&           replace(size_t pos, size_t N, const
                        wchar_t* cs);
```
Replaces N characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the null-terminated string `cs`. Returns a reference to self.

```
RWWString&           replace(size_t pos, size_t N1,
                        const wchar_t* cs, size_t N2);
```
Replaces `N1` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the string `cs`. Exactly `N2` characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N2` bytes long. Returns a reference to self.

```
RWWString&           replace(size_t pos, size_t N, const
                        RWWString& str);
```
Replaces `N` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the string `str`. Returns a reference to self.

```
RWWString&           replace(size_t pos, size_t N1,
                        const RWWString& str, size_t N2);
```
Replaces `N1` characters or to the end of string (whichever comes first) starting at position `pos` with a copy of the first `N2` characters, or the length of `str` (whichever is less), from `str`. Returns a reference to self.

```
void                 resize(size_t n);
```
Changes the length of self, adding blanks (*i.e.*, `W' '`) or truncating as necessary.

```
RWWSubString         strip(stripType s = trailing,
                        wchar_t c=L' ');
```
Returns a substring of self where the character `c` has been stripped off the beginning, end, or both ends of the string. The enum `stripType` can take values:

stripType	**Meaning**
leading	Remove characters at beginning
trailing	Remove characters at end
both	Remove characters at both ends

```
RWWSubString         subString(const wchar_t* cs,size_t
                        start=0, caseCompare=exact);
const RWWSubString   subString(const wchar_t* cs, size_t
                        start=0, caseCompare=exact) const;
```

Returns a substring representing the first occurence of the null-terminated string pointed to by "`cs`". Case sensitivity is according to the `caseCompare` argument; it defaults to `RWWString::exact`.

```
RWCString             toAscii() const;
```
Returns an `RWCString` object of the same length as self, containing only ASCII characters. Any non-ASCII characters in self simply have the high bits stripped off. Use `isAscii()` to determine whether this function is safe to use.

```
RWCString             toMultiByte() const;
```
Returns an `RWCString` containing the result of applying the standard C library function `wcstombs()` to self. This function is always safe to use.

```
void                  toLower();
```
Changes all upper-case letters in self to lower-case. Uses the C library function `towlower()`.

```
void                  toUpper();
```
Changes all lower-case letters in self to upper-case. Uses the C library function `towupper()`.

Static public member functions

```
static size_t         initialCapacity(size_t ic = 15);
```
Sets the minimum initial capacity of an `RWWString`, and returns the old value. The initial setting is 15 characters. Larger values will use more memory, but result in fewer resizes when concatenating or reading strings. Smaller values will waste less memory, but result in more resizes.

```
static size_t              maxWaste(size_t mw = 15);
```
Sets the maximum amount of unused space allowed in a string should it shrink, and returns the old value. The initial setting is 15 characters. If more than `mw` characters are wasted, then excess space will be reclaimed.

```
static size_t              resizeIncrement(size_t ri = 16);
```
Sets the resize increment when more memory is needed to grow a string. Returns the old value. The initial setting is 16 characters.

It's not safe to modify `initialCapacity`, `maxWaste`, or `resizeIncrement` when more than one thread is present.

Related global operators

```
RWBoolean             operator==(const RWWString&,  const
                         wchar_t*   );
RWBoolean             operator==(const wchar_t*,    const
                          RWWString&);
RWBoolean              operator==(const RWWString&,  const
                          RWWString&);
RWBoolean              operator!=(const RWWString&,  const
                          wchar_t*   );
RWBoolean              operator!=(const wchar_t*,    const
                          RWWString&);
RWBoolean              operator!=(const RWWString&,  const
                          RWWString&);
```
Logical equality and inequality. Case sensitivity is *exact*.

```
RWBoolean              operator< (const RWWString&, const
                          wchar_t*   );
RWBoolean              operator< (const wchar_t*,   const
                          RWWString&);
RWBoolean              operator< (const RWWString&, const
                          RWWString&);
RWBoolean              operator> (const RWWString&, const
                          wchar_t*   );
RWBoolean              operator> (const wchar_t*,   const
                          RWWString&);
RWBoolean              operator> (const RWWString&, const
                          RWWString&);
RWBoolean              operator<= (const RWWString&, const
                          wchar_t*   );
RWBoolean              operator<= (const wchar_t*,   const
                          RWWString&);
RWBoolean              operator<= (const RWWString&, const
                          RWWString&);
RWBoolean              operator>= (const RWWString&, const
                          wchar_t*   );
RWBoolean              operator>= (const wchar_t*,   const
                          RWWString&);
RWBoolean              operator>= (const RWWString&, const
                          RWWString&);
```
Comparisons are done lexicographically, byte by byte. Case sensitivity is *exact*. Use member `collate()` or `strxfrm()` for locale sensitivity.

```
RWWString              operator+(const RWWString&,   const
                          RWWString&);
RWWString              operator+(const wchar_t*,     const
                          RWWString&);
RWWString             operator+(const RWWString&,   const
                          wchar_t*   );
```
Concatenation operators.

```
ostream&              operator<<(ostream& s,        const
                                 RWWString& str);
```
Output an `RWWString` on ostream `s`. Each character of `str` is first converted to a multibyte character before being shifted out to `s`.

```
istream&              operator>>(istream& s, RWWString&
                                 str);
```
Calls `str.readToken(s)`. That is, a token is read from the input stream `s`.

```
RWvostream&           operator<<(RWvostream&,       const
                                 RWWString& str);
RWFile&               operator<<(RWFile&,           const
                                 RWWString& str);
```
Saves string `str` to a virtual stream or `RWFile`, respectively.

```
RWvistream&           operator>>(RWvistream&, RWWString&
                                 str);
RWFile&               operator>>(RWFile&,      RWWString&
                                 str);
```
Restores a wide character string into `str` from a virtual stream or `RWFile`, respectively, replacing the previous contents of `str`.

Related global functions

```
RWWString             strXForm(const RWWString&);
```
Returns a string transformed by `::wsxfrm()`, to allow quicker collation than `RWWString::collate()`.

```
RWWString             toLower(const RWWString& str);
```
Returns a version of `str` where all upper-case characters have been replaced with lower-case characters. Uses the C library function `towlower()`.

```
RWWString             toUpper(const RWWString& str);
```
Returns a version of `str` where all lower-case characters have been replaced with upper-case characters. Uses the C library function `towupper()`.

RWWSubString

Note

Solaris 2.*x* systems only.

Synopsis

```
#include <rw/wstring.h>
RWWString s(L"test string");
s(6,3);                    // "tri"
```

Description

The class `RWWSubString` allows some subsection of a `RWWString` to be addressed by defining a *starting position* and an *extent*. For example the 7'th through the 11'th elements, inclusive, would have a starting position of 7 and an extent of 5. The specification of a starting position and extent can also be done in your behalf by such functions as `RWWString::strip()` or the overloaded function call operator taking a regular expression as an argument. There are no public constructors—`RWWSubStrings` are constructed by various functions of the `RWWString` class and then destroyed immediately.

A *zero lengthed* substring is one with a defined starting position and an extent of zero. It can be thought of as starting just before the indicated character, but not including it. It can be used as an lvalue. A null substring is also legal and is frequently used to indicate that a requested substring, perhaps through a search, does not exist. A null substring can be detected with member function `isNull()`. However, it cannot be used as an lvalue.

Example

```
#include <rw/rstream.h>
#include <rw/wstring.h>
main()
{
  RWWString s(L"What I tell you is true.");
  // Create a substring and use it as an lvalue:
  s(19,0) = RWWString(L"three times");
  cout << s << endl;
  return 0;
}
```

Program output:

```
What I tell you is three times true.
```

Assignment operators

```
void                      operator=(const RWWString&);
```

Assignment to a `RWWString`. The statements:

```
RWWString a;
RWWString b;
...
b(2, 3) = a;
```

will copy a's data into the substring b(2,3). The number of elements need not match: if they differ, b will be resized appropriately. If self is the null substring, then the statement has no effect.

```
void                      operator=(const wchar_t*);
```

Assignment from a wide character string. Example:

```
    RWWString a(L"Mary had a little lamb");
    wchar_t dat[] = L"Perrier";
    a(11,4) = dat;     // "Mary had a Perrier"
```

Note – the number of characters selected need not match: if they differ, a will be resized appropriately. If self is the null substring, then the statement has no effect.

Indexing operators

```
wchar_t             operator[](size_t i);
cchar_t&            operator[](size_t i) const;
```

Returns the i'th character of the substring. The first variant can be used as an lvalue, the second cannot. The index i must be between zero and the length of the substring, less one. Bounds checking is performed: if the index is out of range, then an exception of type `RWBoundsErr` will be thrown.

```
wchar_t             operator()(size_t i);
wchar_t&            operator()(size_t i) const;
```

Returns the i'th character of the substring. The first variant can be used as an lvalue, the second cannot. The index i must be between zero and the length of the substring, less one. Bounds checking is enabled by defining the pre-processor macro `RWBOUNDS_CHECK` before including <`rw/wstring.h`>. In this case, if the index is out of range, then an exception of type `RWBoundsErr` will be thrown.

Public member functions

```
RWBoolean              isNull() const;
```
Returns TRUE if this is a null substring.

```
size_t                 length() const;
```
Returns the extent (*i.e.*, length) of the RWWSubString.

```
RWBoolean              operator!() const;
```
Returns TRUE if this is a null substring.

```
size_t                 start() const;
```
Returns the starting element of the RWWSubString.

```
void                   toLower();
```
Changes all upper-case letters in self to lower-case. Uses the C library function towlower().

```
void                   toUpper();
```
Changes all lower-case letters in self to upper-case. Uses the C library function towupper().

Global logical operators

```
RWBoolean              operator==(const RWWSubString&, const
                          RWWSubString&);
RWBoolean              operator==(const RWWString&,    const
                          RWWSubString&);
RWBoolean              operator==(const RWWSubString&, const
                          RWWString&    );
RWBoolean              operator==(const wchar_t*,      const
                          RWWSubString&);
RWBoolean              operator==(const RWWSubString&, const
                          wchar_t*      );
```
Returns TRUE if the substring is lexicographically equal to the character string or RWWString argument. Case sensitivity is *exact*.

```
RWBoolean              operator!=(const RWWString&     const
                          RWWString&    );
RWBoolean              operator!=(const RWWString&,    const
                          RWWSubString&);
RWBoolean              operator!=(const RWWSubString&, const
                          RWWString&    );
RWBoolean              operator!=(const wchar_t*,      const
                          RWWString&    );
```

```
RWBoolean           operator!=(const RWWString&,   const
                         wchar_t*      );
```

Returns the negation of the respective `operator==()`.

RWWTokenizer

Note

Solaris 2.*x* systems only.

Synopsis

```
#include <rw/wtoken.h>
RWWString str("a string of tokens", RWWString::ascii);
RWWTokenizer(str);      // Lex the above string
```

Description

Class `RWWTokenizer` is designed to break a string up into separate tokens, delimited by an arbitrary “white space”. It can be thought of as an iterator for strings and as an alternative to the C library function `wstok`() which has the unfortunate side effect of changing the string being tokenized.

Example

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

main()
{
   RWWString a(L"Something is rotten in the state of Denmark");

   RWWTokenizer next(a); // Tokenize the string a

   RWWString token; // Will receive each token

   // Advance until the null string is returned:
   while (!(token=next()).isNull())
          cout << token << "\n";
}
```

Program output:

```
Something
is
rotten
in
the
```

```
  state
  of
  Denmark
```

Public constructor

`RWWTokenizer(const RWWString& s);`
Construct a tokenizer to lex the string `s`.

Public member function

`RWWSubString      operator()(const wchar_t* s =L" \t\n");`
Advance to the next token and return it as a substring. The tokens are considered to be deliminated by any of the characters in `s`.

RWXDRistream

```
RWXDRistream
     |           |
RWvistream RWios
      |
 RWvios
```

Synopsis

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

XDR xdr;
xdrstdio_create(&xdr, stdin, XDR_DECODE);
RWXDRistream rw_xdr(&xdr);
```

Description

Class `RWXDRistream` is a portable input stream based on XDR routines. Class `RWXDRistream` encapsulates a portion of the XDR library routines that are used for external data representation. XDR routines allow programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls (RPC) are transmitted using XDR routines.

Class `RWXDRistream` enables one to decode an XDR structure to a machine representation. Class `RWXDRistream` provides the capability to decode all the standard data types and vectors of those data types.

An XDR stream must first be created by calling the appropriate creation routine. XDR streams currently exist for encoding/decoding of data to or from standard I/O FILE streams, TCP/IP connections and Unix files, and memory. These creation routines take arguments that are tailored to the specific properties of the stream. After the XDR stream has been created, it can then be used as the argument to the constructor for a `RWXDRistream` object.

`RWXDRistream` can be interrogated as to the status of the stream using member functions `bad()`, `clear()`, `eof()`, `fail()`, `good()`, and `rdstate()`. See `RWvistream` for more information on the semantics of individual member functions.

Example

The example that follows is a "reader" program that decodes an XDR structure from a FILE stream. The example for class `RWXDRostream` is the "writer" program that encodes the XDR structures onto the FILE stream.

Applications using XDR must link (on SunOS 5.x) with `libnsl`..

```
#include “rw/xdrstrea.h”
#include “rw/rstream.h”
#include <stdio.h>

main()
{
  XDR xdr;
  FILE* fp = fopen(“test”,”r+”);
  xdrstdio_create(&xdr, fp, XDR_DECODE);

  RWXDRistream rw_xdr(&xdr);

  int data;
  for(int i=0; i<10; ++i)
  {
    rw_xdr >> data;      // decode integer data
    if(data == i)
      cout << data << endl;
    else
      cout << “Bad input value” << endl;
  }
  fclose(fp);
}
```

Public constructors

`RWXDRistream(XDR* xp);`
Initialize an `RWXDRistream` from the XDR structure xp.

Public member functions

`virtual int                 get();`
Redefined from class `RWvistream`. Gets and return the next character from the XDR input stream. If the operation fails, it sets the failbit and returns EOF.

`virtual RWvistream&        get(char& c);`
Redefined from class `RWvistream`. Gets the next character from the XDR input stream and stores it in `c`. If the operation fails, it sets the failbit.

`virtual RWvistream&        get(wchar_t& wc);`
Redefined from class `RWvistream`. Gets the next wide character from the XDR input stream and stores it in `wc`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(unsigned char& c);
```
Redefined from class `RWvistream`. Gets the next unsigned character from the XDR input stream and stores it in `c`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(char* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of characters from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(unsigned char* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of unsigned characters from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(double* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of doubles from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(float* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of floats from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(int* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of integers from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(unsigned int* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of unsigned integers from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(long* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of longs from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(unsigned long* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of unsigned longs from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(short* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of shorts from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&       get(unsigned short* v, size_t N);
```
Redefined from class `RWvistream`. Gets a vector of unsigned shorts from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

`virtual RWvistream&        get(wchar_t* v, size_t N);`
Redefined from class `RWvistream`. Gets a vector of wide characters from the XDR input stream and stores them in `v`. If the operation fails, it sets the failbit.

```
virtual RWvistream&        getString(char* s, size_t
                            maxlen);
```
Redefined from class `RWvistream`. Restores a character string from the XDR input stream and stores them in the array starting at `s`. The function stops reading at the end of the string or after `maxlen-1` characters, whichever comes first. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(char& c );`
Redefined from class `RWvistream`. Gets the next character from the XDR input stream and stores it in `c`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(double& d);`
Redefined from class `RWvistream`. Gets the next double from the XDR input stream and stores it in `d`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(float& f);`
Redefined from class `RWvistream`. Gets the next float from the XDR input stream and stores it in `f`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(int&  i);`
Redefined from class `RWvistream`. Gets the next integer from the XDR input stream and stores it in `i`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(long& l);`
Redefined from class `RWvistream`. Gets the next long from the XDR input stream and stores it in `l`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(short& s);`
Redefined from class `RWvistream`. Gets the next short from the XDR input stream and stores it in `s`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(wchar_t& wc);`
Redefined from class `RWvistream`. Gets the next wide character from the XDR input stream and stores it in `wc`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(unsigned char& c);`
Redefined from class `RWvistream`. Gets the next unsigned character from the XDR input stream and stores it in `c`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(unsigned int& i);`
Redefined from class `RWvistream`. Gets the next unsigned integer from the XDR input stream and stores it in `i`. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(unsigned long& l);`
Redefined from class RWvistream. Gets the next unsigned long from the XDR input stream and stores it in l. If the operation fails, it sets the failbit.

`virtual RWvistream&        operator>>(unsigned short& s);`
Redefined from class `RWvistream`. Gets the next unsigned short from the XDR input stream and stores it in `s`. If the operation fails, it sets the failbit.

Related Global Functions

`int xdr(XDR* xp, RWCollectable*& cp);`
This can be passed to functions that expect an “XDR function”. The behavior depends on the direction of the XDR stream argument. For direction `XDR_ENCODE`, `cp` is transmitted via `xp` by `recursiveSaveOn`. For direction `XDR_DECODE`, an object or family of objects is allocated by calling `RWCollectable::recursiveRestoreFrom`, and a pointer to the object(s) is stored into cp. Direction `XDR_FREE` does nothing.

When restoring objects, this function allocates storage that the caller must free. You should cp to the null pointer before using this to restore objects. If built with `RWDEBUG`, the function will check this condition to help remind you to free the storage.

Example

This example shows how you might write an RPC server program using the function xdr. This receives and prints the string passed to it by an RPC client program.

```
#include <unistd.h>
#include <iostream.h>
#include <rpc/rpc.h>
#include <rw/xdrstream.h>
#include <rw/collstr.h>

const unsigned long DEMOPROG = 0x20000dad;
const unsigned long DEMOVERS = 1;
const unsigned long DEMOPROC = 1;
typedef void *(*rpcproc_t)(XDR*, void*);

extern “C” {
bool_t rpc_reg(const unsigned long, const unsigned
             long, const unsigned long,
             const rpcproc_t, const xdrproc_t,
             const xdrproc_t,
             const char *);
};

static RWCollectableString dummy(
  “Starting the object RPC server”);

int& demo(RWCollectable*& xp, svc_req&) {
  static int value;
  if (xp->isA() == __RWCOLLECTABLESTRING) {
    cout << (RWCollectableString&)*xp << endl;
    free xp; xp = 0;
    value = 1;
  }
  value = 0;
  return value;
}

main() {
  if (rpc_reg(DEMOPROG, DEMOVERS, DEMOPROC,
            (rpcproc_t)demo, (xdrproc_t)xdr,
            (xdrproc_t)xdr_int, “visible”) == -1) {
    cerr << “Couldn’t register\n”;
    exit(1);
  }

  cout << dummy << endl;

  svc_run();
  cerr << “Error: svc_run returned!\n”;
  exit(1);
  return 0;
}
```

RWXDRostream

```
RWXDRostream
    |        |
RWvostream RWios
    |
   RWv ios
```

Synopsis

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

XDR xdr;
xdrstdio_create(&xdr, stdout, XDR_ENCODE);
RWXDRostream rw_xdr(&xdr);
```

Description

Class `RWXDRostream` is a portable output stream based on XDR routines. Class `RWXDRostream` encapsulates a portion of the XDR library routines that are used for external data representation. XDR routines allow programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls (RPC) are transmitted using XDR routines.

Class `RWXDRostream` enables one to output from a stream and encode an XDR structure from a machine representation. Class `RWXDRostream` provides the capability to encode the standard data types and vectors of those data types.

An XDR stream must first be created by calling the appropriate creation routine. XDR streams currently exist for encoding/decoding of data to or from standard I/O FILE streams, TCP/IP connections and Unix files, and memory. These creation routines take arguments that are tailored to the specific properties of the stream. After the XDR stream has been created, it can then be used as an argument to the constructor for a `RWXDRostream` object.

`RWXDRostream` can be interrogated as to the status of the stream using member functions `bad()`, `clear()`, `eof()`, `fail()`, `good()`, and `rdstate()`. See the documentation of `RWostream` for further information on the semantics of individual functions. Applications that use XDR on SunOS must link with `libnsl`.

Example

The example that follows is a "writer" program that encodes an XDR structure onto a FILE stream. The example for class `RWXDRistream` is the "reader" program that decodes the XDR structures into a machine representation for a data type. The library that supports XDR routines must be linked in. The name of this library is not standard.

```
#include "rw/xdrstrea.h"
#include "rw/rstream.h"
#include <stdio.h>

main()
{
  XDR xdr;
  FILE* fp = fopen("test","w+");
  xdrstdio_create(&xdr, fp, XDR_ENCODE);

  RWXDRostream rw_xdr(&xdr);

  for(int i=0; i<10; ++i)
    rw_xdr << i;// encode integer data

  fclose(fp);
}
```

Public constructors

`RWXDRostream(XDR* xp);`
Initialize an `RWXDRostream` from the XDR structure xp.

Public member functions

`virtual RWvostream&        operator<<(const char* s);`
Redefined from class `RWvostream`. Store the character string starting at `s` to the output stream using the XDR format. The character string is expected to be null terminated.

`virtual RWvostream&        operator<<(char c);`
Redefined from class `RWvostream`. Store the character `c` to the output stream using the XDR format. Note that `c` is treated as a character, not a number.

`virtual RWvostream&        operator<<(wchar_t wc);`
Redefined from class `RWvostream`. Store the wide character `wc` to the output stream using the XDR format. Note that wc is treated as a character, not a number.

`virtual RWvostream&        operator<<(unsigned char c);`
Redefined from class `RWvostream`. Store the unsigned character `c` to the output stream using the XDR format. Note that `c` is treated as a character, not a number.

`virtual RWvostream&        operator<<(double d);`
Redefined from class `RWvostream`. Store the double `d` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(float f);`
Redefined from class `RWvostream`. Store the float `f` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(int i);`
Redefined from class `RWvostream`. Store the integer `i` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(unsigned int i);`
Redefined from class `RWvostream`. Store the unsigned integer `i` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(long l);`
Redefined from class `RWvostream`. Store the long `l` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(unsigned long l);`
Redefined from class `RWvostream`. Store the unsigned long `l` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(short s);`
Redefined from class `RWvostream`. Store the short `s` to the output stream using the XDR format.

`virtual RWvostream&        operator<<(unsigned short );`
Redefined from class `RWvostream`. Store the unsigned short `s` to the output stream using the XDR format.

`virtual RWvostream&        put(char c);`
Redefined from class `RWvostream`. Store the character `c` to the output stream using the XDR format. If the operation fails, it sets the failbit.

`virtual RWvostream&        put(unsigned char c);`
Redefined from class `RWvostream`. Store the unsigned character `c` to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(wchar_t wc);
```
Redefined from class RWvostream. Store the wide character wc to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const char* p, size_t N);
```
Redefined from class RWvostream. Store the vector of characters starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const wchar_t* p, size_t N);
```
Redefined from class RWvostream. Store the vector of wide characters starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const short* p, size_t N);
```
Redefined from class RWvostream. Store the vector of shorts starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const unsigned short* p,
                            size_t N);
```
Redefined from class RWvostream. Store the vector of unsigned shorts starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const int* p, size_t N);
```
Redefined from class RWvostream. Store the vector of integers starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const unsigned int* p,
                            size_t N);
```
Redefined from class RWvostream. Store the vector of unsigned integers starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const long* p, size_t N);
```
Redefined from class RWvostream. Store the vector of longs starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
virtual RWvostream&        put(const unsigned long* p,
                            size_t N);
```
Redefined from class RWvostream. Store the vector of unsigned longs starting at p to the output stream using the XDR format. If the operation fails, it sets the failbit.

`virtual RWvostream&        put(const float* p, size_t N);`
Redefined from class `RWvostream`. Store the vector of floats starting at `p` to the output stream using the XDR format. If the operation fails, it sets the failbit.

`virtual RWvostream&        put(const double* p, size_t N);`
Redefined from class `RWvostream`. Store the vector of doubles starting at `p` to the output stream using the XDR format. If the operation fails, it sets the failbit.

Related Global Functions

See class `RWXDRistream` for the function `xdr`.

Example

This example shows how you might write an RPC client program using the function `xdr` for output. This sends the string passed it on the command line to an RPC server.

```
#include <unistd.h>
#include <iostream.h>
#include <rpc/rpc.h>
#include <rw/collstr.h>
#include <rw/xdrstream.h>

const unsigned long DEMOPROG = 0x20000dad;
const unsigned long DEMOVERS = 1;
const unsigned long DEMOPROC = 1;
typedef void *(*rpcproc_t)(XDR*, void*);

main(int argc, char **argv) {
  clnt_stat status;
  static int value;
  static RWCollectable* arg;

  if (argc != 3) {
    fprintf(stderr, “Usage: rpcexam hostname msg”);
    exit(1);
  }

  arg = new RWCollectableString(argv[2]);

  if(rpc_call(argv[1], DEMOPROG, DEMOVERS, DEMOPROC,
              (xdrproc_t)xdr, (char*)&arg,
              (xdrproc_t)xdr_int, (char*)&value,
              “visible”) != RPC_SUCCESS) {
    clnt_perrno(status);
    return 1;
  }
  return 0;
}
```

RWZone

Synopsis

```
#include <time.h>
#include <rw/zone.h>
```

(abstract base class)

Description

RWZone is an abstract base class. It defines an interface for time zone issues such as whether or not daylight savings time is in use, the names and offsets from UTC (also known as GMT) for both standard and daylight savings times, and the start and stop dates for daylight savings time, if used.

Note – that because it is an abstract base class, there is no way to actually enforce these goals—the description here is merely the model of how a class derived from RWZone should act.

Most programs interact with RWZone only by passing an RWZone reference to an RWTime or RWDate member function that expects one.

RWZoneSimple is an implementation of the abstract RWZone interface sufficient to represent U.S.A. daylight savings time rules. Three instances of RWZoneSimple are initialized from the global environment at program startup to represent local, standard, and universal time. They are available via calls to the static member functions RWZone::local(), RWZone::standard(), and RWZone::utc(), respectively. See the class RWZoneSimple for details.

Example

```
#include <rw/zone.h>
#include <rw/rwtime.h>
#include <rw/rstream.h>

main() {
    RWTime now;
    cout << now.asString(RWZone::local()) << endl;
    cout << now.asString(RWZone::utc()) << endl;
    return 0;
}
```

Enumerations

`enum RWZone::DstRule { NoDST, NoAm, WeEu };`
Used by the static member function `dstRule()`, described below, and by constructors for classes derived from `RWZone`.

```
enum RWZone::StdZone {
    NewZealand = -12,CarolineIslands,MarianaIslands,
    Japan,China,Java,
    Kazakh,Pakistan,CaspianSea,
    Ukraine,Nile,Europe,
    Greenwich,Azores,Oscar,
    Greenland,Atlantic,USEastern,
    USCentral,USMountain,USPacific,
    Yukon,Hawaii,Bering
};
```

`StdZone` is provided to name the standard time zones. Its values are intended to be passed to constructors of classes derived from `RWZone`.

Public member functions

`virtual int        timeZoneOffset() const = 0;`
Returns the number of seconds west of UTC for standard time in this zone. The number is negative for zones east of Greenwich, England.

`virtual int        altZoneOffset() const = 0;`
Returns the number of seconds west of UTC for daylight savings time in this zone.

`virtual RWBoolean  daylightObserved() const = 0;`
Returns `TRUE` if daylight savings time is observed for this zone.

`virtual RWBoolean  isDaylight(const struct tm* tspec) const = 0;`
Returns `TRUE` if the time and date represented in the `struct tm` argument is in the range of daylight savings time for this zone. The elements of the `struct tm` argument must all be self-consistent; in particular, the `tm_wday` member must agree with the `tm_year, tm_mon`, and `tm_day` members.

```
virtual void       getBeginDaylight(struct tm*)  const = 0;
virtual void       getEndDaylight  (struct tm*)  const = 0;
```

Return with the `struct tm` argument set to the (local) time that daylight savings time begins, or ends, for the year indicated by the `tm_year` member passed in. If daylight savings time is not observed, the `struct tm` members are all set to a negative value.

Note – that in the southern hemisphere, daylight savings time ends at an earlier date than it begins.

```
virtual RWCString     timeZoneName() const = 0;
virtual RWCString     altZoneName() const = 0;
```

Return the name of (respectively) the standard and daylight savings time zones represented, such as “PST” and “PDT”.

Static public member functions

```
static const RWZone&  local();
```

Returns a reference to an `RWZone` representing local time, with daylight savings time if observed. This is used as the default argument value for `RWDate` and `RWTime` functions that take an `RWZone`.

```
static const RWZone&  standard();
```

Returns a reference to an `RWZone` representing standard local time, with no daylight savings time corrections.

```
static const RWZone&  utc();
```

Returns a reference to an `RWZone` representing UTC (GMT) universal time.

```
static const RWZone*  local(const RWZone*)
static const RWZone*  standard(const RWZone*)
```

These functions allow the values returned by the other functions above to be set. Each returns the previous value. The local and standard time zones are global per-thread. They are inteded to represent the acutal time zone. It is particularly important in a multithreaded application to set the values once before doing any other operation that depends on the time zone.

```
static constRWDaylightRule* dstRule(DstRule = NoAm);
```

Function `dstRule()` is provided for convenience in constructing `RWZoneSimple` instances for time zones in which common daylight savings time rules are obeyed. Currently two such rule systems are provided, `NoAm` for the U.S.A. and Canada, and `WeEu` for most of Western Europe. The result of calling `dstRule()` is normally passed to the `RWZoneSimple` constructor.

RWZoneSimple

RWZoneSimple
```
    |
RWZone
```

Synopsis

```
#include <time.h>
#include <rw/zone.h>

RWZoneSimple myZone(USCentral);
```

Description

`RWZoneSimple` is an implementation of the abstract interface defined by class `RWZone`. It implements a simple daylight savings time rule sufficient to represent all historical U.S. conventions and many European and Asian conventions. It is table-driven and depends on parameters given by class `RWDaylightRule`.

Direct use of `RWDaylightRule` affords the most general interface to `RWZoneSimple`. However, a much simpler programmatic interface is offered, as illustrated by the examples below.

Three instances of `RWZoneSimple` are automatically constructed at program startup, to represent UTC, Standard, and local time. They are available via calls to the static member functions `RWZone::utc()`, `RWZone::standard()`, and `RWZone::local()`, respectively. These are set up according to the time zone facilities provided in the execution environment (defined by the environment variable `TZ`). By default, if DST is observed at all, then the local zone instance will use U.S. (`RWZone::NoAm`) daylight savings time rules.

Other instances of `RWZoneSimple` may be constructed to represent other time zones, and may be installed globally using `RWZone` static member functions `RWZone::local(const RWZone*)` and `RWZone::standard(const RWZone*)`.

Examples

To install US Central time as your global "local" time use:

```
RWZone::local(new RWZoneSimple(RWZone::USCentral));
```

To install Hawaiian time (where daylight savings time is not observed) one would say,

```
RWZone::local(new RWZoneSimple(RWZone::Hawaii,
RWZone::NoDST));
```

Likewise for Japan:

```
RWZone::local(new RWZoneSimple(RWZone::Japan,
RWZone::NoDST));
```

For France:

```
RWZone::local(new RWZoneSimple(RWZone::Europe,
RWZone::WeEu));
```

Here are the rules used internally for the `RWZone::NoAm` and `RWZone::WeEu` values of `RWZone::DstRule:`

```
// last Sun in Apr to last in Oct:
       const RWDaylightRuleusRuleAuld =
       {0,                     0000, 1, { 3, 4, 0, 120 }, { 9, 4,
        0, 120 } };
// first Sun in Apr to last in Oct
       const RWDaylightRuleusRule67 =
       { &usRuleAuld,       1967, 1, { 3, 0, 0, 120 }, { 9, 4, 0,
        120 } };
// first Sun in Jan to last in Oct:
       const RWDaylightRuleusRule74 =
       {&usRule67,          1974, 1, { 0, 0, 0, 120 }, { 9, 4, 0,
        120 } };
// last Sun in Feb to last in Oct
       const RWDaylightRuleusRule75 =
       {&usRule74,          1975, 1, { 1, 4, 0, 120 }, { 9, 4, 0,
        120 } };
// last Sun in Apr to last in Oct
       const RWDaylightRuleusRule76 =
       {&usRule75,          1976, 1, { 3, 4, 0, 120 }, { 9, 4, 0,
        120 } };
// first Sun in Apr to last in Oct
       const RWDaylightRuleusRuleLate =
       {&usRule76,          1987, 1, { 3, 0, 0, 120 }, { 9, 4, 0,
        120 } };
// last Sun in Mar to last in Sep
       const RWDaylightRuleeuRuleLate =
       {0,                  0000, 1, { 2, 4, 0, 120 }, { 8, 4, 0,
        120 } };
```

Given these definitions,

```
RWZone::local(new RWZoneSimple(RWZone::USCentral,
&usRuleLate));
```

is equivalent to the first example given above and repeated here:

```
RWZone::local(new RWZoneSimple(RWZone::USCentral));
```

Daylight savings time systems that cannot be represented with `RWDaylightRule` and `RWZoneSimple` must be modeled by deriving from `RWZone` and implementing its virtual functions.

For example, under Britain's madcap Summer Time rules, alternate timekeeping begins the morning after the third Saturday in April, unless that is Easter (in which case it begins the week before) or unless the Council decides on some other time for that year. In some years Summer Time has been two hours ahead, or has extended through winter without a break. British Summer Time clearly deserves a `RWZone` class all its own.

Constructors

```
RWZoneSimple(RWZone::StdZone zone, RWZone::DstRule =
RWZone::NoAm);
```

Constructs an `RWZoneSimple` instance using internally held `RWDaylightRules`. This is the simplest interface to `RWZoneSimple`. The first argument is the time zone for which an `RWZoneSimple` is to be constructed. The second argument is the daylight savings time rule which is to be followed.

```
RWZoneSimple(const RWDaylightRule* rule,
                         long tzoff,   const RWCString& tzname,
                         long altoff, const RWCString& altname);
```

Constructs an `RWZoneSimple` instance which Daylight Savings Time is computed according to the rule specified. Variables `tzoff` and `tzname` are the offset from UTC (in seconds, positive if west of 0 degrees longitude) and the name of standard time. Arguments `altoff` and `altname` are the offset (typically equal to `tzoff - 3600`) and name when daylight savings time is in effect. If `rule` is zero, daylight savings time is not observed.

```
RWZoneSimple(long tzoff, const RWCString& tzname);
```

Constructs an `RWZoneSimple` instance in which Daylight Savings Time is not observed. Argument `tzoff` is the offset from UTC (in seconds, positive if west of 0 degrees longitude) and `tzname` is the name of the zone.

`RWZoneSimple(RWZone::StdZone zone, const RWDaylightRule* rule);`

Constructs an `RWZoneSimple` instance in which offsets and names are specified by the `StdZone` argument. Daylight Savings Time is computed according to the rule argument, if non-zero; otherwise, DST is not observed.

Part 3— Templates

Templates 23

RWTBitVec<size>

Synopsis

```
#include <rw/tbitvec.h>
RWTBitVec<22>         // A 22 bit long vector
```

`RWTBitVec`<*size*> is a parameterized bit vector of fixed length *size*. Unlike class `RWBitVec`, its length cannot be changed at run time. Its advantage of `RWBitVec` is its smaller size, and one less level of indirection, resulting in a slight speed advantage.

Bits are numbered from 0 through *size*–1, inclusive.

The copy constructor and assignment operator use *copy* semantics.

Example

In this example, a bit vector 24 bits long is exercised:

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

main()
{
  RWTBitVec<24> a, b;     // Allocate two vectors.
  a(2) = TRUE;   // Set bit 2 (the third bit) of a on.
  b(3) = TRUE;   // Set bit 3 (the fourth bit) of b on.
  RWTBitVec<24> c = a ^ b;       // Set c to the XOR of a and b.
}
```

Public constructor

`RWTBitVec<`*size*`>();`
Constructs an instance with all bits set to FALSE.

`RWTBitVec<`*size*`>(RWBoolean val);`
Constructs an instance with all bits set to `val`.

Assignment operators

```
RWTBitVec<size>&           operator=(const
                            RWTBitVec<size>& v);
```
Sets self to a copy of `v`.

`RWTBitVec&operator=(RWBoolean val);`
Sets all bits in self to the value `val`.

```
RWTBitVec&operator&=(const RWTBitVec& v);
RWTBitVec&operator^=(const RWTBitVec& v)
RWTBitVec&operator|=(const RWTBitVec& v)
```
Logical assignments. Sets each bit of self to the logical AND, XOR, or OR, respectively, of self and the corresponding bit in `v`.

`RWBitRefoperator[](size_t i);`
Returns a reference to the `i`'th bit of self. This reference can be used as an lvalue. The index `i` must be between 0 and *size*–1, inclusive. Bounds checking will occur.

`RWBitRefoperator()(size_t i);`
Returns a reference to the `i`'th bit of self. This reference can be used as an lvalue. The index `i` must be between 0 and *size*–1, inclusive. No bounds checking is done.

Logical operators

`RWBoolean                  operator==(RWBoolean b) const;`
Returns `TRUE` if every bit of self is set to the value b. Otherwise, returns `FALSE`.

`RWBooleanoperator!=(RWBoolean b) const;`
Returns `TRUE` if any bit of self is not set to the value `b`. Otherwise, returns `FALSE`.

```
RWBooleanoperator==(const RWTBitVec& v)
 const;
```
Returns `TRUE` if each bit of self is set to the same value as the corresponding bit in `v`. Otherwise, returns `FALSE`.

`RWBooleanoperator!=(const RWTBitVec& v) const;`
Returns `TRUE` if any bit of self is not set to the same value as the corresponding bit in `v`. Otherwise, returns `FALSE`.

`voidclearBit(size_t i);`
Clears (*i.e.*, sets to `FALSE`) the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following two lines are equivalent, although `clearBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
    a(i) = FALSE;
    a.clearBit(i);
```

`const RWByte*data() const;`
Returns a const pointer to the raw data of self. Should be used with care.

`size_tfirstFalse() const;`
Returns the index of the first OFF (False) bit in self. Returns `RW_NPOS` if there is no OFF bit.

`size_tfirstTrue() const;`
Returns the index of the first ON (True) bit in self. Returns `RW_NPOS` if there is no ON bit.

`voidsetBit(size_t i);`
Sets (*i.e.*, sets to `TRUE`) the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following two lines are equivalent, although `setBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
    a(i) = TRUE;
    a.setBit(i);
```

`RWBooleantestBit(size_t i) const;`
Tests the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following are equivalent, although `testBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
    if( a(i) )           doSomething();
    if( a.testBit(i) )   doSomething();
```

Related global functions

```
RWTBitVec operator&(const RWTBitVec& v1, const RWTBitVec&
v2);
RWTBitVec operator^(const RWTBitVec& v1, const RWTBitVec&
v2);
RWTBitVec operator|(const RWTBitVec& v1, const RWTBitVec&
v2);
```

Return the logical AND, XOR, and OR, respectively, of vectors `v1` and `v2`.

RWTIsvDlist<T>

Synopsis

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

RWTIsvDlist    list;
```

Descripton

Class `RWTIsvDlist<T>` is a class that implements intrusive doubly-linked lists.

An intrusive list is one where the member of the list must inherit from a common base class, in this case `RWIsvDlink`. The advantage of such a list is that memory and space requirements are kept to a minimum. The disadvantage is that the inheritance hierarchy is inflexible, making it slightly more difficult to use with an existing class. Class `RWTValDlist<T>` is offered as an alternative, non-intrusive, linked list.

See *Stroustrup* (1991; Section 8.3.1) for more information about intrusive lists.

Caution – 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.

Example

```
#include <rw/tidlist.h>
#include <rw/rstream.h>
#include <string.h>

struct Symbol : public RWIsvDlink {
  char name[10];
  Symbol( const char* cs)
    { strncpy(name, cs, sizeof(name)); name[9] = '\0'; }
};

void printem(Symbol* s, void*) { cout << s->name << endl; }

main(){
  RWTIsvDlist<Symbol> list;
  list.insert( new Symbol("one") );
  list.insert( new Symbol("two") );
  list.prepend( new Symbol("zero") );

  list.apply(printem, 0);
  list.clearAndDestroy();  // Deletes the items inserted into the
list
  return 0;
}
```

Program Output:

```
zero
one
two
```

Public constructors

`RWTIsvDlist();`
Constructs an empty list.

`RWTIsvDlist(T* a);`
Constructs a list with the single item pointed to by `a` in it.

Public Members Functions

`void                    append(T* a);`
Appends the item pointed to by `a` to the end of the list.

`voidapply(void (*applyFun)(T*,`
` void*), void* d);`
Calls the function pointed to by `applyFun` to every item in the collection. This must have the prototype:

```
void  yourFun(T* item, void* d);
```

The item will be passed in as argument item. Client data may be passed through as parameter `d`.

`T*at(size_t i) const;`
Returns the item at index `i`. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

`voidclear();`
Removes all items from the list.

`voidclearAndDestroy();`
Removes *and calls delete* for each item in the list.

Note – This assumes that each item was allocated off the heap.

`RWBooleancontains(RWBoolean`
` (*testFun)(const T*, void*),`
`  void* d) const;`
Returns `TRUE` if the list contains an item for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . The tester function must have the prototype:

```
RWBoolean  yourTester(const T* item, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`RWBooleancontainsReference(const T* a)`
` const;`
Returns `TRUE` if the list contains an item with the address `a`.

`size_tentries() const;`
Returns the number of items currently in the list.

`T*find(RWBoolean (*testFun)(const`
` T*, void*), void* d) const;`
Returns the first item in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE`. If there is no such item, then returns nil. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T* item, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`T*first() const;`
Returns (but does not remove) the first item in the list, or nil if the list is empty.

`T*get();`
Returns *and removes* the first item in the list, or nil if the list is empty.

`size_tindex(RWBoolean`
` (*testFun)(const T*, void*),`
`  void* d) const;`
Returns the index of the first item in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE`. If there is no such item, then returns `RW_NPOS`. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T* item, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`voidinsert(T* a);`
Appends the item pointed to by `a` to the end of the list. This item cannot be inserted into more than one list, nor can it be inserted into the same list more than once.

`voidinsertAt(size_t i, T* a);`
Insert the item pointed to by `a` at the index position `i`. This position must be between zero and the number of items in the list, or an exception of type `TOOL_INDEX` will be thrown. The item cannot be inserted into more than one list, nor can it be inserted into the same list more than once.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`T*last() const;`
Returns (but does not remove) the last item in the list, or nil if the list is empty.

```
size_toccurrencesOf(RWBoolean
 (*testFun)(const T*, void*),
  void* d) const;
```
Traverses the list and returns the number of times for which the user-defined "tester" function pointed to by `testFun` returned `TRUE` . The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T* item, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
size_toccurrencesOfReference(const T*
 a) const;
```
Returns the number of times which the item pointed to by `a` occurs in the list. Because items cannot be inserted into a list more than once, this function can only return zero or one.

```
voidprepend(T* a);
```
Prepends the item pointed to by `a` to the beginning of the list.

```
T*remove(RWBoolean
 (*testFun)(const T*, void*),
  void* d);
```
Removes and returns the first item for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or nil if there is no such item. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T* item, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
T*removeAt(size_t i);
```
Removes and returns the item at index `i.` The index `i` must be between zero and the number of items in the collection less one or an exception of type `TOOL_INDEX` will be thrown.

```
T*removeFirst();
```
Removes and returns the first item in the list, or nil if there are no items in the list.

`T*removeLast();`
Removes and returns the last item in the list, or nil if there are no items in the list.

`T*removeReference(T* a);`
Removes and returns the item with address `a`, or nil if there is no such item.

RWTIsvDlistIterator<T>

Synopsis

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

RWTIsvDlist<T> list;

RWTIsvDlistIterator<T> iterator(list);
```

Description

Iterator for class `RWTIsvDlist<T>`, allowing sequential access to all the elements of a doubly-linked parameterized intrusive list. Elements are accessed in order, in either direction.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTIsvDlistIterator(RWTIsvDlist<T>& c);`
Constructs an iterator to be used with the list `c`.

Public operators

`T*                    operator++();`
Advances the iterator one position, returning a pointer to the new link, or nil if the end of the list has been reached.

`T*operator--();`
Reverses the iterator one position, returning a pointer to the new link, or nil if the beginning of the list has been reached.

`T*operator+=(size_t n);`
Advances the iterator n positions, returning a pointer to the new link, or nil if the end of the list has been reached.

`T*operator-=(size_t n);`
Reverses the iterator n positions, returning a pointer to the new link, or nil if the beginning of the list has been reached.

`T*operator()();`
Advances the iterator one position, returning a pointer to the new link, or nil if the end of the list has been reached.

Public member functions

```
RWTIsvDlist<T>*           container() const;
```

Returns a pointer to the collection over which this iterator is iterating.

```
T*findNext(RWBoolean
 (*testFun)(const T*, void*),
  void*);
```

Advances the iterator to the first link for which the tester function pointed to by `testFun` returns `TRUE` and returns it, or nil if there is no such link.

```
voidinsertAfterPoint(T* a);
```

Inserts the link pointed to by a into the iterator's associated collection in the position immediately after the iterator's current position.

```
T*key() const;
```

Returns the link at the iterator's current position. Returns nil if the iterator is not valid.

```
T*remove();
```

Removes and returns the current link from the iterator's associated collection. Returns nil if unsuccessful. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed link.

```
T*removeNext(RWBoolean
 (*testFun)(const T*, void*),
  void*);
```

Advances the iterator to the first link for which the tester function pointed to by `testFun` returns `TRUE`, removes and returns it. Returns `FALSE` if unsuccessful. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

```
voidreset();
```

Resets the iterator to the state it had immediately after construction.

```
voidreset(RWTIsvDlist<T>& c);
```

Resets the iterator to iterate over the collection `c`.

RWTIsvSlist<T>

Synopsis

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

RWTIsvSlist<T> list;
```

Descripton

Class `RWTIsvSlist<T>` is a class that implements intrusive singly-linked lists.

An intrusive list is one where the member of the list must inherit from a common base class, in this case `RWIsvSlink`. The advantage of such a list is that memory and space requirements are kept to a minimum. The disadvantage is that the inheritance hierarchy is inflexible, making it slightly more difficult to use with an existing class. Class `RWTValSlist<T>` is offered as an alternative, non-intrusive, linked list.

See *Stroustrup* (1991; Section 8.3.1) for more information about intrusive lists.

Caution – 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. For this reason, the copy constructor and assignment operator for `RWIsvSlink` have been made private.

Example

```
#include <rw/tislist.h>
#include <rw/rstream.h>
#include <string.h>

struct Symbol : public RWIsvSlink {
  char name[10];
  Symbol( const char* cs)
    { strncpy(name, cs, sizeof(name)); name[9] = '\0'; }
};

void printem(Symbol* s, void*) { cout << s-name << endl; }

main(){
  RWTIsvSlist<Symbol> list;
  list.insert( new Symbol("one") );
  list.insert( new Symbol("two") );
  list.prepend( new Symbol("zero") );

  list.apply(printem, 0);
  list.clearAndDestroy();  // Deletes the items inserted into the
list
  return 0;
}
```

Program Output:

```
zero
one
two
```

Public constructors

`RWTIsvSlist();`
Constructs an empty list.

`RWTIsvSlist(T* a);`
Constructs a list with the single item pointed to by `a` in it.

Public member functions

`void                    append(T* a);`
Appends the item pointed to by `a` to the end of the list.

```
voidapply(void (*applyFun)(T*,
 void*), void* d);
```

Calls the function pointed to by `applyFun` to every item in the collection. This must have the prototype:

```
    void  yourFun(T* item, void* d);
```

The item will be passed in as argument item. Client data may be passed through as parameter `d`.

```
T*at(size_t i) const;
```

Returns the item at index `i`. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

```
voidclear();
```

Removes all items from the list.

```
voidclearAndDestroy();
```

Removes *and calls delete* for each item in the list.

Note – This assumes that each item was allocated off the heap.

```
RWBooleancontains(RWBoolean
 (*testFun)(const T*, void*),
  void* d) const;
```

Returns `TRUE` if the list contains an item for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . The tester function must have the prototype:

```
    RWBoolean  yourTester(const T* item, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
RWBooleancontainsReference(const T* a)
 const;
```

Returns `TRUE` if the list contains an item with the address `a`.

```
size_tentries() const;
```

Returns the number of items currently in the list.

`T*find(RWBoolean (*testFun)(const`
` T*, void*), void* d) const;`

Returns the first item in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE`. If there is no such item, then returns nil. The tester function must have the prototype:

```
RWBoolean  yourTester(const T* item, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`T*first() const;`

Returns (but does not remove) the first item in the list, or nil if the list is empty.

`T*get();`

Returns *and removes* the first item in the list, or nil if the list is empty.

`size_tindex(RWBoolean`
` (*testFun)(const T*, void*),`
`  void* d) const;`

Returns the index of the first item in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE`. If there is no such item, then returns `RW_NPOS`. The tester function must have the prototype:

```
RWBoolean  yourTester(const T* item, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`voidinsert(T* a);`

Appends the item pointed to by `a` to the end of the list. This item cannot be inserted into more than one list, nor can it be inserted into the same list more than once.

`voidinsertAt(size_t i, T* a);`

Insert the item pointed to by `a` at the index position `i`. This position must be between zero and the number of items in the list, or an exception of type `TOOL_INDEX` will be thrown. The item cannot be inserted into more than one list, nor can it be inserted into the same list more than once.

`RWBooleanisEmpty() const;`

Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`T*last() const;`

Returns (but does not remove) the last item in the list, or nil if the list is empty.

```
size_toccurrencesOf(RWBoolean
 (*testFun)(const T*, void*),
  void* d) const;
```
Traverses the list and returns the number of times for which the user-defined "tester" function pointed to by `testFun` returned `TRUE` . The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T* item, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
size_toccurrencesOfReference(const T*
 a) const;
```
Returns the number of times which the item pointed to by `a` occurs in the list. Because items cannot be inserted into a list more than once, this function can only return zero or one.

```
voidprepend(T* a);
```
Prepends the item pointed to by `a` to the beginning of the list.

```
T*remove(RWBoolean
 (*testFun)(const T*, void*),
  void* d);
```
Removes and returns the first item for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or nil if there is no such item. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T* item, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
T*removeAt(size_t i);
```
Removes and returns the item at index `i`. The index `i` must be between zero and the number of items in the collection less one or an exception of type `TOOL_INDEX` will be thrown.

```
T*removeFirst();
```
Removes and returns the first item in the list, or nil if there are no items in the list.

`T*removeLast();`
Removes and returns the last item in the list, or nil if there are no items in the list. This function is relatively slow because removing the last link in a singly-linked list necessitates access to the next-to-the-last link, requiring the whole list to be searched.

`T*removeReference(T* a);`
Removes and returns the link with address `a`. The link must be in the list. In a singly-linked list this function is not very efficient.

RWTIsvSlistIterator<T>

Synopsis

```
#include <rw/tislist.h>
RWTIsvSlist<T> list;
RWTIsvSlistIterator<T> iterator(list);
```

Description

Iterator for class `RWTIsvSlist<T>`, allowing sequential access to all the elements of a singly-linked parameterized intrusive list. Elements are accessed in order, from first to last.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTIsvSlistIterator(RWTIsvSlist<T>& c);`
Constructs an iterator to be used with the list `c`.

Public operators

`T*                    operator++();`
Advances the iterator one position, returning a pointer to the new link, or nil if the end of the list has been reached.

`T*operator+=(size_t n);`
Advances the iterator `n` positions, returning a pointer to the new link, or nil if the end of the list has been reached.

`T*operator()();`
Advances the iterator one position, returning a pointer to the new link, or nil if the end of the list has been reached.

Public member functions

`RWTIsvSlist<T>*            container() const;`
Returns a pointer to the collection over which this iterator is iterating.

```
T*findNext(RWBoolean
 (*testFun)(const T*, void*),
  void*);
```
Advances the iterator to the first link for which the tester function pointed to by `testFun` returns `TRUE` and returns it, or nil if there is no such link.

`voidinsertAfterPoint(T* a);`
Inserts the link pointed to by `a` into the iterator's associated collection in the position immediately after the iterator's current position.

`T*key() const;`
Returns the link at the iterator's current position. Returns nil if the iterator is not valid.

`T*remove();`
Removes and returns the current link from the iterator's associated collection. Returns nil if unsuccessful. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed link. This function is relatively inefficient for a singly-linked list.

```
T*removeNext(RWBoolean
 (*testFun)(const T*, void*),
  void*);
```
Advances the iterator to the first link for which the tester function pointed to by `testFun` returns `TRUE`, removes and returns it. Returns `FALSE` if unsuccessful. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTIsvSlist<TL>& c);`
Resets the iterator to iterate over the collection `c`.

RWTPtrDlist<T>

Synopsis

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

RWTPtrDlist<T> list;
```

Description

This class maintains a collection of pointers to type `T`, implemented as a doubly linked list. This is a *pointer* based list: pointers to objects are copied in and out of the links that make up the list.

Parameter `T` represents the type of object to be inserted into the list, either a class or built in type. The class `T` must have:

- well-defined equality semantics (`T::operator==(const T&)`).

Example

In this example, a doubly-linked list of pointers to the user type Dog is exercised. Contrast this approach with the example given under `RWTValDlist<T>`.

Code Example 23-1 (1 of 2)

```
#include <rw/tpdlist.h>
#include <rw/rstream.h>
#include <string.h>

class Dog {
  char* name;
public:
  Dog( const char* c) {
    name = new char[strlen(c)+1];
    strcpy(name, c); }

  ~Dog() { delete name; }

  // Define a copy constructor:
  Dog(const Dog& dog) {
    name = new char[strlen(dog.name)+1];
    strcpy(name, dog.name); }

  // Define an assignment operator:
  void operator=(const Dog& dog) {
```

Code Example 23-1 (2 of 2)

```
    if (this!=&dog) {
      delete name;
      name = new char[strlen(dog.name)+1];
      strcpy(name, dog.name);
    }
  }

  // Define an equality test operator:
  int operator==(const Dog& dog) const {
    return strcmp(name, dog.name)==0; }

  friend ostream& operator<<(ostream& str, const Dog& dog){
    str << dog.name;
    return str;}
};

main()
{
  RWTPtrDlist<dog> terriers;
  terriers.insert(new Dog("Cairn Terrier"));
  terriers.insert(new Dog("Irish Terrier"));
  terriers.insert(new Dog("Schnauzer"));

  Dog key1("Schnauzer");
  cout << "The list " <<
    (terriers.contains(&key1) ? "does " : "does not ") <<
    "contain a Schnauzer\n";

  Dog key2("Irish Terrier");
  terriers.insertAt(
      terriers.index(&key2),
      new Dog("Fox Terrier")
    );

  Dog* d;
  while (!terriers.isEmpty()) {
    d = terriers.get();
    cout << *d << endl;
    delete d;
  }

  return 0;
}
```

Program output:

```
The list does contain a Schnauzer
Cairn Terrier
Fox Terrier
Irish Terrier
Schnauzer
```

Public constructors

`RWTPtrDlist<T>();`
Constructs an empty list.

`RWTPtrDlist<T>(const RWTPtrDlist<T>& c);`
Constructs a new doubly-linked list as a shallow copy of `c`. After construction, pointers will be shared between the two collections.

Public operators

```
RWTPtrDlist&                operator=(const RWTPtrDlist<T>&
                             c);
```
Sets self to a shallow copy of `c`. Afterwards, pointers will be shared between the two collections.

```
T*&operator[](size_t i);
T*operator[](size_t i) const;
```
Returns a pointer to the `i`'th value in the list. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

```
void                        append(T* a);
```
Appends the item pointed to by `a` to the end of the list.

```
voidapply(void (*applyFun)(T*,
 void*), void* d);
```
Applies the user-defined function pointed to by `applyFun` to every item in the list. This function must have the prototype:

```
void yourFun(T* a, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`T*&at(size_t i);`
`T*at(size_t i) const;`
Returns a pointer to the `i`'th value in the list. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

`voidclear();`
Removes all items from the collection.

`voidclearAndDestroy();`
Removes all items from the collection *and* calls their destructors.

`RWBooleancontains(T* a) const;`
Returns `TRUE` if the list contains an object that is equal to the object pointed to by `a`, `FALSE` otherwise. Equality is measured by the class-defined equality operator for type `T`.

```
RWBooleancontains(RWBoolean
 (*testFun)(const T*, void*),
 void* d) const;
```
Returns `TRUE` if the list contains an item for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . Returns `FALSE` otherwise. The tester function must have the prototype:

`RWBoolean` *yourTester*`(T*, void* d);`

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`size_tentries() const;`
Returns the number of items that are currently in the collection.

`T*find(const T* a) const;`
Returns a pointer to the first object encountered which is equal to the object pointed to by `a`, or nil if no such object can be found. Equality is measured by the class-defined equality operator for type `T`.

```
T*find(RWBoolean (*testFun)
(const T*, void*), void* d,)
const;
```
Returns a pointer to the first object encountered for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or nil if no such object can be found. The tester function must have the prototype:

```
    RWBoolean   yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`T*first() const;`
Returns a pointer to the first item in the list. The behavior is undefined if the list is empty.

`T*get();`
Returns a pointer to the first item in the list and removes the item. The behavior is undefined if the list is empty.

`size_tindex(const T* a);`
Returns the index of the first object that is equal to the object pointed to by `a`, or `RW_NPOS` if there is no such object. Equality is measured by the class-defined equality operator for type T.

```
size_tindex(RWBoolean (*testFun)
(T*, void*), void* d)
const;
```
Returns the index of the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or `RW_NPOS` if there is no such object. The tester function must have the prototype:

```
    RWBoolean   yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`void insert(T* a);`
Adds the object pointed to by `a` to the end of the list.

`voidinsertAt(size_t i, T* a);`
Adds the object pointed to by `a` at the index position `i`. This position must be between zero and the number of items in the list, or an exception of type `TOOL_INDEX` will be thrown.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`T*last() const;`
Returns a pointer to the last item in the list. The behavior is undefined if the list is empty.

`size_toccurrencesOf(const T* a) const;`
Returns the number of objects in the list that are equal to the object pointed to by `a`. Equality is measured by the class-defined equality operator for type `T`.

```
size_toccurrencesOf(RWBoolean
 (*testFun)(T*, void*), void* d)
  const;
```
Returns the number of objects in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . The tester function must have the prototype:

`RWBoolean` *yourTester*`(T*, void* d);`

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`voidprepend(T* a);`
Adds the item pointed to by `a` to the beginning of the list.

`T*remove(const T* a);`
Removes the first object which is equal to the object pointed to by `a` and returns a pointer to it, or nil if no such object could be found. Equality is measured by the class-defined equality operator for type `T`.

```
T*remove(RWBoolean (*testFun)(T*,
 void*), void* d);
```
Removes the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE` and returns a pointer to it, or nil if there is no such object. The tester function must have the prototype:

`RWBoolean` *yourTester*`(T*, void* d);`

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`size_tremoveAll(const T* a);`
Removes all objects which are equal to the object pointed to by `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator for type `T`.

```
size_tremoveAll(RWBoolean
 (*testFun)(T*, void*), void*
   d);
```

Removes all objects for which the user-defined tester function pointed to by `testFun` returns `TRUE`. Returns the number of objects removed. The tester function must have the prototype:

```
    RWBoolean   yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`T*removeAt(size_t i);`
Removes the object at index `i` and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`T*removeFirst();`
Removes the first item in the list and returns a pointer to it. The behavior is undefined if the list is empty.

`T*removeLast()`
Removes the last item in the list and returns a pointer to it. The behavior is undefined if the list is empty.

RWTPtrDlistIterator<T>

Synopsis

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

RWTPtrDlist<T> list;

RWTPtrDlistIterator<T> iterator(list);
```

Description

Iterator for class `RWTPtrDlist<T>`, allowing sequential access to all the elements of a doubly-linked parameterized list. Elements are accessed in order, in either direction.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTPtrDlistIterator<T>(RWTPtrDlist<T>& c);`
Constructs an iterator to be used with the list `c`.

Public member operators

`RWBoolean                    operator++();`
Advances the iterator to the next item and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator--();`
Retreats the iterator to the previous item and returns `TRUE`. When the beginning of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator+=(size_t n);`
Advances the iterator `n` positions and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator-=(size_t n);`
Retreats the iterator `n` positions and returns `TRUE`. When the beginning of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`T*operator()();`
Advances the iterator to the next item and returns a pointer to it. When the end of the collection is reached, returns nil and the position of the iterator will be undefined.

Public member functions

`RWTPtrDlist<T>              container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`T*findNext(const T* a);`
Advances the iterator to the first element that is equal to the object pointed to by `a` and returns a pointer to it. If no item is found, returns nil and the position of the iterator will be undefined. Equality is measured by the class-defined equality operator for type `T`.

```
T*findNext(RWBoolean
 (*testFun)(T*, void*), void*);
```
Advances the iterator to the first element for which the tester function pointed to by `testFun` returns `TRUE` and returns a pointer to it. If no item is found, returns nil and the position of the iterator will be undefined.

`voidinsertAfterPoint(T* a);`
Inserts the object pointed to by `a` into the iterator's associated collection in the position immediately after the iterator's current position.

`T*key() const;`
Returns a pointer to the object at the iterator's current position. The results are undefined if the iterator is no longer valid.

`T*remove();`
Removes and returns the object at the iterator's current position from the iterator's associated collection. Afterwards, the iterator will be positioned at the element immediately before the removed element. Returns nil if unsuccessful in which case the position of the iterator is undefined.

`T*removeNext(const T* a);`
Advances the iterator to the first element that is equal to the object pointed to by `a`, then removes and returns it. Afterwards, the iterator will be positioned at the element immediately before the removed element. Returns nil if unsuccessful in which case the position of the iterator is undefined. Equality is measured by the class-defined equality operator for type `T`.

```
T*removeNext(RWBoolean
 (*testFun)(T*, void*), void*);
```
Advances the iterator to the first element for which the tester function pointed to by `testFun` returns `TRUE`, then removes and returns it. Afterwards, the iterator will be positioned at the element immediately before the removed element. Returns nil if unsuccessful in which case the position of the iterator is undefined.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTPtrDlist<T>& c);`
Resets the iterator to iterate over the collection `c`.

RWTPtrHashDictionary<K,V>

Synopsis

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

unsigned hashFun(const K&);

RWTPtrHashDictionary<K,V> dictionary(hashFun);
```

Description

`RWTPtrHashDictionary<KV>` is a dictionary of keys of type `K` and values of type `V`, implemented using a hash table. While duplicates of values are allowed, duplicates of keys are not.

It is a *pointer* based collection: pointers to the keys and values are copied in and out of the hash buckets.

Parameters `K` and `V` represent the type of the key and the type of the value, respectively, to be inserted into the table. These can be either classes or built in types. Class `K` must have

- well-defined equality semantics (`K::operator==(const K&)`).

Class `V` can be of any type.

A user-supplied hashing function for type `K` must be supplied to the constructor when creating a new table. If `K` is a Tools.h++ class, then this requirement is usually trivial because all Tools.h++ objects know how to return a hashing value. This function has prototype:

`unsigned` *hFun*`(const K& a);`

and should return a suitable hash value for the object `a`.

To find a value, the key is first hashed to determine in which bucket the key and value can be found. The bucket is then searched for an object that is equal (as determined by the equality operator) to the key.

The initial number of buckets in the table is set by the constructor. There is a default value. If the number of (key / value) pairs in the collection greatly exceeds the number of buckets then efficiency will sag because each bucket must be searched linearly. The number of buckets can be changed by calling member function `resize()`. This is relatively expensive because all of the keys must be rehashed.

If you wish for this to be done automatically, then you can subclass from this class and implement your own special `insert()` and `remove()` functions which perform a `resize()` as necessary.

Example

Code Example 23-2

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

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

main()
{
  RWTPtrHashDictionary<RWCString, RWDate> birthdays(hashString);

  birthdays.insertKeyAndValue
  (
        new RWCString("John"),
        new RWDate(12, "April", 1975)
  );
  birthdays.insertKeyAndValue
  (
        new RWCString("Ivan"),
        new RWDate(2, "Nov", 1980)
  );

  // Alternative syntax:
  birthdays[new RWCString("Susan")] =
    new RWDate(30, "June", 1955);
  birthdays[new RWCString("Gene")] =
    new RWDate(5, "Jan", 1981);

  // Print a birthday:
  RWCString key("John");
  cout << *birthdays[&key] << endl;
  return 0;

}
```

Program output:

```
April 12, 1975
```

Public constructors

```
RWTPtrHashDictionary<KV>(unsigned (*hashKey)(const K&),
                    size_t buckets = RWDEFAULT_CAPACITY);
```

Constructs an empty hash dictionary. The first argument is a pointer to a user-defined hashing function for items of type `K` (the key). The table will initally have `buckets` buckets although this can be changed with member function `resize()`.

```
RWTPtrHashDictionary<KV>(const RWTPtrHashDictionary<KV>& c);
```

Constructs a new hash dictionary as a shallow copy of `c`. After construction, pointers will be shared between the two collections. The new object will use the same hashing function and have the same number of buckets as `c`. Hence, the keys will not be rehashed.

Public operators

```
RWTPtrHashDictionary<KV&
       operator=(const RWTPtrHashDictionary<KV>& c);
```

Sets self to a shallow copy of `c`. Afterwards, pointers will be shared between the two collections. Self will use the same hashing function and have the number of buckets as `c`. Hence, the keys will not be rehashed.

```
V*&operator[](K* key);
```

Look up the key `key` and return a reference to the pointer of its associated value. If the key is not in the dictionary, then it is added to the dictionary. In this case, the pointer to the value will be undefined. Because of this, if there is a possibility that a key will not be in the dictionary, then this operator can only be used as an l-value.

Public member functions

```
void                        applyToKeyAndValue(void
                             (*applyFun)(K*,V*&,void*),
                              void* d);
```

Applies the user-defined function pointed to by `applyFun` to every key-value pair in the dictionary. This function must have prototype:

```
void yourFun(K* key, V*& value, void* d);
```

This function will be called for each key value pair in the dictionary, with a pointer to the key as the first argument and a reference to a pointer to the value as the second argument. The key should not be changed or touched. A new value can be substituted, or the old value can be changed. Client data may be passed through as parameter `d`.

`voidclear();`
Removes all key value pairs from the collection.

`voidclearAndDestroy();`
Removes all key value pairs from the collection *and* calls the destructor for both the keys and the values.

`RWBooleancontains(const K* key) const;`
Returns `TRUE` if the dictionary contains a key which is equal to the key pointed to by `key`. Returns `FALSE` otherwise. Equality is measured by the class-defined equality operator for type `K`.

`size_tentries() const;`
Returns the number of key-value pairs currently in the dictionary.

`K*find(const K* key) const;`
Returns a pointer to the *key* which is equal to the key pointed to by `key`, or nil if no such item could be found. Equality is measured by the class-defined equality operator for type `K`.

`V*findValue(const K* key) const;`
Returns a pointer to the *value* associated with the key pointed to by `key`, or nil if no such item could be found. Equality is measured by the class-defined equality operator for type `K`.

```
K*findKeyAndValue(const K* key,
V*& retVal) const;
```
Returns a pointer to the *key* associated with the key pointed to by `key`, or nil if no such item could be found. If a key is found, the pointer to its associated value is put in `retVal`. Equality is measured by the class-defined equality operator for type `K`.

```
voidinsertKeyAndValue(K* key, V*
 value);
```
If the key pointed to by `key` is in the dictionary, then its associated value is changed to `value`. Otherwise, a new key value pair is inserted into the dictionary.

`RWBooleanisEmpty() const;`
Returns `TRUE` if the dictionary has no items in it, `FALSE` otherwise.

`K*remove(K* key);`
Removes the key and value pair where the key is equal to the key pointed to by `key`. Returns the *key* or nil if no match was found. Equality is measured by the class-defined equality operator for type `K`.

`voidresize(size_t N);`
Changes the number of buckets to `N`. This will result in all of the keys being rehashed.

RWTPtrHashDictionaryIterator<K,V>

Synopsis

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

unsigned hashFun(const K&);

RWTPtrHashDictionary<KV> dictionary(hashFun);

RWTPtrHashDictionaryIterator<KV> iterator(dictionary);
```

Description

Iterator for class `RWTPtrHashDictionary<KV>`, allowing sequential access to all keys and values of a parameterized hash dictionary. Elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTPtrHashDictionaryIterator(RWTPtrHashDictionary& c);`
Constructs an iterator to be used with the dictionary `c`.

Public operators

`RWBoolean                    operator++();`
Advances the iterator to the next key-value pair and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`K*operator()();`
Advances the iterator to the next key-value pair and returns a pointer to the key. When the end of the collection is reached, returns nil and the position of the iterator will be undefined. Use member function `value()` to recover the dictionary value.

Public member functions

`RWTPtrHashDictionary*      container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`K*key() const;`
Returns a pointer to the key at the iterator's current position. The results are undefined if the iterator is no longer valid.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTPtrHashDictionary& c);`
Resets the iterator to iterate over the collection `c`.

`V*value() const;`
Returns a pointer to the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

RWTPtrHashSet<T>

RWTPtrHashSet<T>
|
RWTPtrHashTable<T>

Synopsis

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

unsigned hashFun(const T&);

RWTPtrHashSet(hashFun) set;
```

Description

`RWTPtrHashSet<T>` is a derived class of `RWTPtrHashTable<T>` where the `insert()` function has been overridden to accept only one item of a given value. Hence, each item in the collection will have a unique value.

As with class `RWTPtrHashTable<T>`, you must supply a hashing function to the constructor.

The class `T` must have:

- well-defined equality semantics (`T::operator==(const T&)`).

Example

This examples exercises a set of `RWCStrings`.

Code Example 23-3

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

unsigned hashIt(const RWCString& str){ return str.hash(); }

main()
{
  RWTPtrHashSet<RWCString> set(hashIt);

  set.insert(new RWCString("one"));
  set.insert(new RWCString("two"));
  set.insert(new RWCString("three"));
  set.insert(new RWCString("one"));
```

Code Example 23-3 (Continued)

```
  cout << set.entries() << endl;// Prints "3"
  return 0;
}
```

Program output:

```
3
```

Public constructor

`RWTPtrHashSet<T>(unsigned (*hashFun)(const T&),`
`                      size_t buckets = RWDEFAULT_CAPACITY);`

Constructs an empty hashing set. The first argument is a pointer to a user-defined hashing function for items of type `T`. The table will initally have buckets buckets although this can be changed with member function `resize()`.

Public member functions

`void                          apply(void (*applyFun)(T*,`
`                               void*), void* d);`

Inherited from class `RWTPtrHashTable<T>`.

`voidclear();`

Inherited from class `RWTPtrHashTable<T>`.

`voidclearAndDestroy();`

Inherited from class `RWTPtrHashTable<T>`.

`RWBooleancontains(const T* a) const;`

Inherited from class `RWTPtrHashTable<T>`.

`size_tentries() const;`

Inherited from class `RWTPtrHashTable<T>`.

`T*find(const T* a) const;`

Inherited from class `RWTPtrHashTable<T>`.

`virtual voidinsert(T* a);`

Redefined from class `RWTPtrHashTable<T>` to allow an object of a given value to be inserted only once.

`RWBooleanisEmpty() const;`
Inherited from class `RWTPtrHashTable<T>`.

`size_toccurrencesOf(const T* a) const;`
Inherited from class `RWTPtrHashTable<T>`.

`T*remove(const T* a);`
Inherited from class `RWTPtrHashTable<T>`.

`size_tremoveAll(const T* a);`
Inherited from class `RWTPtrHashTable<T>`.

`voidresize(size_t N);`
Inherited from class `RWTPtrHashTable<T>`.

RWTPtrHashTable<T>

Synopsis

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

unsigned hashFun(const T&);

RWTPtrHashTable<T> table(hashFun);
```

Description

This class implements a parameterized hash table of types `T`. It uses chaining to resolve hash collisions. Duplicates are allowed.

It is a *pointer* based collection: pointers to objects are copied in and out of the hash buckets.

Parameter `T` represents the type of object to be inserted into the table, either a class or built in type. The class `T` must have:

- well-defined equality semantics (`T::operator==(const T&)`).

A user-supplied hashing function for type `T` must be supplied to the constructor when creating a new table. If `T` is a Tools.h++ class, then this requirement is usually trivial because all Tools.h++ objects know how to return a hashing value. This function has prototype:

`unsigned` *hFun*`(const T& a);`

and should return a suitable hash value for the object a.

To find an object, it is first hashed to determine in which bucket it occurs. The bucket is then searched for an object that is equal (as determined by the equality operator) to the candidate.

The initial number of buckets in the table is set by the constructor. There is a default value. If the number of items in the collection greatly exceeds the number of buckets then efficiency will sag because each bucket must be searched linearly. The number of buckets can be changed by calling member function `resize()`. This is relatively expensive because all of the keys must be rehashed.

If you wish for this to be done automatically, then you can subclass from this class and implement your own special `insert()` and `remove()` functions which perform a `resize()` as necessary.

Example

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

unsigned hashIt(const RWCString& str) {return str.hash();}

main()
{
  RWTPtrHashTable<RWCString> table(hashIt);

  table.insert(new RWCString("Alabama"));
  table.insert(new RWCString("Pennsylvania"));
  table.insert(new RWCString("Oregon"));
  table.insert(new RWCString("Montana"));

  RWCString key("Oregon");
  cout << "The table " <<
    (table.contains(&key) ? "does " : "does not ") <<
    "contain Oregon\n";

  table.removeAll(&key);

  cout << "The table " <<
    (table.contains(&key) ? "does " : "does not ") <<    "contain
Oregon";
  return 0;
}
```

Program output:

```
The table does contain Oregon
The table does not contain Oregon
```

Public constructors

```
RWTPtrHashTable<T>(unsigned (*hashFun)(const T&),
                      size_t buckets = RWDEFAULT_CAPACITY);
```

Constructs an empty hash table. The first argument is a pointer to a user-defined hashing function for items of type `T`. The table will initally have `buckets` buckets although this can be changed with member function `resize()`.

`RWTPtrHashTable<T>(const RWTPtrHashTable<T>& c);`
Constructs a new hash table as a shallow copy of `c`. After construction, pointers will be shared between the two collections. The new object will have the same number of buckets as `c`. Hence, the keys will not be rehashed.

Public operators

`RWTPtrHashTable&<T>operator=(const RWTPtrHashTable<T>& c);`
Sets self to a shallow copy of `c`. Afterwards, pointers will be shared between the two collections and self will have the same number of buckets as `c`. Hence, the keys will not be rehashed.

Public member functions

```
void                          apply(void (*applyFun)(T*,
                               void*), void* d);
```
Applies the user-defined function pointed to by `applyFun` to every item in the table. This function must have prototype:

```
    void yourFun(T* a, void* d);
```

Client data may be passed through as parameter `d`. The items should not be changed in any way that could change their hash value.

`voidclear();`
Removes all items from the collection.

`voidclearAndDestroy();`
Removes all items from the collection *and* calls their destructors.

`RWBooleancontains(contains T* p) const;`
Returns `TRUE` if the collection contains an item which is equal to the item pointed to by `p`. Returns `FALSE` otherwise. Equality is measured by the class-defined equality operator for type `T`.

`size_tentries() const;`
Returns the number of items currently in the collection.

`T*find(const T* a) const;`
Returns a pointer to the object which is equal to the object pointed to by `a`, or nil if no such object can be found. Equality is measured by the class-defined equality operator for type `T`.

`voidinsert(T* a);`
Adds the object pointed to by `a` to the collection.

`RWBooleanisEmpty() const;`
Returns `TRUE` if the collection has no items in it, `FALSE` otherwise.

`size_toccurrencesOf(const T* a) const;`
Returns the number of objects in the collection which are equal to the object pointed to by `a`. Equality is measured by the class-defined equality operator for type `T`.

`T*remove(const T* a);`
Removes the object which is equal to the object pointed to by `a` and returns a pointer to it, or nil if no such object could be found. Equality is measured by the class-defined equality operator for type `T`.

`size_tremoveAll(const T* a);`
Removes all objects which are equal to the object pointed to by `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator for type `T`.

`voidresize(size_t N);`
Changes the number of buckets to `N`. This will result in all of the objects in the collection being rehashed.

RWTPtrHashTableIterator<T>

Synopsis

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

RWTPtrHashTable<T> table;

RWTPtrHashTableIterator<T> iterator(table);
```

Description

Iterator for class `RWTPtrHashTable<T>`, allowing sequential access to all the elements of a hash table. Elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTPtrHashTableIterator(RWTPtrHashTable<T>& c);`
Constructs an iterator to be used with the table `c`.

Public operators

`RWBoolean                    operator++();`
Advances the iterator to the next item and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`T*operator()();`
Advances the iterator to the next item and returns a pointer to it. When the end of the collection is reached, returns nil and the position of the iterator will be undefined.

Public member functions

`RWTPtrHashTable<T>*          container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`T*key() const`
Returns a pointer to the item at the iterator's current position. The results are undefined if the iterator is no longer valid.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTPtrHashTable<T>& c);`
Resets the iterator to iterate over the collection `c`.

RWTPtrOrderedVector<T>

Synopsis

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

RWTPtrOrderedVector<T> ordvec;
```

Description

`RWTPtrOrderedVector<T>` is a pointer-based *ordered* collection. That is, the items in the collection have a meaningful ordered relationship with respect to one another and can be accessed by an index number. The order is set by the order of insertion. Duplicates are allowed. The class is implemented as a vector, allowing efficient insertion and retrieval from the end of the collection, but somewhat slower from the beginning of the collection.

The class `T` must have:

- well-defined equality semantics (`T::operator==(const T&)`).

Example

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

main() {

  RWTPtrOrderedVector<double> vec;

  vec.insert(new double(22.0));
  vec.insert(new double(5.3));
  vec.insert(new double(-102.5));
  vec.insert(new double(15.0));
  vec.insert(new double(5.3));

  cout << vec.entries() << " entries\n" << endl;// Prints "5"
  for (int i=0; i<vec.length(); i++)
    cout << *vec[i] << endl;

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

Program output:

```
5 entries
22
5.3
-102.5
15
5.3
```

Public constructors

```
RWTPtrOrderedVector<T>(size_t capac=RWDEFAULT_CAPACITY);
```

Creates an empty ordered vector with capacity `capac`. Should the number of items exceed this value, the vector will be resized automatically.

```
RWTPtrOrderedVector<T>(const RWTPtrOrderedVector<T>& c);
```

Constructs a new ordered vector as a shallow copy of `c`. After construction, pointers will be shared between the two collections.

Public operators

```
RWTPtrOrderedVector<T>&    operator=(const
                             RWTPtrOrderedVector& c);
```

Sets self to a shallow copy of `c`. Afterwards, pointers will be shared between the two collections.

```
T*&operator()(size_t i);
T*operator()(size_t i) const;
```

Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one. No bounds checking is performed.

```
T*&operator[](size_t i);
T*operator[](size_t i) const;
```

Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

```
void                       append(T* a);
```

Appends the item pointed to by `a` to the end of the vector. The collection will automatically be resized if this causes the number of items in the collection to exceed the capacity.

`T*&at(size_t i);`
`T*at(size_t i) const;`
Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

`voidclear();`
Removes all items from the collection.

`voidclearAndDestroy();`
Removes all items from the collection *and* calls their destructors.

`RWBooleancontains(const T* a) const;`
Returns `TRUE` if the collection contains an item that is equal to the object pointed to by `a`, `FALSE` otherwise. A linear search is done. Equality is measured by the class-defined equality operator for type T.

`T* const *data() const;`
Returns a pointer to the raw data of the vector. The contents should not be changed. Should be used with care.

`size_tentries() const;`
Returns the number of items currently in the collection.

`T*find(const T* a) const;`
Returns a pointer to the first object encountered which is equal to the object pointed to by `a`, or nil if no such object can be found. Equality is measured by the class-defined equality operator for type `T`.

`T*first() const;`
Returns a pointer to the first item in the vector. An exception of type `TOOL_INDEX` will occur if the vector is empty.

`size_tindex(const T* a) const;`
Performs a linear search, returning the index of the first object that is equal to the object pointed to by `a`, or `RW_NPOS` if there is no such object. Equality is measured by the class-defined equality operator for type `T`.

`voidinsert(T* a);`
Adds the object pointed to by `a` to the end of the vector. The collection will be resized automatically if this causes the number of items to exceed the capacity.

`voidinsertAt(size_t i, T* a);`
Adds the object pointed to by a at the index position `i`. The item previously at position `i` is moved to `i+1`, *etc*. The collection will be resized automatically if this causes the number of items to exceed the capacity. The index `i` must be between 0 and the number of items in the vector or an exception of type `TOOL_INDEX` will occur.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

`T*last() const;`
Returns a pointer to the last item in the collection. If there are no items in the collection then an exception of type `TOOL_INDEX` will occur.

`size_tlength() const;`
Returns the number of items currently in the collection.

`size_toccurrencesOf(const T* a) const;`
Performs a linear search, returning the number of objects in the collection that are equal to the object pointed to by `a`. Equality is measured by the class-defined equality operator for type `T`.

`voidprepend(T* a);`
Adds the item pointed to by `a` to the beginning of the collection. The collection will be resized automatically if this causes the number of items to exceed the capacity.

`T*remove(const T* a);`
Performs a linear search, removing the first object which is equal to the object pointed to by `a` and returns a pointer to it, or nil if no such object could be found. Equality is measured by the class-defined equality operator for type `T`.

`size_tremoveAll(const T* a);`
Performs a linear search, removing all objects which are equal to the object pointed to by `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator for type `T`.

`T*removeAt(size_t i);`
Removes the object at index `i` and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`T*removeFirst();`
Removes the first item in the collection and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`T*removeLast();`
Removes the last item in the collection and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`voidresize(size_t N);`
Changes the capacity of the collection to `N`.

Note – The number of objects in the collection does not change, just the capacity.

RWTPtrSlist<T>

Synopsis

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

RWTPtrSlist<T> list;
```

Description

This class maintains a collection of pointers to type `T`, implemented as a singly-linked list. This is a *pointer* based list: pointers to objects are copied in and out of the links that make up the list.

Parameter `T` represents the type of object to be inserted into the list, either a class or built in type. The class `T` must have:

- well-defined equality semantics (`T::operator==(const T&)`).

Example

In this example, a singly-linked list of `RWDates` is exercised.

Code Example 23-4

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


main()
{
  RWTPtrSlist<RWDate> dates;
  dates.insert(new RWDate(2, "June", 52));    // 6/2/52
  dates.insert(new RWDate(30, "March", 46));  // 3/30/46
  dates.insert(new RWDate(1, "April", 90));   // 4/1/90

  // Now look for one of the dates:
  RWDate key(2, "June", 52);
  RWDate* d = dates.find(&key);
  if (d){
    cout << "Found date " << *d << endl;
  }

  // Remove in reverse order:
  while (!dates.isEmpty()){
    d = dates.removeLast();
    cout << *d << endl;
```

Code Example 23-4 (Continued)

```
    delete d;
  }

  return 0;
}
```

Program output:

```
Found date June 2, 1952
April 1, 1990
March 30, 1946
June 2, 1952
```

Public constructors

`RWTPtrSlist<T>();`
Construct an empty list.

`RWTPtrSlist<T>(const RWTPtrSlist<T>& c);`
Constructs a new singly-linked list as a shallow copy of `c`. After construction, pointers will be shared between the two collections.

Public operators

`RWTPtrSlist&                operator=(const RWTPtrSlist& c);`
Sets self to a shallow copy of `c`. Afterwards, pointers will be shared between the two collections.

`T*&operator[](size_t i);`
`T*operator[](size_t i) const;`
Returns a pointer to the `i`'th value in the list. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

`void                        append(T* a);`
Appends the item pointed to by a to the end of the list.

```
voidapply(void (*applyFun)(T*,
 void*), void* d);
```
Applies the user-defined function pointed to by `applyFun` to every item in the list. This function must have the prototype:

```
    void yourFun(T* a, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter d.

```
T*&at(size_t i);
T*at(size_t i) const;
```

Returns a pointer to the i'th value in the list. The first variant can be used as an l-value, the second cannot. The index i must be between zero and the number of items in the collection less one, or an exception of type TOOL_INDEX will be thrown.

```
voidclear();
```

Removes all items from the collection.

```
voidclearAndDestroy();
```

Removes all items from the collection *and* calls their destructors.

```
RWBooleancontains(T* a) const;
```

Returns TRUE if the list contains an object that is equal to the object pointed to by a, FALSE otherwise. Equality is measured by the class-defined equality operator for type T.

```
RWBooleancontains(RWBoolean
 (*testFun)(T*, void*),void* d)
  const;
```

Returns TRUE if the list contains an item for which the user-defined "tester" function pointed to by testFun returns TRUE . Returns FALSE otherwise. The tester function must have the prototype:

```
    RWBoolean   yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter d.

```
size_tentries() const;
```

Returns the number of items that are currently in the collection.

```
T*find(T* a) const;
```

Returns a pointer to the first object encountered which is equal to the object pointed to by a, or nil if no such object can be found. Equality is measured by the class-defined equality operator for type T.

`T*find(RWBoolean (*testFun)(T*,`
`void*), void* d,) const;`
Returns a pointer to the first object encountered for which the user-defined tester function pointed to by testFun returns `TRUE`, or nil if no such object can be found. The tester function must have the prototype:

    RWBoolean   *yourTester*(T*, void* d);

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`T*first() const;`
Returns a pointer to the first item in the list. The behavior is undefined if the list is empty.

`T*get();`
Returns a pointer to the first item in the list and removes the item. The behavior is undefined if the list is empty.

`size_tindex(T* a);`
Returns the index of the first object that is equal to the object pointed to by `a`, or `RW_NPOS` if there is no such object. Equality is measured by the class-defined equality operator for type `T`.

`size_tindex(RWBoolean (*testFun)(T*,`
` void*), void* d) const;`
Returns the index of the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or `RW_NPOS` if there is no such object. The tester function must have the prototype:

    RWBoolean   *yourTester*(T*, void* d);

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`void insert(T* a);`
Adds the object pointed to by `a` to the end of the list.

`voidinsertAt(size_t i, T* a);`
Adds the object pointed to by `a` at the index position `i`. This position must be between zero and the number of items in the list, or an exception of type `TOOL_INDEX` will be thrown.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`T*last() const;`

Returns a pointer to the last item in the list. The behavior is undefined if the list is empty.

`size_toccurrencesOf(T* a) const;`

Returns the number of objects in the list that are equal to the object pointed to by `a`. Equality is measured by the class-defined equality operator for type `T`.

```
size_toccurrencesOf(RWBoolean
 (*testFun)(T*, void*), void* d)
  const;
```

Returns the number of objects in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . The tester function must have the prototype:

```
    RWBoolean   yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`voidprepend(T* a);`

Adds the item pointed to by `a` to the beginning of the list.

`T*remove(T* a);`

Removes the first object which is equal to the object pointed to by `a` and returns a pointer to it, or nil if no such object could be found. Equality is measured by the class-defined equality operator for type `T`.

```
T*remove(RWBoolean (*testFun)(T*,
 void*), void* d);
```

Removes the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE` and returns a pointer to it, or nil if there is no such object. The tester function must have the prototype:

```
    RWBoolean   yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`size_tremoveAll(T* a);`

Removes all objects which are equal to the object pointed to by `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator for type `T`.

`size_tremoveAll(RWBoolean`
`(*testFun)(T*, void*), void* d);`
Removes all objects for which the user-defined tester function pointed to by `testFun` returns `TRUE`. Returns the number of objects removed. The tester function must have the prototype:

```
    RWBoolean  yourTester(T*, void* d);
```

This function will be called for each item in the list, with a pointer to the item as the first argument. Client data may be passed through as parameter `d`.

`T*removeAt(size_t i);`
Removes the object at index `i` and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`T*removeFirst();`
Removes the first item in the list and returns a pointer to it. The behavior is undefined if the list is empty.

`T*removeLast()`
Removes the last item in the list and returns a pointer to it. The behavior is undefined if the list is empty. This function is relatively slow because removing the last link in a singly-linked list necessitates access to the next-to-the-last link, requiring that the whole list be searched.

RWTPtrSlistIterator<T>

Synopsis

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

RWTPtrSlist<T> list;

RWTPtrSlistIterator<T> iterator(list);
```

Description

Iterator for class `RWTPtrSlist<T>`, allowing sequential access to all the elements of a singly-linked parameterized list. Elements are accessed in order, from first to last.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTPtrSlistIterator<T>(RWTPtrSlist<T>& c);`
Constructs an iterator to be used with the list `c`.

Public member operators

`RWBoolean                   operator++();`
Advances the iterator to the next item and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator+=(size_t n);`
Advances the iterator `n` positions and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`T*operator()();`
Advances the iterator to the next item and returns a pointer to it. When the end of the collection is reached, returns nil and the position of the iterator will be undefined.

Public member functions

`RWTPtrSlist<T>*             container() const;`

Returns a pointer to the collection over which this iterator is iterating.

`T*findNext(const T* a);`

Advances the iterator to the first element that is equal to the object pointed to by `a` and returns a pointer to it. If no item is found, returns nil and the position of the iterator will be undefined. Equality is measured by the class-defined equality operator for type `T`.

```
T*findNext(RWBoolean
 (*testFun)(T*, void*), void*);
```

Advances the iterator to the first element for which the tester function pointed to by `testFun` returns `TRUE` and returns a pointer to it. If no item is found, returns nil and the position of the iterator will be undefined.

`voidinsertAfterPoint(T* a);`

Inserts the object pointed to by `a` into the iterator's associated collection in the position immediately after the iterator's current position.

`T*key() const;`

Returns a pointer to the object at the iterator's current position. The results are undefined if the iterator is no longer valid.

`T*remove();`

Removes and returns the object at the iterator's current position from the iterator's associated collection. Afterwards, the iterator will be positioned at the element immediately before the removed element. Returns nil if unsuccessful in which case the position of the iterator is undefined. This function is relatively inefficient for a singly-linked list.

`T*removeNext(const T* a);`

Advances the iterator to the first element that is equal to the object pointed to by `a`, then removes and returns it. Afterwards, the iterator will be positioned at the element immediately before the removed element. Returns nil if unsuccessful in which case the position of the iterator is undefined. Equality is measured by the class-defined equality operator for type `T`.

`T*removeNext(RWBoolean`
`(*testFun)(T*, void*), void*);`
Advances the iterator to the first element for which the tester function pointed to by `testFun` returns `TRUE`, then removes and returns it. Afterwards, the iterator will be positioned at the element immediately before the removed element. Returns nil if unsuccessful in which case the position of the iterator is undefined.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTPtrSlist<T>& c);`
Resets the iterator to iterate over the collection `c`.

RWTPtrSortedVector<T>

Synopsis

```
#include <rw/tpsrtvec.h>
RWTPtrSortedVector<T> sortvec;
```

Description

`RWTPtrSortedVector<T>` is a pointer-based *ordered* collection. That is, the items in the collection have a meaningful ordered relationship with respect to each other and can be accessed by an index number. In the case of `RWTPtrSortedVector<T>`, objects are inserted such that objects "less than" themselves are before the object, objects "greater than" themselves after the object. An insertion sort is used. Duplicates are allowed.

Stores a *pointer* to the inserted item into the collection according to an ordering determined by the less-than (<) operator.

The class `T` must have:

- well-defined equality semantics (`T::operator==(const T&)`);
- well-defined less-than semantics (`T::operator<(const T&)`);

Example

This example inserts a set of dates into a sorted vector in no particular order, then prints them out in order.

Code Example 23-5

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

main()
{
  RWTPtrSortedVector<RWDate> vec;
  vec.insert(new RWDate(10, "Aug", 1991));
  vec.insert(new RWDate(9, "Aug", 1991));
  vec.insert(new RWDate(1, "Sept", 1991));
  vec.insert(new RWDate(14, "May", 1990));
  vec.insert(new RWDate(1, "Sept", 1991));  // Add a duplicate
  vec.insert(new RWDate(2, "June", 1991));

  for (int i=0; i<vec.length(); i++)
```

Code Example 23-5 (Continued)

```
    cout << *vec[i] << endl;

  vec.clearAndDestroy();

  return 0;
}
```

Program output:

```
May 14, 1990
June 2, 1991
August 9, 1991
August 10, 1991
September 1, 1991
September 1, 1991
```

Public constructor

`RWTPtrSortedVector(size_t capac = RWDEFAULT_CAPACITY);`
Create an empty sorted vector with an initial capacity equal to `capac`. The vector will be automatically resized should the number of items exceed this amount.

`RWTPtrSortedVector<T>(const RWTPtrSortedVector<T>& c);`
Constructs a new ordered vector as a shallow copy of `c`. After construction, pointers will be shared between the two collections.

Public operators

```
RWTPtrSortedVector<T>&     operator=(const
                           RWTPtrSortedVector& c);
```
Sets self to a shallow copy of `c`. Afterwards, pointers will be shared between the two collections.

```
T*&operator()(size_t i);
T*operator()(size_t i) const;
```
Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one. No bounds checking is performed.

`T*&operator[](size_t i);`
`T*operator[](size_t i) const;`
Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

```
T*&                        at(size_t i);
T*                         at(size_t i) const;
```
Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

`voidclear();`
Removes all items from the collection.

`voidclearAndDestroy();`
Removes all items from the collection *and* calls their destructors.

`RWBooleancontains(const T* a) const;`
Returns `TRUE` if the collection contains an item that is equal to the object pointed to by `a`, `FALSE` otherwise. A binary search is done. Equality is measured by the class-defined equality operator for type `T`.

`T* const *data() const;`
Returns a pointer to the raw data of the vector. The contents should not be changed. Should be used with care.

`size_tentries() const;`
Returns the number of items currently in the collection.

`T*find(const T* a) const;`
Returns a pointer to the first object encountered which is equal to the object pointed to by `a`, or nil if no such object can be found. A binary search is used. Equality is measured by the class-defined equality operator for type `T`.

`T*first() const;`
Returns a pointer to the first item in the vector. An exception of type `TOOL_INDEX` will occur if the vector is empty.

`size_tindex(const T* a) const;`
Performs a binary search, returning the index of the first object that is equal to the object pointed to by `a`, or `RW_NPOS` if there is no such object. Equality is measured by the class-defined equality operator for type `T`.

`voidinsert(T* a);`
Performs a binary search, inserting the object pointed to by `a` after all items that compare less than or equal to it, but before all items that do not. "Less Than" is measured by the class-defined '<' operator for type `T`. The collection will be resized automatically if this causes the number of items to exceed the capacity.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

`T*last() const;`
Returns a pointer to the last item in the collection. If there are no items in the collection then an exception of type `TOOL_INDEX` will occur.

`size_tlength() const;`
Returns the number of items currently in the collection.

`size_toccurrencesOf(const T* a) const;`
Performs a binary search, returning the number of items that are equal to the object pointed to by `a`. Equality is measured by the class-defined equality operator for type `T`.

`T*remove(const T* a);`
Performs a binary search, removing the first object which is equal to the object pointed to by `a` and returns a pointer to it, or nil if no such object could be found. Equality is measured by the class-defined equality operator for type `T`.

`size_tremoveAll(const T* a);`
Performs a binary search, removing all objects which are equal to the object pointed to by `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator for type `T`.

`T*removeAt(size_t i);`
Removes the object at index `i` and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`T*removeFirst();`
Removes the first item in the collection and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`T*removeLast();`
Removes the last item in the collection and returns a pointer to it. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`voidresize(size_t N);`
Changes the capacity of the collection to `N`.

Note – The number of objects in the collection does not change, just the capacity.

RWTPtrVector<T>

Synopsis

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

RWTPtrVector<T> vec;
```

Descripton

Class `RWTPtrVector<T>` is a simple parameterized vector of pointers to objects of type `T`. It is most useful when you know precisely how many objects have to be held in the collection. If the intention is to "insert" an unknown number of objects into a collection, then class `RWTPtrOrderedVector<T>` may be a better choice.

The class `T` can be of any type.

Example

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

main() {

  RWTPtrVector<RWDate> week(7);

  RWDate begin;   // Today's date

  for (int i=0; i<7; i++)
    week[i] = new RWDate(begin++);

  for (i=0; i<7; i++)
    cout << *week[i] << endl;

  return 0;
}
```

Program output:

```
March 16, 1992
March 17, 1992
March 18, 1992
March 19, 1992
March 20, 1992
March 21, 1992
March 22, 1992
```

Public constructors

`RWTPtrVector<T>();`
Constructs an empty vector of length zero.

`RWTPtrVector<T>(size_t n);`
Constructs a vector of length `n`. The initial values of the elements is undefined. Hence, they can (and probably will) be garbage.

`RWTPtrVector<T>(size_t n, T* ival);`
Constructs a vector of length `n`, with each element initialized to the value `ival`.

`RWTPtrVector<T>(const RWTPtrVector& v);`
Constructs self as a shallow copy of `v`. After construction, pointers will be shared between the two vectors.

Public operators

```
RWTPtrVector<T>&           operator=(const
                            RWTPtrVector<T>& v);
```
Sets self to a shallow copy of `v`. Afterwards, the two vectors will have the same length and pointers will be shared between them.

`RWTPtrVector<T>&operator=(T* p);`
Sets all elements in self to the value `p`.

```
T*&operator()(size_t i);
T*operator()(size_t i) const;
```
Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the length of the vector, less one. No bounds checking is performed.

`T*&operator[](size_t i);`
`T*operator[](size_t i) const;`
Returns a pointer to the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index i must be between zero and the length of the vector, less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

`T* const *                    data() const;`
Returns a pointer to the raw data of the vector. Should be used with care.

`size_tlength() const;`
Returns the length of the vector.

`voidreshape(size_t N);`
Changes the length of the vector to `N`. If this results in the vector being lengthened, then the initial value of the additional elements is undefined.

`voidresize(size_t N);`
Changes the length of the vector to `N`. If this results in the vector being lengthened, then the initial value of the additional elements is set to zero.

RWTQueue<T,C>

Synopsis

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

RWTQueue<T, C> queue;
```

Description

This class represents a parameterized queue. Not only can the type of object inserted into the queue be parameterized, but also the implementation.

Parameter `T` represents the type of object in the queue, either a class or built in type. The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- any other semantics required by class `C`.

Parameter `C` represents the class used for implementation. Useful choices are `RWTValSlist<T>` or `RWTValDlist<T>`. Vectors, such as `RWTValOrderedVector<T>`, can also be used, but tend to be less efficient at removing an object from the front of the list.

Example

In this example a queue of `RWCStrings`, implemented as a singly-linked list, is exercised.

```
#include <rw/tqueue.h>
#include <rw/cstring.h>
#include <rw/tvslist.h>
#include <rw/rstream.h>

main() {
  RWTQueue<RWCString, RWTValSlist<RWCString> > queue;

  queue.insert("one");   // Type conversion occurs
  queue.insert("two");
  queue.insert("three");

  while (!queue.isEmpty())
    cout << queue.get() << endl;

  return 0;
}
```

Program output:

```
one
two
three
```

Public constructor

`RWTQueue<T>,C>();`
Construct an empty queue of objects of type `T`, implemented using class `C`.

Public member functions

`void                          clear();`
Removes all items from the queue.

`size_tentries() const;`
Returns the number of items in the queue.

`Tfirst() const;`
Returns, but does not remove, the first item in the queue (the item least recently inserted into the queue).

`Tget();`
Returns and removes the first item in the queue (the item least recently inserted into the queue).

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the queue, otherwise `FALSE`.

`voidinsert(const T& a);`
Inserts the item `a` at the end of the queue.

`Tlast() const;`
Returns, but does not remove, the last item in the queue (the item most recently inserted into the queue).

RWTStack<T,C>

Synopsis

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

RWTStack<T, C> stack;
```

Description

This class maintains a stack of values. Not only can the type of object inserted onto the stack be parameterized, but also the implementation of the stack.

Parameter `T` represents the type of object in the stack, either a class or built in type. The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- any other semantics required by class `C`.

Parameter `C` represents the class used for implementation. Useful choices are `RWTValOrderedVector<T>` or `RWTValDlist<T>`. Class `RWTValSlist<T>` can also be used, but note that singly-linked lists are less efficient at removing the last item of a list (`function pop()`), because of the necessity of searching the list for the next-to-the-last item.

Example

In this example a stack of ints, implemented as an ordered vector, is exercised.

```
#include <rw/tstack.h>
#include <rw/tvordvec.h>
#include <rw/rstream.h>

main() {

  RWTStack<int, RWTValOrderedVector<int> > stack;

  stack.push(1);
  stack.push(5);
  stack.push(6);

  while (!stack.isEmpty())
    cout << stack.pop() << endl;
  return 0;
}
```

Program output:

```
6
5
1
```

Public constructor

`RWTStack<T,C>();`
Constructs an empty stack of objects of type `T`, implemented using class `C`.

Public member functions

`void                         clear();`
Removes all items from the stack.

`size_tentries() const;`
Returns the number of items currently on the stack.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are currently no items on the stack, `FALSE` otherwise.

`voidpush(const T& a);`
Push the item `a` onto the top of the stack.

`Tpop();`
Pop (remove and return) the item at the top of the stack. If there are no items on the stack then an exception of type `TOOL_INDEX` will occur.

`Ttop() const;`
Returns (but does not remove) the item at the top of the stack.

RWTValDlist<T>

Synopsis

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

RWTValDlist<T> list;
```

Description

This class maintains a collection of values, implemented as a doubly linked list. This is a *value* based list: objects are copied in and out of the links that make up the list. Unlike intrusive lists (see class `RWTIsvDlist<T>`), the objects need not inherit from a link class. However, this makes the class slightly less efficient than the intrusive lists because of the need to allocate a new link off the heap with every insertion and to make a copy of the object in the newly allocated link.

Parameter `T` represents the type of object to be inserted into the list, either a class or built in type. The class `T` must have:

- A default constructor;
- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- well-defined equality semantics (`T::operator==(const T&)`).

Example

In this example, a doubly-linked list of user type Dog is exercised.

Code Example 23-6 (1 of 2)

```
#include <rw/tvdlist.h>
#include <rw/rstream.h>
#include <string.h>

class Dog {
  char* name;
public:
  Dog( const char* c = "") {
    name = new char[strlen(c)+1];
    strcpy(name, c); }

  ~Dog() { delete name; }

  // Define a copy constructor:
```

Code Example 23-6 (2 of 2)

```
  Dog(const Dog& dog) {
    name = new char[strlen(dog.name)+1];
    strcpy(name, dog.name); }

  // Define an assignment operator:
  void operator=(const Dog& dog) {
    if (this!=&dog) {
      delete name;
      name = new char[strlen(dog.name)+1];
      strcpy(name, dog.name);
    }
  }

  // Define an equality test operator:
  int operator==(const Dog& dog) const {
    return strcmp(name, dog.name)==0; }

  friend ostream& operator<<(ostream& str, Dog& dog){
    str << dog.name;
    return str;}
};

main()
{
  RWTValDlist<dog> terriers;
  terriers.insert("Cairn Terrier");   // NB: type conversion
occurs
  terriers.insert("Irish Terrier");
  terriers.insert("Schnauzer");

  cout << "The list " <<
    (terriers.contains("Schnauzer") ? "does " : "does not ") <<
    "contain a Schnauzer\n";

  terriers.insertAt(
      terriers.index("Irish Terrier"),
      "Fox Terrier"
    );

  while (!terriers.isEmpty())
    cout << terriers.get() << endl;

  return 0;
}
```

Program output:

```
The list does contain a Schnauzer
Cairn Terrier
Fox Terrier
Irish Terrier
Schnauzer
```

Public constructors

`RWTValDlist<T>();`
Construct an empty list.

`RWTValDlist<T>(const RWTValDlist<T>& list);`
Construct a copy of the list `list`. Depending on the nature of the copy constructor of `T`, this could be relatively expensive because every item in the list must be copied.

Public operators

```
RWTValDlist&                 operator=(const RWTValDlist<T>&
                              list);
```
Sets self to a copy of the list `list`. Depending on the nature of the copy constructor of `T`, this could be relatively expensive because every item in the list must be copied.

`T&operator[](size_t i);`
Returns a reference to the item at index `i`. The results can be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`Toperator[](size_t i) const;`
Returns a copy of the item at index `i`. The results cannot be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

Public member functions

```
void                         append(const T& a);
```
Adds the item a to the end of the list.

```
voidapply(void (*applyFun)(T&,
 void*), void* d);
```
Applies the user-defined function pointed to by `applyFun` to every item in the list. This function must have prototype:

`void` *yourFun*`(T& a, void* d);`

Client data may be passed through as parameter `d`.

`T&at(size_t i);`
Returns a reference to the item at index `i`. The results can be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`Tat(size_t i) const;`
Returns a copy of the item at index `i`. The results cannot be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`voidclear();`
Removes all items from the list. Their destructors (if any) will be called.

`RWBooleancontains(const T& a) const;`
Returns `TRUE` if the list contains an object that is equal to the object `a`. Returns `FALSE` otherwise. Equality is measured by the class-defined equality operator.

```
RWBoolean             contains(RWBoolean (*testFun)
                       (const T&, void*), void* d) const;
```
Returns `TRUE` if the list contains an item for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . Returns `FALSE` otherwise. The tester function must have the prototype:

```
    RWBoolean   yourTester(const T&, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`size_tentries() const;`
Returns the number of items that are currently in the collection.

`RWBooleanfind(const T& a, T& k) const;`
Returns `TRUE` if the list contains an object that is equal to the object a and puts a copy of the matching object into `k`. Returns `FALSE` otherwise and does not touch `k`. Equality is measured by the class-defined equality operator. If you do not need a copy of the found object, use `contains()` instead.

```
RWBooleanfind(RWBoolean (*testFun)
 (const T&, void*), void* d,
 T& k) const;
```
Returns `TRUE` if the list contains an object for which the user-defined tester

function pointed to by `testFun` returns `TRUE` and puts a copy of the matching object into `k`. Returns `FALSE` otherwise and does not touch `k`. The tester function must have the prototype:

RWBoolean *yourTester*(const T&, void* d);

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`. If you do not need a copy of the found object, use `contains()` instead.

```
Tfirst() const;
```
Returns (but does not remove) the first item in the list. The behavior is undefined if the list is empty.

```
Tget();
```
Returns and removes the first item in the list. The behavior is undefined if the list is empty.

```
size_tindex(const T& a);
```
Returns the index of the first object that is equal to the object `a`, or `RW_NPOS` if there is no such object. Equality is measured by the class-defined equality operator.

```
size_tindex(RWBoolean (*testFun)
 (const T&, void*), void* d)
 const;
```
Returns the index of the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or `RW_NPOS` if there is no such object. The tester function must have the prototype:

RWBoolean *yourTester*(const T&, void* d);

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
void insert(const T& a);
```
Adds the item a to the end of the list.

```
voidinsertAt(size_t i, const T& a);
```
Insert the item a at the index position `i`. This position must be between zero and the number of items in the list, or an exception of type `TOOL_INDEX` will be thrown.

```
RWBooleanisEmpty() const;
```
Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`Tlast() const;`
Returns (but does not remove) the last item in the list. The behavior is undefined if the list is empty.

`size_toccurrencesOf(const T& a) const;`
Returns the number of objects in the list that are equal to the object `a`. Equality is measured by the class-defined equality operator.

```
size_t                occurrencesOf(RWBoolean (*testFun)
                       (const T&, void*),void* d) const;
```
Returns the number of objects in the list for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . The tester function must have the prototype:

```
RWBoolean   yourTester(const T&, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`voidprepend(const T& a);`
Adds the item a to the beginning of the list.

`RWBooleanremove(const T& a);`
Removes the first object which is equal to the object `a` and returns `TRUE`. Returns `FALSE` if there is no such object. Equality is measured by the class-defined equality operator.

```
RWBooleanremove(RWBoolean (*testFun)
 (const T&, void*),void* d);
```
Removes the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE`, and returns `TRUE`. Returns `FALSE` if there is no such object. The tester function must have the prototype:

```
RWBoolean   yourTester(const T&, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`size_tremoveAll(const T& a);`
Removes all objects which are equal to the object `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator.

`size_tremoveAll(RWBoolean (*testFun)`
` (const T&, void*), void* d);`
Removes all objects for which the user-defined tester function pointed to by `testFun` returns `TRUE`. Returns the number of objects removed. The tester function must have the prototype:

```
    RWBoolean  yourTester(const T&, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`TremoveAt(size_t i);`
Removes and returns the object at index `i`. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`TremoveFirst();`
Removes and returns the first item in the list. The behavior is undefined if the list is empty.

`TremoveLast()`
Removes and returns the last item in the list. The behavior is undefined if the list is empty.

RWTValDlistIterator<T>

Synopsis

```
#include <rw/tvdlist.h>
RWTValDlist<T> list;
RWTValDlistIterator<T> iterator(list);
```

Description

Iterator for class `RWTValDlist<T>`, allowing sequential access to all the elements of a doubly-linked parameterized list. Elements are accessed in order, in either direction.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTValDlistIterator<T>(RWTValDlist<T>& c);`
Constructs an iterator to be used with the list `c`.

Public member operators

`RWBoolean                    operator++();`
Advances the iterator to the next item and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator--();`
Retreats the iterator to the previous item and returns `TRUE`. When the beginning of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator+=(size_t n);`
Advances the iterator `n` positions and returns `TRUE`. When the end of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator-=(size_t n);`
Retreats the iterator `n` positions and returns `TRUE`. When the beginning of the collection is reached, returns `FALSE` and the position of the iterator will be undefined.

`RWBooleanoperator()();`
Advances the iterator to the next item. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

Public member functions

`RWTValDlist<T>*            container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`RWBooleanfindNext(const T& a);`
Advances the iterator to the first element that is equal to a and returns `TRUE`, or `FALSE` if there is no such element. Equality is measured by the class-defined equality operator for type `T`.

```
RWBooleanfindNext(RWBoolean (*testFun)
 (const T&, void*), void*);
```
Advances the iterator to the first element for which the tester function pointed to by `testFun` returns `TRUE` and returns `TRUE`, or `FALSE` if there is no such element.

`voidinsertAfterPoint(const T& a);`
Inserts the value `a` into the iterator's associated collection in the position immediately after the iterator's current position.

`Tkey() const;`
Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

`RWBooleanremove();`
Removes the value from the iterator's associated collection at the current position of the iterator. Returns `TRUE` if successful, `FALSE` otherwise. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

`RWBoolean removeNext(const T& a);`
Advances the iterator to the first element that is equal to `a` and removes it. Returns `TRUE` if successful, `FALSE` otherwise. Equality is measured by the class-defined equality operator for type `T`. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

```
RWBoolean removeNext(RWBoolean (*testFun)
 (const T&, void*), void*);
```
Advances the iterator to the first element for which the tester function pointed to by `testFun` returns `TRUE` and removes it. Returns `TRUE` if successful, `FALSE` otherwise. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

`void reset();`
Resets the iterator to the state it had immediately after construction.

`void reset(RWTValDlist<T>& c);`
Resets the iterator to iterate over the collection `c`.

RWTValHashDictionary<K,V>

Synopsis

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

unsigned hashFun(const K&);

RWTValHashDictionary<K,V> dictionary(hashFun);
```

Description

`RWTValHashDictionary<K,V>` is a dictionary of keys of type `K` and values of type `V`, implemented using a hash table. While duplicates of values are allowed, duplicates of keys are not.

It is a *value* based collection: keys and values are copied in and out of the hash buckets.

Parameters `K` and `V` represent the type of the key and the type of the value, respectively, to be inserted into the table. These can be either classes or built in types. Classes `K` and `V` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.).

In addition, class `K` must have

- well-defined equality semantics (`K::operator==(const K&)`.

A user-supplied hashing function for type `K` must be supplied to the constructor when creating a new table. If `K` is a " class, then this requirement is usually trivial because all Tools.h++ objects know how to return a hashing value. This function has prototype:

```
    unsigned hFun(const K& a);
```

and should return a suitable hash value for the object `a`.

To find a value, the key is first hashed to determine in which bucket the key and value can be found. The bucket is then searched for an object that is equal (as determined by the equality operator) to the key.

The initial number of buckets in the table is set by the constructor. There is a default value. If the number of (key/value) pairs in the collection greatly exceeds the number of buckets then efficiency will sag because each bucket must be searched linearly. The number of buckets can be changed by calling member function `resize()`. This is an expensive proposition because not only must all the items be copied into the new buckets, but all of the keys must be rehashed.

If you wish this to be done automatically, then you can subclass from this class and implement your own special `insert()` and `remove()` functions which perform a `resize()` as necessary.

Example

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

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

main()
{
  RWTValHashDictionary<RWCString, RWDate> birthdays(hashString);

  birthdays.insertKeyAndValue("John", RWDate(12, "April",
1975));
  birthdays.insertKeyAndValue("Ivan", RWDate(2, "Nov", 1980));

  // Alternative syntax:
  birthdays["Susan"] = RWDate(30, "June", 1955);
  birthdays["Gene"] = RWDate(5, "Jan", 1981);

  // Print a birthday:
  cout << birthdays["John"] << endl;
  return 0;
}
```

Program output:

```
April 12, 1975
```

Public constructors

```
RWTValHashDictionary<K,V>(unsigned (*hashKey)(const K&),
 size_t buckets = RWDEFAULT_CAPACITY);
```

Constructs a new hash dictionary. The first argument is a pointer to a user-defined hashing function for items of type `K` (the key). The table will initally have `buckets` buckets although this can be changed with member function `resize()`.

```
RWTValHashDictionary<K,V>(const RWTValHashDictionary<K,V>&
 dict);
```
Copy constructor. Constructs a new hash dictionary as a copy of `dict`. The new dictionary will have the same number of buckets as the old table. Hence, although the keys and values must be copied into the new table, the keys will not be rehashed.

Public operators

```
RWTValHashDictionary<K,V>&
 operator=(const RWTValHashDictionary<K,V>& dict);
```
Sets self to a copy of `dict`. Afterwards, the new table will have the same number of buckets as the old table. Hence, although the keys and values must be copied into the new table, the keys will not be rehashed.

```
V&operator[](const K& key);
```
Look up the key `key` and return its associated value as an lvalue reference. If the key is not in the dictionary, then it is added to the dictionary. In this case, the value associated with the key will be provided by the default constructor for objects of type `V`.

Public member functions

```
void                      applyToKeyAndValue(void
                           (*applyFun)(const K&,V&,void*),
                           void* d);
```
Applies the user-defined function pointed to by `applyFun` to every key-value pair in the dictionary. This function must have prototype:

```
    void yourFun(const K& key, V& value, void* d);
```

The key will be passed by value and hence cannot be changed. The value will be passed by reference and can be modified. Client data may be passed through as parameter `d`.

```
voidclear();
```
Removes all items from the collection.

`RWBooleancontains(const K& key) const;`
Returns TRUE if the dictionary contains a key which is equal to key. Returns FALSE otherwise. Equality is measured by the class-defined equality operator for class K.

`size_tentries() const;`
Returns the number of key-value pairs currently in the dictionary.

```
RWBooleanfind(const K& key, K& retKey)
 const;
```
Returns TRUE if the dictionary contains a key which is equal to `key` and puts the matching *key* into `retKey`. Returns FALSE otherwise and leaves `retKey` untouched. Equality is measured by the class-defined equality operator for class K.

```
RWBooleanfindValue(const K& key, V&
 retVal) const;
```
Returns TRUE if the dictionary contains a key which is equal to `key` and puts the associated *value* into `retVal`. Returns FALSE otherwise and leaves `retVal` untouched. Equality is measured by the class-defined equality operator for class K.

```
RWBooleanfindKeyAndValue(const K& key, K&
 retKey, V& retVal) const;
```
Returns TRUE if the dictionary contains a key which is equal to `key` and puts the matching *key* into `retKey` and the associated *value* into `retVal`. Returns FALSE otherwise and leaves `retKey` and `retVal` untouched. Equality is measured by the class-defined equality operator for class K.

```
voidinsertKeyAndValue(const K&
 key, V value);
```
Inserts the key key and value value into the dictionary.

`RWBooleanisEmpty() const;`
Returns TRUE if the dictionary has no items in it, FALSE otherwise.

`RWBooleanremove(const K& key);`
Returns TRUE and removes the (key/value) pair where the key is equal to the `key`. Returns FALSE if there is no such key. Equality is measured by the class-defined equality operator for class K.

`voidresize(size_t N);`
Changes the number of buckets to `N`, a relatively expensive operation if there are many items in the collection.

RWTValHashDictionaryIterator<K,V>

Synopsis

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

unsigned hashFun(const K&);

RWTValHashDictionary<K,V> dictionary(hashFun);

RWTValHashDictonaryIterator<K,V> iterator(dictionary);
```

Description

Iterator for class `RWTValHashDictionary<K,V>`, allowing sequential access to all keys and values of a parameterized hash dictionary. Elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTValHashDictionaryIterator(RWTValHashDictionary& c);`
Constructs an iterator to be used with the dictionary `c`.

Public operators

`RWBoolean                    operator++();`
Advances the iterator one position. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

`RWBooleanoperator()();`
Advances the iterator one position. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

Public member functions

`RWTValHashDictionary*      container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`Kkey() const;`
Returns the key at the iterator's current position. The results are undefined if the iterator is no longer valid.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTValHashDictionary& c);`
Resets the iterator to iterate over the collection `c`.

`Vvalue() const;`
Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

RWTValHashSet<T>

RWTValHashSet<T>
|
RWTValHashTable<T>

Synopsis

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

unsigned hashFun(const T&);

RWTValHashSet(hashFun) set;
```

Description

`RWTValHashSet<T>` is a derived class of `RWTValHashTable<T>` where the `insert()` function has been overridden to accept only one item of a given value. Hence, each item in the collection will be unique.

As with class `RWTValHashTable<T>`, you must supply a hashing function to the constructor.

The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- well-defined equality semantics (`T::operator==(const T&)`).

Example

This examples exercises a set of `RWCStrings`.

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

unsigned hashIt(const RWCString& str){ return str.hash(); }

main()
{
  RWTValHashSet<RWCString> set(hashIt);

  set.insert("one");
  set.insert("two");
  set.insert("three");
  set.insert("one");   // Rejected: already in collection

  cout << set.entries() << endl;// Prints "3"
  return 0;
}
```

Program output:

```
3
```

Public constructors

```
RWTValHashSet<T>(unsigned (*hashFun)(const T&),
                              size_t buckets =
                              RWDEFAULT_CAPACITY);
```

Constructs a new hash table. The first argument is a pointer to a user-defined hashing function for items of type `T`. The table will initally have `buckets` buckets although this can be changed with member function `resize()`.

```
RWTValHashSet<T>(const RWTValHashSet<T>& table);
```

Constructs a new hash table as a copy of `table`. The new table will have the same number of buckets as the old table. Hence, although objects must be copied into the new table, they will not be hashed.

Public operators

`RWTValHashSet&` `operator=(const RWTValHashSet<T>&);`

Sets self to a copy of `table`. Afterwards, the new table will have the same number of buckets as the old table. Hence, although objects must be copied into the new table, they will not be hashed.

Public member functions

`void` `apply(void (*applyFun)(const T&, void*), void* d);`

Inherited from class `RWTValHashTable<T>`.

`voidclear();`

Inherited from class `RWTValHashTable<T>`.

`RWBooleancontains(const T& val) const;`

Inherited from class `RWTValHashTable<T>`.

`size_tentries() const;`

Inherited from class `RWTValHashTable<T>`.

`RWBooleanfind(const T& a, T& k) const;`

Inherited from class `RWTValHashTable<T>`.

`voidinsert(const T& val);`

Redefined from class `RWTValHashTable<T>` to allow an object of a given value to be inserted only once.

`RWBooleanisEmpty() const;`

Inherited from class `RWTValHashTable<T>`.

`size_toccurrencesOf(const T& val) const;`

Inherited from class `RWTValHashTable<T>`.

`RWBooleanremove(const T& val);`

Inherited from class `RWTValHashTable<T>`.

`size_tremoveAll(const T& val);`

Inherited from class `RWTValHashTable<T>`.

`voidresize(size_t N);`

Inherited from class `RWTValHashTable<T>`.

RWTValHashTable<T>

Synopsis

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

unsigned hashFun(const T&);

RWTValHashTable<T> table(hashFun);
```

Description

This class implements a parameterized hash table of types `T`. It uses chaining to resolve hash collisions. Duplicates are allowed.

It is a *value* based collection: objects are copied in and out of the hash buckets.

Parameter `T` represents the type of object to be inserted into the table, either a class or built in type. The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- well-defined equality semantics (`T::operator==(const T&)`).

A user-supplied hashing function for type `T` must be supplied to the constructor when creating a new table. If T is a Tools.h++ class, then this requirement is usually trivial because all Tools.h++ objects know how to return a hashing value. This function has prototype:

```
    unsigned hFun(const T& a);
```

and should return a suitable hash value for the object `a`.

To find an object, it is first hashed to determine in which bucket it occurs. The bucket is then searched for an object that is equal (as determined by the equality operator) to the candidate.

The initial number of buckets in the table is set by the constructor. There is a default value. If the number of items in the collection greatly exceeds the number of buckets then efficiency will sag because each bucket must be searched linearly. The number of buckets can be changed by calling member function `resize()`. This is an expensive proposition because not only must all items be copied into the new buckets, but they must also be rehashed.

If you wish this to be automatically done, then you can subclass from this class and implement your own special `insert()` and `remove()` functions which perform a `resize()` as necessary.

Example

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

unsigned hashIt(const RWCString& str) {return str.hash();}

main()
{
  RWTValHashTable<RWCString> table(hashIt);

  table.insert("Alabama");   // NB: Type conversion occurs
  table.insert("Pennsylvania");
  table.insert("Oregon");
  table.insert("Montana");

  cout << "The table " <<
    (table.contains("Oregon") ? "does " : "does not ") <<
    "contain Oregon\n";

  table.removeAll("Oregon");

  cout << "The table " <<
    (table.contains("Oregon") ? "does " : "does not ") <<
    "contain Oregon";
  return 0;
}
```

Program output:

```
The table does contain Oregon
The table does not contain Oregon
```

Public constructors

```
RWTValHashTable<T>(unsigned (*hashFun)(const T&),
                            size_t buckets =
                            RWDEFAULT_CAPACITY);
```

Constructs a new hash table. The first argument is a pointer to a user-defined hashing function for items of type `T`. The table will initally have `buckets` buckets although this can be changed with member function `resize()`.

```
RWTValHashTable<T>(const RWTValHashTable<T>& table);
```

Constructs a new hash table as a copy of `table`. The new table will have the same number of buckets as the old table. Hence, although objects must be copied into the new table, they will not be hashed.

Public operators

```
RWTValHashTable&           operator=(const
                            RWTValHashTable<T>&);
```

Sets self to a copy of `table`. Afterwards, the new table will have the same number of buckets as the old table. Hence, although objects must be copied into the new table, they will not be hashed.

Public member functions

```
void                        apply(void (*applyFun)(const T&,
                             void*), void* d);
```

Applies the user-defined function pointed to by `applyFun` to every item in the table. This function must have prototype:

`void` *yourFun*`(const T& a, void* d);`

Client data may be passed through as parameter `d`.

```
voidclear();
```

Removes all items from the collection.

```
RWBooleancontains(const T& val) const;
```

Returns `TRUE` if the collection contains an item which is equal to `val`. Returns `FALSE` otherwise. Equality is measured by the class-defined equality operator.

```
size_tentries() const;
```

Returns the number of items currently in the collection.

```
RWBooleanfind(const T& a, T& k) const;
```

Returns `TRUE` if the collection contains an item which is equal to `val` and puts the matching object into `k`. Returns `FALSE` otherwise and leaves `k` untouched. Equality is measured by the class-defined equality operator.

`voidinsert(const T& val);`
Inserts the value val into the collection.

`RWBooleanisEmpty() const;`
Returns `TRUE` if the collection has no items in it, `FALSE` otherwise.

```
size_toccurrencesOf(const T& val)
 const;
```
Returns the number of items in the collection which are equal to `val`. Equality is measured by the class-defined equality operator.

`RWBooleanremove(const T& val);`
Removes the first object which is equal to the object `a` and returns `TRUE`. Returns `FALSE` if there is no such object. Equality is measured by the class-defined equality operator.

`size_tremoveAll(const T& val);`
Removes all objects which are equal to the object `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator.

`voidresize(size_t N);`
Changes the number of buckets to `N`, a relatively expensive operation if there are many items in the collection.

RWTValHashTableIterator<T>

Synopsis

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

RWTValHashTable<T> table;

RWTValHashTableIterator<T> iterator(table);
```

Description

Iterator for class `RWTValHashTable<T>`, allowing sequential access to all the elements of a hash table. Elements are not accessed in any particular order.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTValHashTableIterator(RWTValHashTable<T>& c);`
Constructs an iterator to be used with the table `c`.

Public operators

`RWBoolean                    operator++();`
Advances the iterator one position. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

`RWBooleanoperator()();`
Advances the iterator one position. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

Public member functions

`RWTValHashTable<T>*          container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`Tkey() const`
Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

`voidreset();`
Resets the iterator to the state it had immediately after construction.

`voidreset(RWTValHashTable<T>& c);`
Resets the iterator to iterate over the collection `c`.

RWTValOrderedVector<T>

Synopsis

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

```
RWTValOrderedVector<T> ordvec;
```

Description

`RWTValOrderedVector<T>` is an *ordered* collection. That is, the items in the collection have a meaningful ordered relationship with respect to one another and can be accessed by an index number. The order is set by the order of insertion. Duplicates are allowed. The class is implemented as a vector, allowing efficient insertion and retrieval from the end of the collection, but somewhat slower from the beginning of the collection.

The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- well-defined equality semantics (`T::operator==(const T&)`);
- a default constructor.

Note that an ordered vector has a *length* (the number of items returned by `length()` or `entries()`) and a *capacity*. Necessarily, the capacity is always greater than or equal to the length. Although elements beyond the collection's length are not used, nevertheless, in a value-based collection, they are occupied. If each instance of class `T` requires considerable resources, then you should insure that the collection's capacity is not much greater than its length, otherwise unnecessary resources will be tied up.

Example

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

main() {

  RWTValOrderedVector<double> vec;

  vec.insert(22.0);
  vec.insert(5.3);
  vec.insert(-102.5);
  vec.insert(15.0);
  vec.insert(5.3);

  cout << vec.entries() << " entries\n" << endl;   // Prints "5"
  for (int i=0; i<vec.length(); i++)
    cout << vec[i] << endl;

  return 0;
}
```

Program output:

```
5 entries
22
5.3
-102.5
15
5.3
```

Public constructor

`RWTValOrderedVector<T>(size_t capac=RWDEFAULT_CAPACITY);`
Create an empty ordered vector with capacity `capac`. Should the number of items exceed this value, the vector will be resized automatically.

`RWTValOrderedVector<T>(const RWTValOrderedVector<T>& c);`
Constructs a new ordered vector as a copy of `c`. The copy constructor of all elements in the vector will be called. The new vector will have the same capacity and number of members as the old vector.

Public operators

```
RWTValOrderedVector<T>&    operator=(const
                              RWTValOrderedVector& c);
```

Sets self to a copy of `c`. The copy constructor of all elements in the vector will be called. Self will have the same capacity and number of members as the old vector.

```
T&operator()(size_t i);
Toperator()(size_t i) const;
```

Returns the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one. No bounds checking is performed.

```
T&operator[](size_t i);
Toperator[](size_t i) const;
```

Returns the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

```
void                         append(const T& a);
```

Appends the value `a` to the end of the vector. The collection will automatically be resized if this causes the number of items in the collection to exceed the capacity.

```
T&at(size_t i);
Tat(size_t i) const;
```

Return the `i`'th value in the vector. The first variant can be used as an lvalue, the second cannot. The index `i` must be between 0 and the length of the vector less one, or an exception of type `TOOL_INDEX` will be thrown .

```
voidclear();
```

Removes all items from the collection.

```
RWBooleancontains(const T& a) const;
```

Returns `TRUE` if the collection contains an item that is equal to `a`. A linear search is done. Equality is measured by the class-defined equality operator.

```
const T*data() const;
```

Returns a pointer to the raw data of the vector. The contents should not be changed. Should be used with care.

```
size_tentries() const;
```

Returns the number of items currently in the collection.

`RWBooleanfind(const T& a,T& ret) const;`
Performs a linear search and returns `TRUE` if the vector contains an object that is equal to the object a and puts a copy of the matching object into `ret`. Returns `FALSE` otherwise and does not touch `ret`. Equality is measured by the class-defined equality operator.

`Tfirst() const;`
Returns the first item in the collection. An exception of type `TOOL_INDEX` will occur if the vector is empty.

`size_tindex(const T& a) const;`
Performs a linear search, returning the index of the first item that is equal to `a`. Returns `RW_NPOS` if there is no such item. Equality is measured by the class-defined equality operator.

`voidinsert(const T& a);`
Appends the value `a` to the end of the vector. The collection will automatically be resized if this causes the number of items in the collection to exceed the capacity.

`voidinsertAt(size_t i, const T& a);`
Inserts the value a into the vector at index `i`. The item previously at position `i` is moved to `i+1`, *etc*. The collection will automatically be resized if this causes the number of items in the collection to exceed the capacity. The index `i` must be between 0 and the number of items in the vector or an exception of type `TOOL_INDEX` will occur.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

`Tlast() const;`
Returns the last item in the collection. If there are no items in the collection then an exception of type `TOOL_INDEX` will occur.

`size_tlength() const;`
Returns the number of items currently in the collection.

`size_toccurrencesOf(const T& a) const;`
Performs a linear search, returning the number of items that are equal to `a`. Equality is measured by the class-defined equality operator.

`voidprepend(const T& a);`
Prepends the value `a` to the beginning of the vector. The collection will automatically be resized if this causes the number of items in the collection to exceed the capacity.

`RWBooleanremove(const T& a);`
Performs a linear search, removing the first object which is equal to the object `a` and returns `TRUE`. Returns `FALSE` if there is no such object. Equality is measured by the class-defined equality operator.

`size_tremoveAll(const T& a);`
Removes all items which are equal to `a`, returning the number removed. Equality is measured by the class-defined equality operator.

`TremoveAt(size_t i);`
Removes and returns the object at index `i`. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`TremoveFirst();`
Removes and returns the first object in the collection. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`TremoveLast();`
Removes and returns the last object in the collection. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`voidresize(size_t N);`
Changes the capacity of the collection to `N`. Note that the number of objects in the collection does not change, just the capacity.

RWTValSlist<T>

Synopsis

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

RWTValSlist<T> list;
```

Description

This class maintains a collection of values, implemented as a singly linked list. This is a *value* based list: objects are copied in and out of the links that make up the list. Unlike intrusive lists (see class `RWTIsvSlist<T>`) the objects need not inherit from a link class. However, this makes the class slightly less efficient than the intrusive lists because of the need to allocate a new link off the heap with every insertion and to make a copy of the object in the newly allocated link.

Parameter `T` represents the type of object to be inserted into the list, either a class or built in type. The class T must have:

- A default constructor;
- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- well-defined equality semantics (`T::operator==(const T&)`).

Example

In this example, a singly-linked list of `RWDates` is exercised.

Code Example 23-7

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


main()
{
  RWTValSlist<RWDate> dates;
  dates.insert(RWDate(2, "June", 52));// 6/2/52
  dates.insert(RWDate(30, "March", 46));// 3/30/46
  dates.insert(RWDate(1, "April", 90));// 4/1/90

  // Now look for one of the dates:
  RWDate ret;
```

Code Example 23-7 (Continued)

```
   if (dates.find(RWDate(2, "June", 52), ret)){
     cout << "Found date " << ret << endl;
   }

   // Remove in reverse order:
   while (!dates.isEmpty())
     cout << dates.removeLast() << endl;

   return 0;
 }
```

Program output:

```
Found date June 2, 1952
April 1, 1990
March 30, 1946
June 2, 1952
```

Public constructors

`RWTValSlist<T>();`
Construct an empty list.

`RWTValSlist<T>(const RWTValSlist<T>& list);`
Construct a copy of the list `list`. Depending on the nature of the copy constructor of `T`, this could be relatively expensive because every item in the list must be copied.

Public operators

`RWTValSlist&<T>operator=(const RWTValSlist<T>& list);`
Sets self to a copy of the list list. Depending on the nature of the copy constructor of `T`, this could be relatively expensive because every item in the list must be copied.

`T&operator[](size_t i);`
Returns a reference to the item at index `i`. The results can be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`T operator[](size_t i) const;`
Returns a copy of the item at index `i`. The results cannot be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

Public member functions

`void append(const T& a);`
Adds the item `a` to the end of the list.

```
void apply(void (*applyFun)(T&,
 void*), void* d);
```
Applies the user-defined function pointed to by `applyFun` to every item in the list. This function must have prototype:

`void` *yourFun*`(T& a, void* d);`

Client data may be passed through as parameter `d`.

`T& at(size_t i);`
Returns a reference to the item at index `i`. The results can be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`T at(size_t i) const;`
Returns a copy of the item at index `i`. The results cannot be used as an lvalue. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`void clear();`
Removes all items from the list. Their destructors (if any) will be called.

`RWBoolean contains(const T& a) const;`
Returns `TRUE` if the list contains an object that is equal to the object `a`. Returns `FALSE` otherwise. Equality is measured by the class-defined equality operator.

```
RWBoolean contains(RWBoolean (*testFun)
 (const T&, void*), void* d)
 const;
```
Returns `TRUE` if the list contains an item for which the user-defined "tester" function pointed to by `testFun` returns `TRUE` . Returns `FALSE` otherwise. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T&, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

```
size_tentries() const;
```
Returns the number of items that are currently in the collection.

```
RWBooleanfind(const T& a, T& k) const;
```
Returns `TRUE` if the list contains an object that is equal to the object `a` and puts a copy of the matching object into `k`. Returns `FALSE` otherwise and does not touch `k`. Equality is measured by the class-defined equality operator. If you do not need a copy of the found object, use `contains()` instead.

```
RWBooleanfind(RWBoolean (*testFun)
 (const T&, void*), void* d,
 T& k) const;
```
Returns `TRUE` if the list contains an object for which the user-defined tester function pointed to by `testFun` returns `TRUE` and puts a copy of the matching object into `k`. Returns `FALSE` otherwise and does not touch `k`. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T&, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`. If you do not need a copy of the found object, use `contains()` instead.

```
Tfirst() const;
```
Returns (but does not remove) the first item in the list. The behavior is undefined if the list is empty.

```
Tget();
```
Returns and removes the first item in the list. The behavior is undefined if the list is empty.

```
size_tindex(const T& a);
```
Returns the index of the first object that is equal to the object `a`, or `RW_NPOS` if there is no such object. Equality is measured by the class-defined equality operator.

```
size_tindex(RWBoolean (*testFun)
 (const T&, void*), void* d)
 const;
```

Returns the index of the first object for which the user-defined tester function pointed to by `testFun` returns `TRUE`, or `RW_NPOS` if there is no such object. The tester function must have the prototype:

```
    RWBoolean   yourTester(const T&, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`void insert(const T& a);`
Adds the item `a` to the end of the list.

`voidinsertAt(size_t i, const T& a);`
Insert the item a at the index position `i`. This position must be between zero and the number of items in the list, or an exception of type `TOOL_INDEX` will be thrown.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`Tlast() const;`
Returns (but does not remove) the last item in the list. The behavior is undefined if the list is empty.

`size_toccurrencesOf(const T& a) const;`
Returns the number of objects in the list that are equal to the object `a`. Equality is measured by the class-defined equality operator.

```
size_t              occurrencesOf(RWBoolean (*testFun)
                     (const T&, void*), void* d) const;
```
Returns the number of objects in the list for which the user-defined "tester" function pointed to by testFun returns `TRUE` . The tester function must have the prototype:

```
    RWBoolean   yourTester(const T&, void* d);
```

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`voidprepend(const T& a);`
Adds the item `a` to the beginning of the list.

`RWBooleanremove(const T& a);`
Removes the first object which is equal to the object a and returns TRUE. Returns FALSE if there is no such object. Equality is measured by the class-defined equality operator.

```
RWBooleanremove(RWBoolean (*testFun)
 (const T&, void*), void* d);
```
Removes the first object for which the user-defined tester function pointed to by `testFun` returns TRUE, and returns TRUE. Returns FALSE if there is no such object. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T&, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`size_tremoveAll(const T& a);`
Removes all objects which are equal to the object `a`. Returns the number of objects removed. Equality is measured by the class-defined equality operator.

```
size_tremoveAll(RWBoolean (*testFun)
 (const T&, void*), void* d);
```
Removes all objects for which the user-defined tester function pointed to by `testFun` returns TRUE. Returns the number of objects removed. The tester function must have the prototype:

`RWBoolean` *yourTester*`(const T&, void* d);`

For each item in the list this function will be called with the item as the first argument. Client data may be passed through as parameter `d`.

`TremoveAt(size_t i);`
Removes and returns the object at index `i`. An exception of type `TOOL_INDEX` will be thrown if i is not a valid index. Valid indices are from zero to the number of items in the list less one.

`TremoveFirst();`
Removes and returns the first item in the list. The behavior is undefined if the list is empty.

`TremoveLast()`
Removes and returns the last item in the list. The behavior is undefined if the list is empty. This function is relatively slow because removing the last link in a singly-linked list necessitates access to the next-to-the-last link, requiring the whole list to be searched.

RWTValSlistIterator<T>

Synopsis

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

RWTValSlist<T< list;

RWTValSlistIterator<T> iterator(list);
```

Description

Iterator for class `RWTValSlist<T>`, allowing sequential access to all the elements of a singly-linked parameterized list. Elements are accessed in order, from first to last.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

Public constructor

`RWTValSlistIterator<T>(RWTValSlist<T>& c);`
Constructs an iterator to be used with the list `c`.

Public member operators

`RWBoolean                operator++();`
Advances the iterator one position. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

`RWBooleanoperator+=(size_t n);`
Advances the iterator n positions. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

`RWBooleanoperator()();`
Advances the iterator one position. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

Public member functions

`RWTValSlist<T>*            container() const;`
Returns a pointer to the collection over which this iterator is iterating.

`RWBooleanfindNext(const T& a);`
Advances the iterator to the first element that is equal to a and returns TRUE, or FALSE if there is no such element. Equality is measured by the class-defined equality operator for type T.

```
RWBooleanfindNext(RWBoolean (*testFun)
 (const T&, void*),void*);
```
Advances the iterator to the first element for which the tester function pointed to by testFun returns TRUE and returns TRUE, or FALSE if there is no such element.

`voidinsertAfterPoint(const T& a);`
Inserts the value a into the iterator's associated collection in the position immediately after the iterator's current position.

`Tkey() const;`
Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

`RWBooleanremove();`
Removes the value from the iterator's associated collection at the current position of the iterator. Returns TRUE if successful, FALSE otherwise. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element. This function is relatively inefficient for a singly-linked list.

`RWBooleanremoveNext(const T& a);`
Advances the iterator to the first element that is equal to a and removes it. Returns TRUE if successful, FALSE otherwise. Equality is measured by the class-defined equality operator for type T. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

```
RWBooleanremoveNext(RWBoolean (*testFun)
 (const T&, void*), void*);
```
Advances the iterator to the first element for which the tester function pointed to by testFun returns TRUE and removes it. Returns TRUE if successful, FALSE otherwise. Afterwards, if successful, the iterator will be positioned at the element immediately before the removed element.

```
voidreset();
```

Resets the iterator to the state it had immediately after construction.

```
voidreset(RWTValSlist& c);
```

Resets the iterator to iterate over the collection `c`.

RWTValSortedVector<T>

Synopsis

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

RWTValSortedVector<T> sortvec;
```

Description

`RWTValSortedVector<T>` is an *ordered* collection. That is, the items in the collection have a meaningful ordered relationship with respect to each other and can be accessed by an index number. In the case of `RWTValSortedVector<T>`, objects are inserted such that objects "less than" themselves are before the object, objects "greater than" themselves after the object. An insertion sort is used. Duplicates are allowed.

Stores a *copy* of the inserted item into the collection according to an ordering determined by the less-than (<) operator.

The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- well-defined equality semantics (`T::operator==(const T&)`);
- well-defined less-than semantics (`T::operator(const T&)`);
- a default constructor.

Note that a sorted vector has a *length* (the number of items returned by `length()` or `entries()`) and a *capacity*. Necessarily, the capacity is always greater than or equal to the length. Although elements beyond the collection's length are not used, nevertheless, in a value-based collection, they are occupied. If each instance of class T requires considerable resources, then you should insure that the collection's capacity is not much greater than its length, otherwise unnecessary resources will be tied up.

Example

This example inserts a set of dates into a sorted vector in no particular order, then prints them out in order.

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

main()
{
  RWTValSortedVector<RWDate> vec;

  vec.insert(RWDate(10, "Aug", 1991));
  vec.insert(RWDate(9, "Aug", 1991));
  vec.insert(RWDate(1, "Sept", 1991));
  vec.insert(RWDate(14, "May", 1990));
  vec.insert(RWDate(1, "Sept", 1991));   // Add a duplicate
  vec.insert(RWDate(2, "June", 1991));

  for (int i=0; i<vec.length(); i++)
    cout << vec[i] << endl;
  return 0;
}
```

Program output:

```
May 14, 1990
June 2, 1991
August 9, 1991
August 10, 1991
September 1, 1991
September 1, 1991
```

Public constructor

`RWTValSortedVector(size_t capac = RWDEFAULT_CAPACITY);`
Create an empty sorted vector with an initial capacity equal to `capac`. The vector will be automatically resized should the number of items exceed this amount.

Public operators

```
T&                         operator()(size_t i);
T                          operator()(size_t i) const;
```
Returns the i'th value in the vector. The first variant can be used as an l-value, the second cannot. The index i must be between zero and the number of items in the collection less one. No bounds checking is performed.

```
T&operator[](size_t i);
Toperator[](size_t i) const;
```
Returns the `i`'th value in the vector. The first variant can be used as an l-value, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

Public member functions

```
T&                         at(size_t i);
T                          at(size_t i) const;
```
Return the `i`'th value in the vector. The first variant can be used as an lvalue, the second cannot. The index `i` must be between 0 and the length of the vector less one, or an exception of type `TOOL_INDEX` will be thrown.

```
voidclear();
```
Removes all items from the collection.

```
RWBooleancontains(const T& a) const;
```
Returns `TRUE` if the collection contains an item that is equal to `a`. A binary search is done. Equality is measured by the class-defined equality operator.

```
const T*data() const;
```
Returns a pointer to the raw data of the vector. The contents should not be changed. Should be used with care.

```
size_tentries() const;
```
Returns the number of items currently in the collection.

```
RWBooleanfind(const T& a,T& ret) const;
```
Performs a binary search and returns `TRUE` if the vector contains an object that is equal to the object `a` and puts a copy of the matching object into `ret`. Returns `FALSE` otherwise and does not touch `ret`. Equality is measured by the class-defined equality operator.

```
Tfirst() const;
```
Returns the first item in the collection. An exception of type `TOOL_INDEX` will occur if the vector is empty.

`size_tindex(const T& a) const;`
Performs a binary search, returning the index of the first item that is equal to `a`. Returns `RW_NPOS` if there is no such item. Equality is measured by the class-defined equality operator.

`voidinsert(const T& a);`
Performs a binary search, inserting a after all items that compare less than or equal to it, but before all items that do not. "Less Than" is measured by the class-defined '<' operator for type `T`. The collection will be resized automatically if this causes the number of items to exceed the capacity.

`RWBooleanisEmpty() const;`
Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

Tlast() const;
Returns the last item in the collection. If there are no items in the collection then an exception of type `TOOL_INDEX` will occur.

`size_tlength() const;`
Returns the number of items currently in the collection.

`size_toccurrencesOf(const T& a) const;`
Performs a binary search, returning the number of items that are equal to `a`. Equality is measured by the class-defined equality operator.

`RWBooleanremove(const T& a);`
Performs a binary search, removing the first object which is equal to the object `a` and returns `TRUE`. Returns `FALSE` if there is no such object. Equality is measured by the class-defined equality operator.

`size_tremoveAll(const T& a);`
Removes all items which are equal to `a`, returning the number removed. Equality is measured by the class-defined equality operator.

`TremoveAt(size_t i);`
Removes and returns the object at index `i`. An exception of type `TOOL_INDEX` will be thrown if `i` is not a valid index. Valid indices are from zero to the number of items in the list less one.

`TremoveFirst();`
Removes and returns the first object in the collection. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`TremoveLast();`
Removes and returns the last object in the collection. An exception of type `TOOL_INDEX` will be thrown if the list is empty.

`voidresize(size_t N);`
Changes the capacity of the collection to `N`. Note that the number of objects in the collection does not change, just the capacity.

RWTValVector<T>

Synopsis

```
#include <rw/tvvector.h>
RWTValVector<T> vec;
```

Descripton

Class `RWTValVector<T>` is a simple parameterized vector of objects of type `T`. It is most useful when you know precisely how many objects have to be held in the collection. If the intention is to "insert" an unknown number of objects into a collection, then class `RWTValOrderedVector<T>` may be a better choice.

The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.);
- a default constructor.

Example

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

main() {

  RWTValVector<RWDate> week(7);

  RWDate begin;   // Today's date

  for (int i=0; i<7; i++)
    week[i] = begin++;

  for (i=0; i<7; i++)
    cout << week[i] << endl;

  return 0;
}
```

Program output:

```
    March 16, 1992
    March 17, 1992
    March 18, 1992
    March 19, 1992
    March 20, 1992
    March 21, 1992
    March 22, 1992
```

Public constructors

`RWTValVector<T>();`
Constructs an empty vector of length zero.

`RWTValVector<T>(size_t n);`
Constructs a vector of length `n`. The values of the elements will be set by the default constructor of class `T`. For a built in type this can (and probably will) be garbage.

`RWTValVector<T>(size_t n, const T& ival);`
Constructs a vector of length `n`, with each element initialized to the value `ival`.

`RWTValVector<T>(const RWTValVector& v);`
Constructs self as a copy of `v`. Each element in v will be *copied* into self.

`~RWTValVector<T>();`
Calls the destructor for every element in self.

Public operators

```
RWTValVector<T>&           operator=(const
                            RWTValVector<T>& v);
```
Sets self to the same length as `v` and then copies all elements of `v` into self.

`RWTValVector<T>&operator=(const T& ival);`
Sets all elements in self to the value `ival`.

```
Toperator()(size_t i) const;
T&operator()(size_t i);
```
Return the `i`'th value in the vector. The index `i` must be between 0 and the length of the vector less one. No bounds checking is performed.

`Toperator[](size_t i) const;`
`T&operator[](size_t i);`
Return the `i`'th value in the vector. The index `i` must be between 0 and the length of the vector less one. Bounds checking will be performed.

Public member functions

`const T*                    data() const;`
Returns a pointer to the raw data of self. Should be used with care.

`size_tlength() const;`
Returns the length of the vector.

`voidreshape(size_t N);`
Changes the length of the vector to `N`. If this results in the vector being lengthened, then the initial value of the additional elements is set by the default constructor of `T`.

RWTValVirtualArray<T>

Synopsis

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

```
RWVirtualPageHeap* heap;
```

```
RWTValVirtualArray<T> array(1000L, heap);
```

Description

This class represents a virtual array of elements of type `T` of almost any length. Individual elements are brought into physical memory on an "as needed" basis. If an element is used as an lvalue it is automatically marked as "dirty" and will be rewritten to the swapping medium.

The swap space is provided by an abstract page heap which is specified by the constructor. Any number of virtual arrays can use the same abstract page heap. You must take care that the destructor of the abstract page heap is not called before all virtual arrays built from it have been destroyed.

The class supports reference counting using a copy-on-write technique, so (for example) returning a virtual array by value from a function is as efficient as it can be. Be aware, however, that if the copy-on-write machinery finds that a copy must ultimately be made, then for large arrays this could take quite a bit of time.

For efficiency, more than one element can (and should) be put on a page. The actual number of elements is equal to the page size divided by the element size, rounded downwards. Example: for a page size of 512 bytes, and an element size of 8, then 64 elements would be put on a page.

The indexing operator (`operator[](long)`) actually returns an object of type `RWTVirtualElement<T>`. Consider this example:

```
    double d = vec[j];
    vec[i] = 22.0;
```

Assume that vec is of type `RWTValVirtualArray<double>`. The expression `vec[j]` will return an object of type `RWTVirtualElement<double>`, which will contain a reference to the element being addressed. In the first line, this expression is being used to initialize a double. The class `RWTVirtualElement<T>` contains a type conversion operator to convert itself to a T, in this case a double. The compiler uses this to initialize `d`. In the second line, the expression `vec[i]` is being used as an lvalue. In this case, the compiler uses the assignment operator for `RWTVirtualElement<T>`. This assignment operator recognizes that the expression is being used as an lvalue and automatically marks the appropriate page as "dirty", thus guaranteeing that it will be written back out to the swapping medium.

Slices, as well as individual elements, can also be addressed. These should be used wherever possible as they are much more efficient because they allow a page to be locked and used multiple times before unlocking.

The class `T` must have:

- well-defined copy semantics (`T::T(const T&)` or equiv.);
- well-defined assignment semantics (`T::operator=(const T&)` or equiv.).

In addition, you must never take the address of an element.

Example

In this example, a virtual vector of objects of type `ErsatzInt` is exercised. A disk-based page heap is used for swapping space.

Code Example 23-8

```
#include <rw/tvrtarry.h>
#include <rw/rstream.h>
#include <rw/diskpage.h>
#include <stdlib.h>
#include <stdio.h>

struct ErsatzInt {
  char  buf[8];
  ErsatzInt(int i) { sprintf(buf, "%d", i); }
  friend ostream& operator<<(ostream& str, ErsatzInt& i)
    { str << atoi(i.buf); return str; }
};

main() {

  RWDiskPageHeap heap;
  RWTValVirtualArray<ErsatzInt> vec1(10000L, &heap);

  for (long i=0; i<1000L; i++)
    vec1[i] = i;   // Some compilers may need a cast here

  cout << vec1[100] << endl;   // Prints "100"
```

Code Example 23-8 (Continued)

```
   cout << vec1[300] << endl;   // Prints "300"

   RWTValVirtualArray<EsatzInt> vec2 = vec.slice(5000L, 500L);
   cout << vec2.length() << endl;   // Prints "500"
   cout << vec2[0] << endl;    // Prints "5000";

   return 0;
 }
```

Program output:

```
100
300
500
5000
```

Public constructors

`RWTValVirtualArray<T>(long size, RWVirtualPageHeap* heap);`
Construct a vector of length size. The pages for the vector will be allocated from the page heap given by heap which can be of any type.

`RWTValVirtualArray<T>(const RWTValVirtualArray<T>& v);`
Constructs a vector as a copy of `v`. The resultant vector will use the same heap and have the same length as `v`. The actual copy will not be made until a write, minimizing the amount of heap allocations and copying that must be done.

`RWTValVirtualArray<T>(const RWTVirtualSlice<T>& sl);`
Constructs a vector from a *slice* of another vector. The resultant vector will use the same heap as the vector whose slice is being taken. Its length will be given by the length of the slice. The copy will be made immediately.

Public destructor

`~RWTValVirtualArray<T>();`
Releases all pages allocated by the vector.

Public operators

```
RWTValVirtualArray&
  operator=(const RWTValVirtualArray<T>& v);
```
Sets self to a copy of `v`. The resultant vector will use the same heap and have the same length as `v`. The actual copy will not be made until a write, minimizing the amount of heap allocations and copying that must be done.

`voidoperator=(const RWTVirtualSlice<T>& sl);`
Sets self equal to a *slice* of another vector. The resultant vector will use the same heap as the vector whose slice is being taken. Its length will be given by the length of the slice. The copy will be made immediately.

`Toperator=(const T& val);`
Sets all elements in self equal to `val`. This operator is actually quite efficient because it can work with many elements on a single page at once.

`Toperator[](long i) const;`
Returns a copy of the value at index `i`. The index `i` must be between zero and the length of the vector less one or an exception of type `TOOL_LONGINDEX` will occur.

`RWTVirtualElement<T>operator[](long);`
Returns a reference to the value at index `i`. The results can be used as an lvalue. The index `i` must be between zero and the length of the vector less one or an exception of type `TOOL_LONGINDEX` will occur.

Public member functions

`long                              length() const;`
Returns the length of the vector.

`Tval(long i) const;`
Returns a copy of the value at index `i`. The index i must be between zero and the length of the vector less one or an exception of type `TOOL_LONGINDEX` will occur.

`voidset(long i, (const T& v);`
Sets the value at the index `i` to `v`. The index `i` must be between zero and the length of the vector less one or an exception of type `TOOL_LONGINDEX` will occur.

`RWTVirtualSlice<T>slice(long start, long length);`
Returns a reference to a *slice* of self. The value `start` is the starting index of the slice, the value length its extent. The results can be used as an lvalue.

`voidreshape(long newLength);`
Change the length of the vector to `newLength`. If this results in the vector being lengthened then the value of the new elements is undefined.

`RWVirtualPageHeap*heap() const;`
Returns a pointer to the heap from which the vector is getting its pages.

Part 4— <generic.h> Classes

<generic.h> Classes 24

RWGBitVec(size)

Synopsis

```
#include <rw/gbitvec.h>
declare(RWGBitVec,size)

RWGBitVec(size) a;
```

Description

`RWGBitVec`(*size*) is a bit vector of fixed length *size*. The length cannot be changed dynamically (see class `RWBitVec` if you need a bit vector whose length can be changed at run time). Objects of type `RWGBitVec`(*size*) are declared with macros defined in the standard C++ header file <*generic.h*>.

Bits are numbered from 0 through *size*–1, inclusive.

Example

In this example, a bit vector 24 bits long is declared and exercised:

```
#include <rw/gbitvec.h>
declare(RWGBitVec,24)// declare a 24 bit long vector
implement(RWGBitVec, 24)// implement the vector
main()
{
  RWGBitVec(24) a, b;// Allocate two vectors.
  a(2) = TRUE;// Set bit 2 (the third bit) of
        a on.
  b(3) = TRUE;// Set bit 3 (the fourth bit
        of b on.
  RWGBitVec(24) c = a ^ b;// Set c to the XOR of a and b.
}
```

Public constructors

`RWGBitVec(`*size*`)();`
Construct a bit vector *size* elements long, with all bits initialized to `FALSE`.

`RWGBitVec(`*size*`)(RWBoolean f);`
Construct a bit vector *size* elements long, with all bits initialized to `f`.

`RWGBitVec(`*size*`)(unsigned long v);`
Construct a bit vector *size* elements long, initialized to the bits of `v`. If *size* is greater than `sizeof(v)`, the extra bits will be set to zero.

Assignment operators

```
RWGBitVec(sz)&           operator=(const RWGBitVec(sz)&
                          v);
```
Set each element of self to the corresponding bit value of `v`. Return a reference to self.

```
RWGBitVec(sz)&           operator=(RWBoolean f);
```
Set all elements of self to the boolean value f.

```
RWGBitVec(sz)&           operator&=(const RWGBitVec(sz)&
                          v);
RWGBitVec(sz)&           operator^=(const RWGBitVec(sz)&
                          v);
RWGBitVec(sz)&           operator|=(const RWGBitVec(sz)&
                          v);
```

Logical assignments. Set each element of self to the logical AND, XOR, or OR, respectively, of self and the corresponding bit in `v`.

Indexing operators

```
RWBitRef                    operator[](size_t i);
```
Returns a reference to the i'th bit of self. This reference can be used as an lvalue. The index `i` must be between 0 and `size–1`, inclusive. Bounds checking will occur.

```
RWBitRef                    operator()(size_t i);
```
Returns a reference to the i'th bit of self. This reference can be used as an lvalue. The index `i` must be between `0` and `size–1`, inclusive. No bounds checking is done.

Public member functions

```
void                        clearBit(size_t i);
```
Clears (*i.e.*, sets to `FALSE`) the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following are equivalent, although `clearBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
  a(i) = FALSE;
  a.clearBit(i);
```

```
const RWByte*               data() const;
```
Returns a const pointer to the raw data of self. Should be used with care.

```
void                        setBit(size_t i);
```
Sets (*i.e.*, sets to `TRUE`) the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following are equivalent, although `setBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`

```
  a(i) = TRUE;
  a.setBit(i);
```

```
RWBoolean                   testBit(size_t i) const;
```
Tests the bit with index `i`. The index `i` must be between 0 and *size*–1. No bounds checking is performed. The following are equivalent, although `testBit(size_t)` is slightly smaller and faster than using `operator()(size_t)`:

```
  if( a(i) )                doSomething();
  if( a.testBit(i) )        doSomething();
```

Related global functions

```
RWGBitVec(sz) operator&(const RWGBitVec(sz)& v1, const
 RWGBitVec(sz)& v2);
RWGBitVec(sz) operator^(const RWGBitVec(sz)& v1, const
 RWGBitVec(sz)& v2);
RWGBitVec(sz) operator|(const RWGBitVec(sz)& v1, const
 RWGBitVec(sz)& v2);
```

Return the logical AND, XOR, and OR, respectively, of vectors `v1` and `v2`.

```
RWBoolean operator==(const RWGBitVec(sz)& v1, const
 RWGBitVec(sz)& v2) const;
```

Returns `TRUE` if each bit of `v1` is set to the same value as the corresponding bit in `v2`. Otherwise, returns `FALSE`.

```
RWBoolean operator!=(const RWGBitVec(sz)& v1, const
 RWGBitVec(sz)& v2)  const;
```

Returns `FALSE` if each bit of `v1` is set to the same value as the corresponding bit in `v2`. Otherwise, returns `TRUE`.

RWGDlist(type)

RWGDlist(type)
|
RWDlist
|
RWSlist

Synopsis

```
#include <rw/gdlist.h>
declare(RWGDlist, type)

RWGDlist(type) a;
```

Description

Class `RWGDlist`(*type*) represents a group of ordered elements of type *type*, not accessible by an external key. Duplicates are allowed. This class is implemented as a doubly-linked list. Objects of type `RWGDlist`(*type*) are declared with macros defined in the standard C++ header file <*generic.h*>.

In order to find a particular item within the collection, a user-provided global "tester" function is required to test for a "match", definable in any consistent way. This function should have prototype:

```
RWBoolean yourTesterFunction(const type* c, const void* d);
```

The argument `c` is a candidate within the collection to be tested for a match. The argument `d` is for your convenience and will be passed to *yourTesterFunction()*. The function should return `TRUE` if a "match" is found between `c` and `d`.

In order to simplify the following documentation, an imaginary typedef

```
typedef RWBoolean (*yourTester)(const type*, const void*);
```

has been used for this tester function.

Example

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

declare(RWGDlist,int)  /* Declare a list of ints */

main()
{
int *ip;
  RWGDlist(int) list;// Define a list of ints
  list.insert(new int(5));// Insert some ints
  list.insert(new int(7));
  list.insert(new int(1));
  list.prepend(new int(11));
  RWGDlistIterator(int) next(list);

 while( ip = next() )
    cout << *ip << endl;// Print out the members
}
```

Program output:

```
11
5
7
1
```

Public constructors

`RWGDlist(`*type*`)();`
Construct an empty collection.

`RWGDlist(`*type*`)(type* a);`
Construct a collection with one entry `a`.

`RWGDlist(`*type*`)(const RWGDlist(`*type*`)& a);`
Copy constructor. A shallow copy of `a` is made.

Assignment operator

`void operator=(const RWGDlist(`*type*`)& a);`
Assignment operator. A shallow copy of `a` is made.

Public member functions

*type** append(*type** a);

Adds an item to the end of the collection. Returns nil if the insertion was unsuccessful.

void apply(void (*ap)(*type**, void*), void*);

Visits all the items in the collection in order, from first to last, calling the user-provided function pointed to by ap for each item. This function should have prototype:

```
void yourApplyFunction(type* c, void*);
```

and can perform any operation on the object at address `c`. The last argument is useful for passing data to the apply function.

*type**& at(size_t i);
const type* at(size_t i) const;

Returns a pointer to the `i`'th item in the collection. The first variant can be used as an lvalue, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

void clear();

Removes all items in the collection.

RWBoolean contains(*yourTester* t, const void* d) const;

Returns `TRUE` if the collection contains an item for which the user-defined function pointed to by `t` finds a match with `d`.

RWBoolean containsReference(const type* e) const;

Returns `TRUE` if the collection contains an item with the addresse `e`.

size_t entries() const;

Returns the number of items in the collection.

*type** find(*yourTester* t, const void* d) const;

Returns the first item in the collection for which the user-provided function pointed to by `t` finds a match with `d`, or nil if no item is found.

*type** findReference(const *type** e) const;

Returns the first item in the collection with the address `e`, or nil if no item is found.

*type**   `first() const;`
Returns the first item of the collection.

*type**   `get();`
Returns and *removes* the first item of the collection.

*type**   `insert(`*type** `e);`
Adds an item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

`void`   `insertAt(size_t indx, type* e);`
Adds a new item to the collection at position `indx`. The item previously at position `i` is moved to `i+1`, *etc*. The index `indx` must be between 0 and the number of items in the collection, or an exception of type `TOOL_INDEX` will be thrown.

`RWBoolean`   `isEmpty() const;`
Returns `TRUE` if the collection is empty, otherwise `FALSE`.

*type**   `last() const;`
Returns the last item of the collection.

`size_t`   `occurrencesOf(`*yourTester* `t, const void* d) const;`
Returns the number of occurrences in the collection for which the user-provided function pointed to by `t` finds a match with `d`.

`size_t`   `occurrencesOfReference(const` *type** `e) const;`
Returns the number of items in the collection with the address `e`.

*type**   `prepend(`*type** `a);`
Adds an item to the beginning of the collection. Returns nil if the insertion was unsuccessful.

*type**   `remove(`*yourTester* `t, const void* d);`
Removes and returns the first item from the collection for which the user-provided function pointed to by `t` finds a match with `d`, or returns nil if no item is found.

*type**   removeReference(const *type** e);
Removes and returns the first item from the collection with the address `e`, or returns nil if no item is found.

RWGDlistIterator(type)

RWGDlistIterator(*type*)
|
RWDlistIterator
|
RWSlistIterator

Synopsis

```
#include <rw/gdlist.h>
declare(RWGDlist, type)

RWGDlist(type) a;
RWGDlistIterator(type) I(a);
```

Description

Iterator for class `RWGDlist`(*type*), which allows sequential access to all the elements of a doubly-linked list. Elements are accessed in order, in either direction.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

In order to simplify the following documentation, an imaginary typedef

```
  typedef RWBoolean (*yourTester)(const type*, const void*);
```

has been used. See the documentation for class `RWGDlist`(*type*) for an explanation of this function.

Example

See class `RWGDlist`(*type*)

Public constructor

`RWGDlistIterator(`*type*`)( RWGDlist(`*type*`)& list);`
Construct an iterator for the `RWGDlist`(*type*) `list`. Immediately after construction, the position of the iterator is undefined.

Public member operators

*type** `operator()();`

Advances the iterator to the next item and returns it. Returns nil if at the end of the collection.

`void operator++();`

Advances the iterator one item.

`void operator--();`

Moves the iterator back one item.

`void operator+=(size_t n);`

Advances the iterator `n` items.

`void operator-=(size_t n);`

Moves the iterator back `n` items.

Public member functions

`RWBoolean atFirst() const;`

Returns `TRUE` if the iterator is at the start of the list, `FALSE` otherwise;

`RWBoolean atLast() const;`

Returns `TRUE` if the iterator is at the end of the list, `FALSE` otherwise;

*type** `findNext(yourTester t,const` *type** `d);`

Moves the iterator to the next item for which the function pointed to by `t` finds a match with `d` and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

*type** `findNextReference(const` *type** `e);`

Moves the iterator to the next item with the address `e` and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

*type** `insertAfterPoint(`*type** `a);`

Adds item `a` after the current iterator position and return the item. The position of the iterator is left unchanged.

*type** `key() const;`

Returns the item at the current iterator position.

*type** `remove();`
Removes and returns the item at the current cursor position. Afterwards, the iterator will be positioned at the previous item in the list.

*type** `removeNext(`*yourTester* `t, const` *type** `d);`
Moves the iterator to the next item for which the function pointed to by `t` finds a "match" with `d` and removes and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

*type** `removeNextReference(const` *type** `a);`
Moves the iterator to the next item with the address `e` and removes and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

`void` `reset();`
Resets the iterator to its initial state.

`void` `toFirst();`
Moves the iterator to the first item in the list.

`void` `toLast();`
Moves the iterator to the last item in the list.

RWGOrderedVector(val)

Synopsis

```
#include <rw/gordvec.h>
declare(RWGVector,val)
declare(RWGOrderedVector,val)
implement(RWGVector,val)
implement(RWGOrderedVector,val)
```

```
RWGOrderedVector(val) v;  // Ordered vector of objects of
 type val.
```

Description

Class `RWGOrderedVector`(*val*) represents an ordered collection of objects of type *val*. Objects are ordered by the order of insertion and are accessible by index. Duplicates are allowed. `RWGOrderedVector`(*val*) is implemented as a vector, using macros defined in the standard C++ header file <*generic.h*>.

Note – It is a *value-based* collection: items are copied in and out of the collection.

The class *val* must have:

- a default constructor;
- well-defined copy semantics (*val*`::`*val*`(const` *val*`&)` or equiv.);
- well-defined assignment semantics (*val*`::operator=(const` *val*`&)` or equiv.);
- well-defined equality semantics (*val*`::operator==(const` *val*`&)` or equiv.).

To use this class you must declare and implement its base class as well as the class itself. For example, here is how you declare and implement an ordered collection of doubles:

```
declare(RWGVector,double)        // Declare base class
declare(RWGOrderedVector,double) // Declare ordered vector
```

// In one and only one `.cc` file you must put the following:

```
implement(RWGVector,double)         // Implement base class
implement(RWGOrderedVector,double)   // Implement ordered
                                        vector
```

For each type of `RWGOrderedVector` you must include one (and only one) call to the macro `implement` somewhere in your code for both the `RWGOrderedVector` itself and for its base class `RWGVector`.

Example

Here's an example that uses an ordered vector of `RWCStrings`.

```
#include <rw/gordvec.h>
#include <rw/cstring.h>
#include <rw/rstream.h>
declare(RWGVector,RWCString)
declare(RWGOrderedVector,RWCString)
implement(RWGVector,RWCString)
implement(RWGOrderedVector,RWCString)
main()
{
  RWGOrderedVector(RWCString) vec;
  RWCString one("First");
  vec.insert(one);
  vec.insert("Second");// Automatic type conversion occurs
  vec.insert("Last");// Automatic type conversion occurs
  for(size_t i=0; i <vec.entries(); i++) cout << vec[i] << endl;
  return 0;
}
```

Program output:

```
First
Second
Last
```

Public constructors

`RWGOrderedVector(`*val*`)(size_t capac=RWDEFAULT_CAPACITY);`
Construct an ordered vector of elements of type *val*. The initial capacity of the vector will be `capac` whose default value is `RWDEFAULT_CAPACITY`. The capacity will be automatically increased as necessary should too many items be inserted, a relatively expensive process because each item must be copied into the new storage.

Public member functions

```
val                           operator()(size_t i) const;
val&                          operator()(size_t i);
```

Return the i'th value in the vector. The index `i` must be between 0 and the length of the vector less one. No bounds checking is performed. The second variant can be used as an lvalue, the first cannot.

```
val                           operator[](size_t i) const;
val&                          operator[](size_t i);
```

Return the i'th value in the vector. The index `i` must be between 0 and the length of the vector less one. Bounds checking will be performed. The second variant can be used as an lvalue, the first cannot.

```
void                          clear();
```

Remove all items from the collection.

```
size_t                        entries() const;
```

Return the number of items currently in the collection.

`size_t` `index(`*val item*`) const;`

Performs a linear search of the collection returning the index of the first item that *isEqual* to the argument `item`. If no item is found, then it returns `RW_NPOS`.

`void` `insert(`*val item*`);`

Add the new value item to the end of the collection.

`void` `insertAt(size_t indx,` *val item*`);`

Add the new value `item` to the collection at position `indx`. The value of `indx` must be between zero and the length of the collection. No bounds checking is performed. Old items from index `indx` upwards will be shifted to higher indices.

```
RWBoolean                     isEmpty() const;
```

Return `TRUE` if the collection has no entries. `FALSE` otherwise.

```
void                          removeAt(size_t indx);
```

Remove the item at position `indx` from the collection. The value of `indx` must be between zero and the length of the collection less one. No bounds checking is performed. Old items from index `indx+1` will be shifted to lower indices. *E.g.*, the item at index `indx+1` will be moved to position `indx`, *etc.*

```
void                         resize(size_t newCapacity);
```

Change the capacity of the collection to `newCapacity`, which must be at least as large as the present number of items in the collection.

Note – The actual number of items in the collection does not change, just the capacity.

RWGQueue(type)

RWGQueue(*type*)
|
RWSlist

Synopsis

```
#include <rw/gqueue.h>
declare(RWGQueue, type)

RWGQueue(type) a;
```

Description

Class `RWGQueue`(*type*) represents a group of ordered elements, not accessible by an external key. A `RWGQueue`(*type*) is a first in, first out (FIFO) sequential list for which insertions are made at one end (the "tail"), but all removals are made at the other (the "head"). Hence, the ordering is determined externally by the ordering of the insertions. Duplicates are allowed. This class is implemented as a singly-linked list. Objects of type `RWGQueue`(*type*) are declared with macros defined in the standard C++ header file <*generic.h*>.

In order to find a particular item within the collection, a user-provided global "tester" function is required to test for a match", definable in any consistent way. This function should have prototype:

`RWBoolean` *yourTesterFunction*`(const` *type*`* c, const void* d);`
The argument `c` is a candidate within the collection to be tested for a match. The argument `d` is for your convenience and will be passed to *yourTesterFunction*`()`. The function should return `TRUE` if a "match" is found between `c` and `d`.

In order to simplify the documentation below, an imaginary typedef
`typedef RWBoolean (*`*yourTester*`)(const` *type*`*, const void*);`
has been used for this tester function.

Public constructors

`RWGQueue(`*type*`)();`
Construct an empty queue.

`RWGQueue(`*type*`)(`*type*`* a);`
Construct a queue with one entry `a`.

`RWGQueue(`*type*`)(const RWGQueue(`*type*`)& q);`
Copy constructor. A shallow copy of `q` is made.

Assignment operator

`void` `operator=(const RWGQueue(`*type*`)& q);`
Assignment operator. A shallow copy of `q` is made.

Public member functions

*type** `append(`*type*`* a);`
Adds `a` to the end of the queue and returns it. Returns nil if the insertion was unsuccessful.

`void` `clear();`
Removes all items from the queue.

`RWBoolean` `contains(`*yourTester* `t, const void* d) const;`
Returns `TRUE` if the queue contains an item for which the user-defined function pointed to by `t` finds a match with `d`.

`RWBoolean` `containsReference(const` *type*`* e)const;`
Returns `TRUE` if the queue contains an item with the address `e`.

`size_t` `entries() const;`
Returns the number of items in the queue.

*type** `first() const;`
Returns the first item in the queue, or nil if the queue is empty.

*type** `get();`
Returns and removes the first item in the queue. Returns nil if the queue is empty.

`RWBoolean` `isEmpty() const;`
Returns `TRUE` if the queue is empty, otherwise `FALSE`.

*type** `insert(`*type*`* a);`
Calls `append(`*type*`*)` with `a` as the argument.

*type** `last();`
Returns the last (most recently inserted) item in the queue, or nil if the queue is empty.

```
size_t                    occurrencesOf(yourTester t,
                           const void* d) const;
```

Returns the number of items in the queue for which the user-provided function pointed to by `t` finds a match with `d`.

```
size_t               occurrencesOfReference(const type* e)
                     const;
```

Returns the number of items in the queue with the address `e`.

RWGSlist(type)

RWGSlist(*type*)
    |
RWSlist

Synopsis

```
#include <rw/gslist.h>
declare(RWGSlist, type)
```

`RWGSlist(`*type*`) a;`

Description

Class `RWGSlist`(*type*) represents a group of ordered elements of type ***type***, not accessible by an external key. Duplicates are allowed. This class is implemented as a singly-linked list. Objects of type `RWGSlist`(*type*) are declared with macros defined in the standard C++ header file <*generic.h*>.

In order to find a particular item within the collection, a user-provided global "tester" function is required to test for a "match", definable in any consistent way. This function should have prototype:

```
RWBoolean yourTesterFunction(const type* c, const void* d);
```

The argument `c` is a candidate within the collection to be tested for a match. The argument `d` is for your convenience and will be passed to *yourTesterFunction()*. The function should return `TRUE` if a "match" is found between `c` and `d`.

In order to simplify the documentation below, an imaginary typedef

```
typedef RWBoolean (*yourTester)(const type*, const void*);
```

has been used for this tester function.

Public constructors

`RWGSlist(`*type*`)();`
Construct an empty collection.

`RWGSlist(`*type*`)(`*type*`* a);`
Construct a collection with one entry `a`.

`RWGSlist(`*type*`)(const RWGSlist(`*type*`)& a);`
Copy constructor. A shallow copy of `a` is made.

Assignment operator

`void                    operator=(const RWGSlist(`*type*`)&);`
Assignment operator. A shallow copy of `a` is made.

Public member functions

```
type*                       append(type* a);
```
Adds an item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

```
void                        apply(void (*ap)(type*, void*),
                             void* );
```
Visits all the items in the collection in order, from first to last, calling the user-provided function pointed to by `ap` for each item. This function should have prototype:

```
void yourApplyFunction(type* c, void*);
```

and can perform any operation on the object at address `c`. The last argument is useful for passing data to the apply function.

```
type*&                      at(size_t i);
const type*                 at(size_t i) const;
```
Returns a pointer to the `i`'th item in the collection. The first variant can be used as an lvalue, the second cannot. The index `i` must be between zero and the number of items in the collection less one, or an exception of type `TOOL_INDEX` will be thrown.

```
void                        clear();
```
Removes all items in the collection.

```
RWBoolean                   contains(yourTester t, const
                             void* d) const;
```
Returns `TRUE` if the collection contains an item for which the user-defined function pointed to by `t` finds a match with `d`.

```
RWBoolean                   containsReference(const type* e)
                             const;
```
Returns `TRUE` if the collection contains an item with the address `e`.

```
size_t                      entries() const;
```
Returns the number of items in the collection.

```
type*                       find(yourTester t, const void* d)
                             const;
```
Returns the first item in the collection for which the user-provided function pointed to by `t` finds a match with `d`, or nil if no item is found.

*type**　　　　`findReference(const` *type** `e) const;`

Returns the first item in the collection with the address `e`, or nil if no item is found.

*type**　　　　`first() const;`

Returns the first item of the collection.

*type**　　　　`get();`

Returns and *removes* the first item of the collection.

*type**　　　　`insert(`*type** `e);`

Adds an item to the end of the collection and returns it. Returns nil if the insertion was unsuccessful.

`void`　　　　`insertAt(size_t indx, type* e);`

Adds a new item to the collection at position `indx`. The item previously at position `i` is moved to `i+1`, *etc.* The index `indx` must be between 0 and the number of items in the collection, or an exception of type `TOOL_INDEX` will be thrown.

`RWBoolean`　　　　`isEmpty() const;`

Returns `TRUE` if the collection is empty, otherwise `FALSE`.

*type**　　　　`last() const;`

Returns the last item of the collection.

`size_t`　　　　`occurrencesOf(`*yourTester* `t, const void* d) const;`

Returns the number of occurrences in the collection for which the user-provided function pointed to by `t` finds a match with `d`.

`size_t`　　　　`occurrencesOfReference(const` *type** `e) const;`

Returns the number of items in the collection with the address `e`.

*type**　　　　`prepend(const` *type** `a);`

Adds an item to the beginning of the collection and returns it. Returns nil if the insertion was unsuccessful.

*type** `remove(`*yourTester* `t, const void* d);`

Removes and returns the first item from the collection for which the user-provided function pointed to by `t` finds a match with `d`, or returns nil if no item is found.

*type** `removeReference(const` *type** `e);`

Removes and returns the first item from the collection with the address `e`, or returns nil if no item is found.

RWGSlistIterator(type)

RWGSlistIterator(*type*)
|
RWSlistIterator

Synopsis

```
#include <rw/gslist.h>
declare(RWGSlist, type)

RWGSlist(type) a;
RWGSlistIterator(type) I(a);
```

Description

Iterator for class `RWGSlist`(*type*), which allows sequential access to all the elements of a singly-linked list. Elements are accessed in order, first to last.

Like all Tools.h++ iterators, the "current item" is undefined immediately after construction—you must define it by using `operator()` or some other (valid) operation.

Once the iterator has advanced beyond the end of the collection it is no longer valid—continuing to use it will bring undefined results.

In order to simplify the documentation below, an imaginary typedef

```
typedef RWBoolean (*yourTester)(const type*, const void*);
```

has been used. See the documentation for class `RWGSlist`(*type*) for an explanation of this function.

Public constructor

`RWGSlistIterator(`*type*`)( RWGSlist(`*type*`)& list);`
Construct an iterator for the `RWGSlist`(*type*) list. Immediately after construction, the position of the iterator is undefined.

Public member operators

*type**   `operator()();`
Advances the iterator to the next item and returns it. Returns nil if at the end of the collection.

`void                    operator++();`
Advances the iterator one item.

`void                    operator+=(size_t n);`
Advances the iterator `n` items.

Public member functions

`RWBoolean` `atFirst() const;`
Returns TRUE if the iterator is at the start of the list, FALSE otherwise;

`RWBoolean` `atLast() const;`
Returns TRUE if the iterator is at the end of the list, FALSE otherwise;

*type** `findNext(`*yourTester* `t,const` *type** `d);`
Moves the iterator to the next item for which the function pointed to by `t` finds a match with `d` and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

*type** `findNextReference(const` *type** `e);`
Moves the iterator to the next item with the address `e` and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

*type** `insertAfterPoint(`*type** `a);`
Adds item `a` after the current iterator position and return the item. The position of the iterator is left unchanged.

*type** `key() const;`
Returns the item at the current iterator position.

*type** `remove();`
Removes and returns the item at the current cursor position. Afterwards, the iterator will be positioned at the previous item in the list. In a singly-linked list, this function is an inefficient operation because the entire list must be traversed, looking for the link before the link to be removed.

*type** `removeNext(`*yourTester* `t, const` *type** `d);`
Moves the iterator to the next item for which the function pointed to by `t` finds a "match" with `d` and removes and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

*type** `removeNextReference(const` *type** `e);`
Moves the iterator to the next item with the address `e` and removes and returns it. Returns nil if no match is found, in which case the position of the iterator will be undefined.

```
void                        reset();
```
Resets the iterator to its initial state.

```
void                        toFirst();
```
Moves the iterator to the start of the list.

```
void                        toLast();
```
Moves the iterator to the end of the list.

RWGSortedVector(val)

RWGSortedVector(*val*)
|
RWGVector(*val*)

Synopsis

```
#include <rw/gsortvec.h>
declare(RWGSortedVector,val)
implement(RWGSortedVector, val)

RWGSortedVector(val) v;  // A sorted vector of val's.
```

Description

Class `RWGSortedVector`(*val*) represents a vector of elements of type *val*, sorted using an insertion sort. The elements can be retrieved using an index or a search. Duplicates are allowed. Objects of type `RWGSortedVector`(*val*) are declared with macros defined in the standard C++ header file <*generic.h*>.

Note – It is a value-based collection: items are copied in and out of the collection.

The class *val* must have:

- a default constructor;
- well-defined copy semantics (*val*`::`*val*`(const` *val*`&)` or equiv.);
- well-defined assignment semantics (*val*`::operator=(const` *val*`&)` or equiv.);
- well-defined equality semantics (*val*`::operator==(const` *val*`&)` or equiv.);
- well-defined less-than semantics (*val*`::operator<(const` *val*`&)` or equiv.).

To use this class you must declare and implement its base class as well as the class itself. For example, here is how you declare and implement a sorted collection of doubles:

```
declare(RWGVector,double)     // Declare base class
declare(RWGSortedVector,double)// Declare sorted vector
```

// In one and only one `.cc` file you must put the following:

```
implement(RWGVector,double)          // Implement base
implement(RWGSortedVector,double)    // Implement sorted
                                        vector
```

For each type of `RWGSortedVector` you must include one (and only one) call to the macro `implement` somewhere in your code for both the `RWGSortedVector` itself and for its base class `RWGVector`.

Insertions and retrievals are done using a binary search.

Note – The constructor of a RWGSortedVector(*val*) requires a pointer to a "comparison function."

This function should have prototype:
`int comparisonFunction(const` *val*`* a, const` *val*`* b);`
and should return an int less than, greater than, or equal to zero, depending on whether the item pointed to by `a` is less than, greater than, or equal to the item pointed to by `b`. Candidates from the collection will appear as `a`, the key as `b`.

Example

Here's an example of a sorted vector of ints:

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

declare(RWGVector,int)
declare(RWGSortedVector,int)
implement(RWGVector,int)
implement(RWGSortedVector,int)

// Declare and define the "comparison function":int compFun(const
int* a, const int* b)
{
    return *a - *b;
}

main()
{
    // Declare and define an instance,
    // using the comparison function 'compFun':
RWGSortedVector(int) avec(compFun);

    // Do some insertions:
    avec.insert(3);// 3
    avec.insert(17);// 3 17
    avec.insert(5);// 3 5 17

    cout < avec(1);// Prints '5'
    cout < avec.index(17);// Prints '2'
}
```

Public constructors

`RWGSortedVector(`*val*`)( int (*f)(const `*val*`*, const `*val*`) );`
Construct a sorted vector of elements of type *val*, using the comparison function pointed to by `f`. The initial capacity of the vector will be set by the value `RWDEFAULT_CAPACITY`. The capacity will automatically be increased should too many items be inserted.

`RWGSortedVector(`*val*`)(int (*f)(const `*val*`*, const `*val*`),`
 `size_t N);`
Construct a sorted vector of elements of type *val*, using the comparison function pointed to by `f`. The initial capacity of the vector will be `N`. The capacity will automatically be increased should too many items be inserted.

Public member functions

val   `operator()(size_t i) const;`
Return the `i`'th value in the vector. The index `i` must be between 0 and the length of the vector less one. No bounds checking is performed.

val   `operator[](size_t i) const;`
Return the `i`'th value in the vector. The index `i` must be between 0 and the length of the vector less one. Bounds checking is performed.

`size t`   `entries() const;`
Returns the number of items currently in the collection.

`size_t`   `index(`*val* `v);`
Return the index of the item with value `v`. The value "`RW_NPOS`" is returned if the value does not occur in the vector. A binary search, using the comparison function, is done to find the value. If duplicates are present, the index of the first instance is returned.

`RWBoolean`   `insert(`*val* `v);`
Insert the new value `v` into the vector. A binary search, using the comparison function, is performed to determine where to insert the value. The item will be inserted after any duplicates. If the insertion causes the vector to exceed its capacity, it will automatically be resized by an amount given by `RWDEFAULT_RESIZE`.

`void`   `resize(size_t newCapacity);`
Change the capacity of the collection to `newCapacity`, which must be at least as large as the present number of items in the collection.

Note – The actual number of items in the collection does not change, just the capacity.

RWGStack(type)

RWGStack(*type*)
|
RWSlist

Synopsis

```
#include <rw/gstack.h>
declare(RWGStack,type)

RWGStack(type) a;
```

Description

Class `RWGStack`(*type*) represents a group of ordered elements, not accessible by an external key. A `RWGStack`(*type*) is a last in, first out (LIFO) sequential list for which insertions and removals are made at the beginning of the list. Hence, the ordering is determined externally by the ordering of the insertions. Duplicates are allowed. This class is implemented as a singly-linked list. Objects of type `RWGStack`(*type*) are declared with macros defined in the standard C++ header file `<generic.h>`.

In order to find a particular item within the collection, a user-provided global "tester" function is required to test for a "match," definable in any consistent way. This function should have prototype:

`RWBoolean` *yourTesterFunction*`(const` *type*`* c, const void* d);`
The argument `c` is a candidate within the collection to be tested for a match. The argument `d` is for your convenience and will be passed to *yourTesterFunction()*. The function should return `TRUE` if a "match" is found between `c` and `d`.

In order to simplify the documentation below, an imaginary typedef

`typedef RWBoolean (*`*yourTester*`)(const` *type*`*, const void*);`
has been used for this tester function.

Public constructors

`RWGStack(`*type*`)();`
Construct an empty stack.

`RWGStack(`*type*`)(`*type*`* a);`
Construct a stack with one entry `a`.

`RWGStack(`*type*`)(const RWGStack(`*type*`)& a);`
Copy constructor. A shallow copy of `a` is made.

Assignment operator

`void` `operator=(const RWGStack (`*type*`)& a);`

Assignment operator. A shallow copy of `a` is made.

Public member functions

`void` `clear();`

Removes all items from the stack.

`RWBoolean` `contains(`*yourTester* `t, const void* d) const;`

Returns `TRUE` if the stack contains an item for which the user-defined function pointed to by `t` finds a match with `d`.

`RWBoolean` `containsReference(const` *type*`* e) const;`

Returns `TRUE` if the stack contains an item with the address `e`

`size_t` `entries() const;`

Returns the number of items in the stack.

`RWBoolean` `isEmpty() const;`

Returns `TRUE` if the stack is empty, otherwise `FALSE`.

`size_t` `occurrencesOf(`*yourTester* `t, const void* d) const;`

Returns the number of items in the stack for which the user-provided function pointed to by `t` finds a match with `d`.

`size_t` `occurrencesOfReference (const type* e) const;`

Returns the number of items in the stack with the address `e`.

*type** `pop();`

Removes and returns the item at the top of the stack, or returns nil if the stack is empty.

`void` `push(`*type*`* a);`

Adds an item to the top of the stack.

*type** `top() const;`

Returns the item at the top of the stack or nil if the stack is empty.

RWGVector(val)

Synopsis

```
#include <rw/gvector.h>
declare(RWGVector,val)
implement(RWGVector,val)

RWGVector (val) a;          // A Vector of val's.
```

Description

Class `RWGVector`(*val*) represents a group of ordered elements, accessible by an index. Duplicates are allowed. This class is implemented as an array. Objects of type `RWGVector`(*val*) are declared with macros defined in the standard C++ header file <*generic.h*>.

Note – It is a value-based collection: items are copied in and out of the collection.

The class *val* must have:

- a default constructor;
- well-defined copy semantics (*val*`::`*val*`(const` *val*&) or equiv.);
- well-defined assignment semantics (*val*`::operator=(const` *val*&) or equiv.).

For each type of `RWGVector`, you must include one (and only one) call to the macro `implement`, somewhere in your code.

Example

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

declare(RWGVector, RWDate)/* Declare a vector of dates */
implement(RWGVector, RWDate)/* Implement a vector of dates */

main()
{
   RWGVector(RWDate) oneWeek(7);
   for (int i=1; i<7; i++)
     oneWeek(i) = oneWeek(0) + i;

   for (i=0; i<7; i++)
     cout << oneWeek(i) << endl;

  return 0;
}
```

Program output:

```
     04/12/93
     04/13/93
     04/14/93
     04/15/93
     04/16/93
     04/17/93
     04/18/93
```

Public constructors

`RWGVector(`*val*`)();`
Construct an empty vector.

`RWGVector(`*val*`)(size_t n);`
Construct a vector with length `n`. The initial values of the elements can (and probably will) be garbage.

`RWGVector(`*val*`)(size_t n,` *val* `v);`
Construct a vector with length `n`. Each element is assigned the value `v`.

`RWGVector(`*val*`)(RWGVector(`*val*`)& s);`
Copy constructor. The entire vector is copied, including all imbedded values.

Public member operators

`RWGVector(`*val*`)                operator=(RWGVector(`*val*`)& s);`
Assignment operator. The entire vector is copied.

`RWGVector(`*val*`)&               operator=(`*val* `v);`
Sets all elements of self to the value `v`.

val`&                             operator()(size_t i);`
Returns a reference to the the `i`'th element of self. The index `i` must be between zero and the length of the vector less one. No bounds checking is performed.

val`&                            operator[](size_t i);`
Returns a reference to the the `i`'th element of self. The index `i` must be between zero and the length of the vector less one. Bounds checking is performed.

Public member functions

`size_t                      length() const;`
Returns the length of the vector.

`void                        reshape(size_t n);`
Resize the vector. If the vector shrinks, it will be truncated. If the vector grows, then the value of the additional elements will be undefined.

Summary of typedefs and macros A

Constants:

```
#define           FALSE              0
#define           rwnil              0
#define           TRUE               1
const int)        RWkeyLen= 16
const RWoffset    RWNIL = -1L;
```

Typedefs:

```
typedef           unsigned short     ClassID;// Unique for each class
typedef           unsigned long      clockTy;// seconds
typedef           unsigned           dayTy;// days
typedef           int                fileDescTy;// File descriptor
typedef           unsigned           hourTy;// hours
typedef           unsigned long      julTy;// Julian days
typedef           unsigned           minuteTy;// minutes
typedef           unsigned           monthTy;// months
typedef           unsigned           secondTy;// seconds
typedef           unsigned           yearTy;// years
typedef           int                RWBoolean;// TRUE or FALSE
typedef           long               RWoffset;// Used for file offsets
typedef           void*              RWvoid;// For arrays of void*'s
typedef           unsigned           RWspace;// Used for file records
typedef           long               RWstoredValue;// Used for file offsets
```

☰ A

Pointers to Functions:

```
typedef          void            (*RWapplyCollectable)(RWCollectable*, void*);
typedef          void            (*RWapplyGeneric)(void*, void*);
typedef          void            (*RWapplyKeyAndValue)
                                     (RWCollectable*,RWCollectable*,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          RWCollectable*  (*RWuserCreator)();
```

Enumerations:

```
enum             RWSeverity      {RWWARNING, RWDEFAULT, RWFATAL}
```

Standard Smalltalk Interface:

```
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          RWIdentityHash             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;
```

Header file compiler.h B

This header file is the repository for all information about compiler related capabilities and limitations. It is included by Tools.h++ libraries.

The general philosophy of the various macros defined in `<rw/compiler.h>` is that an ARM compliant compiler does not have any of these macros defined. That is, macros characterize a compiler's limitations, rather than its abilities.

Table B-1 Summary of preprocessor macros found in `<rw/compiler.h>`.

Macro	**Defined if ...**
`RW_BROKEN_TEMPLATES`	Supports only cfront 3.0 style templates
`RW_CRLF_CONVENTION`	Newlines marked by `<CR><LF>`
`RW_GLOBAL_ENUMS`	Nested enums have global scope
`RW_INLINE86_ASSEMBLY`	Compiler supports inline 80x86 assembly
`RW_IOS_XALLOC_BROKEN`	`ios::xalloc()` does not zero initialize.
`RW_KR_ONLY`	No function prototypes
`RW_NO_ACCESS_ADJUSTMENT`	Does not allow adjustment of visibility of base class
`RW_NO_ANSI_SPRINTF`	`sprintf()` does not return string length
`RW_NO_CLOCK`	Does not have `clock()`
`RW_NO_CONST_OVERLOAD`	Cannot overload on const
`RW_NO_CPP_RECURSION`	Preprocessor does not detect recursions properly

Table B-1 Summary of preprocessor macros found in <rw/compiler.h>. *(Continued)*

Macro	**Defined if ...**
RW_NO_EXCEPTIONS	Does not support exceptions
RW_NO_GETTIMEOFDAY	Does not have gettimeofday()
RW_NO_GLOBAL_TZ	Does not have global variables timezone and daylight
RW_NO_LOCALE	No ANSI locale facilities
RW_NO_MEMMOVE	Does not have memmove()
RW_NO_NESTED_QUOTES	cpp does not escape quotes inside strings
RW_NO_OVERLOAD_UCHAR	Cannot overload on unsigned char
RW_NO_OVERLOAD_WCHAR	Type wchar_t is not a distinct type
RW_NO_POSTFIX	Does not support (*this)++
RW_NO_SCHAR	Does not support signed char
RW_NO_STRFTIME_CAPC	Does not have %C directive to strftime()
RW_NO_STRICMP	Does not have stricmp()
RW_NO_STRNICMP	Does not have strnicmp()
RW_NO_STRSTR	Does not have strstr()
RW_NO_TEMPLATES	Does not support templates at all
RW_NO_WSTR	Does not support wide character strings
RW_NON_ANSI_HEADERS	Function memcpy() found in <memory.h>

Messages C

Table C-1 Core messages. These are messages used by all Tools.h++ libraries. The symbols are defined in `<rw/coreerr.h>`. These messages belong to category "`rwcore6.0`".

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 "`rwtool6.0`".

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"`

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 "`rwtool6.0`". *(Continued)*

Symbol	**Message**
`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 D

Ammeraal, L. *Programs and Data Structures in C*, John Wiley and Sons, 1989, ISBN 0-471-91751-6.

Booch, Grady, *Object-Oriented Design with Applications*, The Benjamin Cummings Publishing Company, Inc., 1991, ISBN 0-8053-0091-0–.

Budd, Timothy, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991, ISBN 0-201-54709-0.

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.

Ellis, Margaret A. and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990, ISBN 0-201-51459-1–.

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.

Lippman, Stanley B. *C++ Primer,* Addison-Wesley, 1989, ISBN 0-201-16487-6–.

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.

Petzold, Charles, *Programming Windows,* Microsoft Press, 1990, ISBN 1-55615-264-7.

Sedgewick, Robert. *Algorithms*, Addison-Wesley, 1988, ISBN 0-201-06673-4.

Stroustrup, Bjarne. *The C++ Programming Language*, Addison-Wesley, 1986, ISBN 0-201-12078-X.

Stroustrup, Bjarne. *The C++ Programming Language,* Second Edition, Addison-Wesley, 1991, ISBN 0-201-53992-6–. – **Highly recommended** –

Index

Adobe PostScript