Tools.h++
Class Reference

Rogue Software
Corvallis, Oregon USA

Authors: *Tools.h++* Team

Tools.h++ Team:

Tools.h++ Development: Anna Dahan, Frank Griswold, Kevin Johnsrude, Tom Pearson, and Jim Shur

Engineering Services: Wade Brittain, Bruce Kyle, Randall Robinson, Howard Sanders, and Tibbi Scott

Manuals: Elaine Cull, Wendi Minne, and Julie Prince

Marketing: Anita Covelli and Michael Nelson

Support: North Krimsly

With invaluable help from: James Fowler, John Vriezen, and the entire Rogue Wave Crew

Printed in the United States of America.

Part # RW30-01-2-032596b

Class Reference Printing History:

March 1996	First Printing

Rogue Wave Software, Inc., 850 SW 35th Street, Corvallis, Oregon, 97333 USA

Product Information: **(541) 754-5010**
1-800-487-3217

Technical support: **(541) 754-2311**
e-mail: support@roguewave.com

FAX: **(541) 757-6650**

BBS: **(541) 754-5011**

World Wide Web: **http://www.roguewave.com**

Please have your product serial number available when calling for technical support.

Table of Contents

Introduction

The *Tools.h++ Class Reference* describes all the classes and functions in *Tools.h++*. It does *not* provide a tutorial on how to program with the *Tools.h++* class library. For information on how to write programs using *Tools.h++*, consult the *Tools.h++ User's Guide*. For information on installing and using *Tools.h++*, review the *Tools.h++ Getting Started Guide*.

Organization of the Class Reference

Immediately following this introduction is a class hierarchy diagram. The class hierarchy lists all the classes, and illustrates the relationships among them. You can refer to it for a bird's-eye view of the inheritance structure used in *Tools.h++*.

The remainder of this reference is an alphabetical listing of classes. The entry for each class begins with an illustration showing the individual class's inheritance hierarchy, followed by a synopsis that lists the header files(s) and the 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. Following the synopsis is a brief description of the class, and a list of member and global functions. These functions are organized in categories according to their general use – for example, "constructors," "global operators," and "public member functions." The categories, although somewhat arbitrary, provide a way of organizing the many functions.

Conventions

All Rogue Wave class names start with the letters *RW*, as in *RW***Collectable**, with the bold font emphasizing the class name rather than the prefix. In some cases, we may refer to an instance of a class by an English name; for example, "the string" instead of "the *RW***CString** instance." We do this to make it easier to read when the meaning should be clear from context, but we use the longer form if there is a possible ambiguity.

All function names begin with a lower case letter, with the first letter of subsequent words capitalized. Function names attempt to accurately describe what a function does. For example, `RWCString::toLower()` changes all uppercase letters in itself to lowercase. Underline characters and abbreviations are not generally used in function names.

Function names, examples, operating system commands, mathematical symbols and code fragments are shown in a courier font, as in `<rw/stream.h>`. Vertical ellipses are used in code examples to indicate that some part of the code is missing.

Throughout this documentation, there are frequent references to "self." This should be read as "`*this`".

Inheritance Notation

Each class that inherits from another class (or other classes) includes an illustration that shows the inheritance hierarchy. For example, the following illustration indicates that class ***A*** inherits from class ***B***:

$$A \longrightarrow B$$

When a class inherits from more than one class, or there are multiple levels of inheritance, all of the inheritance relationships are shown. For example, the following illustration indicates that ***A*** inherits from class ***B*** and from class ***C***, which inherits from class ***D***.

$$\begin{array}{lll} & \longrightarrow B & \\ A & & \\ & \longrightarrow C & \longrightarrow D \end{array}$$

The notation system used in the inheritance hierarchies is based on the Object Modeling Technique (OMT) developed by Rumbaugh and others.[1]

Member Functions

Within their general categories, 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 *RW**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 *RW**BinaryTree***, which is inherited from class *RW**Collection***. When a member function is inherited without being redefined, the member function appears in both places, and this guide refers you to the original definition.

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 *RW**BinaryTree***.

---

[1] The notation is similar to the notation used in *Design Patterns* by Gamma, Helm, Johnson, and Vlissides.

Class Hierarchy

The following list shows the public class hierarchy of the *Tools.h++* classes. Note that this is the *public* class hierarchy--the implementation of a given class may use private inheritance. Additionally, some classes inherit from public, but undocumented, implementation classes. Undocumented classes are omitted from the hierarchy.

Classes derived by multiple inheritance show their additional base(s) in italics to the right of the class name.

Class Hierarchy

- *RW**Bench***
- *RW**BitVec***
- *RW**BTreeOnDisk***
- *RW**CacheManager***
- *RW**Collectable***
    - *RW**Collection***
        - *RW**Bag***
        - *RW**BinaryTree***
        - *RW**BTree***
            - *RW**BTreeDictionary***
        - *RW**HashTable***
            - *RW**Set***
                - *RW**Factory***
                - *RW**HashDictionary***
                    - *RW**IdentityDictionary***
                - *RW**IdentitySet***
        - *RW**Sequenceable***
            - *RW**DlistCollectables***
            - *RW**Ordered***
                - *RW**SortedVector***
            - *RW**SlistCollectables***
                - *RW**SlistCollectablesQueue***
                - *RW**SlistCollectablesStack***
    - *RW**CollectableAssociation***
    - *RW**CollectableDate*** *(&RWDate)*
    - *RW**CollectableInt*** *(&RWInteger)*
    - *RW**CollectableString*** *(&RWCString)*
    - *RW**CollectableTime*** *(&RWTime)*

    - *RW**ModelClient***
- *RW**CRegexp***
- *RW**CRExp***
- *RW**CString***
    - *RW**CollectableString*** *(&RWCollectable)*
- *RW**CSubString***
- *RW**CTokenizer***
- *RW**Date***
    - *RW**CollectableDate*** *(&RWCollectable)*
- *RW**ErrObject***
- *RW**Factory***
- *RW**File***
    - *RW**FileManager***
- *RW**GBitVec(size)***
- *RW**GDlist(type)***
- *RW**GDlistIterator(type)***
- *RW**GOrderedVector(val)***
- *RW**GQueue(type)***
- *RW**GSlist(type)***
- *RW**GSlistIterator(type)***
- *RW**GStack(type)***
- *RW**GVector(val)***
    - *RW**GSortedVector(val)***
- *RW**InstanceManager***
- *RW**Integer***
    - *RW**CollectableInt*** *(&RWCollectable)*
- *RW**Iterator***
    - *RW**BagIterator***
    - *RW**BinaryTreeIterator***
    - *RW**DlistCollectablesIterator***
    - *RW**HashDictionaryIterator***
    - *RW**HashTableIterator***
        - *RW**SetIterator***
    - *RW**OrderedIterator***
    - *RW**SlistCollectablesIterator***
- *RW**Locale***
    - *RW**LocaleSnapshot***
- *RW**Message***
- *RW**Model***
- *RW**Reference***
    - *RW**CStringRef***
    - *RW**VirtualRef***
    - *RW**WStringRef***

RWTime
RWCollectableTime (&RWCollectable)
RWTimer
RWTBitVec<size>
RWTIsvDlist<T>
RWTIsvDlistIterator<TL>
RWTIsvSlist<T>
RWTIsvSlistIterator<TL>
RWTPtrDeque<T>
RWTPtrDlist<T>
RWTPtrDlistIterator<T>
RWTPtrHashMap<Key,Type,Hash,EQ>
RWTPtrHashMapIterator<Key,Type,Hash,EQ>
RWTPtrHashMultiMap<Key,Type,Hash,EQ>
RWTPtrHashMultiMapIterator<Key,Type,Hash,EQ>
RWTPtrHashMultiSet<T,Hash,EQ>
RWTPtrHashMultiSetIterator<T,Hash,EQ>
RWTPtrHashSet<T,Hash,EQ>
RWTPtrHashSetIterator<T,Hash,EQ>
RWTPtrMap<Key,Type,Compare>
RWTPtrMapIterator<Key,Type,Compare>
RWTPtrMultiMap<Key,Type,Compare>
RWTPtrMultiMapIterator<Key,Type,Compare>
RWTPtrMultiSet<T,Compare>
RWTPtrMultiSetIterator<T,Compare>
RWTPtrOrderedVector<T>
RWTPtrSet<T,Compare>
RWTPtrSetIterator<T,Compare>
RWTPtrSlist<T>
RWTPtrSlistIterator<T>
RWTPtrSlistDictionary<KeyP,ValP>
RWTPtrSlistDictionaryIterator<KeyP,ValP>
RWTPtrSortedDlist<T,Compare>
RWTPtrSortedDlistIterator<T,Compare>
RWTPtrSortedVector<T,Compare>
RWTPtrVector<T>
RWTQueue<T,Container>
RWTRegularExpression<charT>
RWTStack<T,Container>
RWTValDeque<T>
RWTValDlist<T>
RWTValDlistIterator<T>
RWTValHashMap<Key,Type,Hash,EQ>
RWTValHashMapIterator<Key,Type,Hash,EQ>

- RWTValHashMultiMap<Key,Type,Hash,EQ>
- RWTValHashMultiMapIterator<Key,Type,Hash,EQ>
- RWTValHashMultiSet<T,Hash,EQ>
- RWTValHashMultiSetIterator<T,Hash,EQ>
- RWTValHashSet<T,Hash,EQ>
- RWTValHashSetIterator<T,Hash,EQ>
- RWTValMap<Key,Type,Compare>
- RWTValMapIterator<Key,Type,Compare>
- RWTValMultiMap<Key,Type,Compare>
- RWTValMultiMapIterator<Key,Type,Compare>
- RWTValMultiSet<T,Compare>
- RWTValMultiSetIterator<T,Compare>
- RWTValOrderedVector<T>
- RWTValSet<T,C>
- RWTValSetIterator<T,C>
- RWTValSlist<T>
- RWTValSlistIterator<T>
- RWTValSlistDictionary<Key,V>
- RWTValSlistDictionaryIterator<Key,V>
- RWTValSortedDlist<T,Compare>
- RWTValSortedDlistIterator<T,Compare>
- RWTValSortedVector<T>
- RWTValVector<T>
- RWTValVirtualArray<T>
- RWvios
  - RWios (virtual)
    - RWvistream
      - RWbistream (&ios: virtual)
        - RWeistream
      - RWpistream
      - RWXDRistream (&RWios)
    - RWvostream
      - RWbostream (&ios: virtual)
        - RWeostream
      - RWpostream
      - RWXDRostream (&RWios)
- RWVirtualPageHeap
  - RWBufferedPageHeap
    - RWDiskPageHeap
- RWWString
- RWWSubString
- RWWTokenizer
- RWZone
  - RWZoneSimple

streambuf
- *RW****AuditStreamBuffer***
- *RW****CLIPstreambuf***
  - *RW****DDEstreambuf***

xmsg
- *RW****xmsg***
  - *RW****ExternalErr***
    - *RW****FileErr***
    - *RW****StreamErr***
  - *RW****InternalErr***
    - *RW****BoundsErr***
  - *RW****xalloc***

Class Reference

*RW***AuditStreamBuffer** ⟶ ***streambuf***

Synopsis

```
#include <rw/auditbuf.h>
#include <iostream.h>
RWAuditStreamBuffer buf(arguments)
ostream os(&buf); // may be used for ostreams
istream is(&buf); // or istreams of any kind
```

Description

Class *RW***AuditStreamBuffer** is used to construct a stream, after which the *RW***AuditStreamBuffer** instance will count all the bytes that pass through the stream. If constructed with a function pointer, *RW***AuditStreamBuffer** will call that function with each byte that passes through the stream. The counting capacity provides for streams the equivalent of the *RW***Collectable** method `recursiveStoreSize()` which is only available for *RW***File**.

Persistence

None

Short Example

```
#include <rw/auditbuf.h>
#include <rw/bstream.h>
#include <rw/pstream.h>
#include <iostream.h>
int main() {
  RWCollectable ct;
  fillCollectable();  // make a collection, somehow
  RWAuditStreamBuffer bcounter, pcounter;
  RWbostream bcount(&bcounter); //ctor takes streambuf pointer
  RWpostream pcount(&pcounter);
//…
  bcount << ct;
  pcount << ct;
cout  << "We just counted " << bcounter
      << " bytes from an RWbostream." << endl;
cout  << "We just counted " << pcounter
      << " bytes from an RWpostream." << endl;
return 0;
}
```

Related Classes

*RW***AuditStreamBuffer** may be used as the streambuf for any stream, including those derived from *RW***vostream** or *RW***vistream**, ***strstream***, ***ifstream***, ***ofstream***, etc.

Global Typedef

```
typedef void (*RWauditFunction)(unsigned char, void*);
```

If you wish to do more than count each character handled by the buffer, you may provide an `RWauditFunction` to the constructor. The first parameter to this function is a byte provided by the stream. The second parameter is the address of the conter to be manipulated by *RW***AuditFunction**.

Public Constructors

```
RWAuditStreamBuffer(RWauditFunction=0, void*=0);
```

Constructs a new *RW**AuditStreamBuffer*** that may be used only to examine and count every byte that passes into an `ostream` that has the *RW**AuditStreamBuffer*** instance as its `streambuf`. It will not forward the bytes to any stream, nor accept bytes from a stream. The second argument to the constructor allows you to supply storage for the byte count. It is optional.

```
RWAuditStreamBuffer(istream&, RWauditFunction=0, void*=0);
```

Constructs a new *RW**AuditStreamBuffer*** that passes bytes from the `istream` on which it is constructed to the `istream` that has the *RW**AuditStreamBuffer*** instance as its `streambuf`. A typical use would be to count or examine the bytes being input from a file through a stream derived from *RW**v***`istream`. The second argument to the constructor allows you to supply storage for the byte count. It is optional.

```
RWAuditStreamBuffer(iostream&, RWauditFunction=0, void*=0);
```

Constructs a new *RW**AuditStreamBuffer*** that passes bytes to and from the i`ostream` on which it is constructed to and from the `istream` that has the *RW**AuditStreamBuffer*** instance as its `streambuf`. A typical use would be to count or examine the bytes being transferred to and from a file used to store and retrieve changing data. The second argument to the constructor allows you to supply storage for the byte count. It is optional.

```
RWAuditStreamBuffer(ostream&, RWauditFunction=0, void*=0);
```

Constructs a new *RW**AuditStreamBuffer*** that passes bytes into the `ostream` on which it is constructed from the `ostream` that has the *RW**AuditStreamBuffer*** instance as its `streambuf`. A typical use would be to count or examine the bytes being output to a file through a stream derived from *RW**vostream***. The second argument to the constructor allows you to supply storage for the byte count. It is optional.

```
RWAuditStreamBuffer(streambuf*, RWauditFunction=0, void*=0);
```

Constructs a new *RW**AuditStreamBuffer*** that passes bytes into the `ostream` on which it is constructed from the `ostream` that has the *RW**AuditStreamBuffer*** instance as its `streambuf`. A typical use would be to count or examine the bytes being output to a file through a stream derived from *RW**vostream***. The second argument to the constructor allows you to supply storage for the byte count. It is optional.

Public Destructor

```
virtual ~RWAuditStreamBuffer();
```

We have provided an empty destructor since some compilers complain if there is no virtual destructor for a class that has virtual methods.

Public Member Operator

```
operator unsigned long();
```

Provides the count of bytes seen so far.

Public Member Function

```
unsigned long
reset(unsigned long value = 0);
```

Resets the count of bytes seen so far. Returns the current count.

Extended Example

```
#include <iostream.h>
#include <fstream.h>
#include <rw/auditbuf.h>
#include <rw/pstream.h>
#include <rw/cstring.h>
void doCrc (unsigned char c, void* x) {
  *(unsigned char*)x ^= c;
}

int main() {
if(1) { // just a block to control variable lifetime
    unsigned char check = '\0';

    // create an output stream
    ofstream                            op("crc.pst");

    // create an RWAuditStreamBuffer that will do CRC
    RWAuditStreamBuffer               crcb(op,doCrc,&check);

    // create an RWpostream to put the data through.
RWpostream                            p(&crcb);

    // now send some random stuff to the stream
    p << RWCString("The value of Tools.h++ is at least ");
    p << (int)4;
    p << RWCString(" times that of the next best library!\n") ;
    p << RWCString("Pi is about ") << (double)3.14159 << '.';

    // finally, save the sum on the stream itself.
p << (unsigned int)check; // alters check, _after_ saving it...

    // just for fun, print out some statistics:
    cout << "We just saved " << crcb
         << " bytes of data to the file." << endl;
    cout << "The checksum for those bytes was " <<check << endl;
} // end of block

  // now read the data back in, checking to see if it survived.
  unsigned char check = '\0';

  // create an instream
  ifstream                            ip("crc.pst");

  // create an RWAuditStreamBuffer that will do CRC
  RWAuditStreamBuffer               crcb(ip,doCrc,&check);
```

```
  // create an RWpistream to interpret the bytes
  RWpistream                           p(&crcb);

  RWCString first, mid1, mid2;
  int value;
  double pi;
  char pnc;
  unsigned int savedCRC;
  unsigned char matchCRC;
  // read in the data. Don\'t read the checksum yet!
  p >> first >> value >> mid1 >> mid2 >> pi >> pnc;
  // save the checksum
  matchCRC = check;
  // Now it is safe to alter the running checksum by reading in
  // the one saved in the file.
p >> savedCRC;

  if(savedCRC != matchCRC) {
    cout << "Checksum error. Saved CRC: " << savedCRC
         << " built CRC: " << matchCRC << dec << endl;
  }
  else {
    cout << "The message was: " << endl;
    cout << first << value << mid1 << mid2 << pi << pnc << endl;
  }
  // just for fun, print out some statistics:
  cout  << "We just read " << crcb
        << " bytes of data from the file." << endl;
  cout  << "The checksum was " << matchCRC << flush;
  cout  << " and the saved checksum was " << savedCRC << endl;
return 0;
}
```

*RW***Bag** → *RW***Collection** → *RW***Collectable**

Synopsis

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

Description

Class *RW***Bag** 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 *RW***Bag** must inherit abstract base class *RW***Collectable**, with suitable definition for virtual functions `hash()` and `isEqual()` (see class *RW***Collectable**). The function `hash()` is used to find objects with the same hash value, then `isEqual()` is used to confirm the match.

Class *RW***Bag** is implemented by using an internal hashed dictionary (*RW***HashDictionary**) 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 that 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 *RW***HashTable** if you want duplicates to be stored, rather than merely counted.

Persistence

Polymorphic

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 *RW**Collection***. 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 or the meaning of "`isEqual`" of the items.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Redefined from class *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Collectable***.

```
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;
```

Inherited from class *RWCollectable*.

```
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);
```

Redefined from class *RWCollection*. Removes the item that `isEqual` to the item pointed to by `target`. Destroys the item as well if it is the last occurrence in the collection.

```
void
resize(size_t n = 0);
```

Resizes the internal hash table to have `n` buckets. The overhead for this function is the hashing of every element in 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``(`*RW***vostream**`&) const;`
`virtual void`
`saveGuts``(RWFile&) const;`

Inherited from class *RW**Collection***.

`RWStringID`
`stringID``();`

(acts virtual) Inherited from class *RW**Collectable***.

*RW***BagIterator** → *RW***Iterator**

Synopsis

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

Description

Iterator for class *RW***Bag**, which allows sequential access to all the elements of *RW***Bag**. Note that because an *RW***Bag** 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` consecutive times.

Like all Rogue Wave 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.

Persistence

None

Public Constructor

`RWBagIterator(const RWBag&);`

Construct an iterator for an *RW***Bag**. After construction, the position of the iterator is undefined.

Public Member Operator

```
virtual RWCollectable*
operator()();
```

Redefined from class *RW***Iterator**. Advances the iterator to the next 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 *RW***Iterator**. 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 *RW***Iterator**. Returns the item at the current iterator position.

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

Redefined from class *RW***Iterator**. Resets the iterator to its starting state.

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 *RW**Bench***, 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. *RW**Bench*** 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 all non-benchmark-related calculations.

Persistence

None

Example

This example benchmarks the time required to return a hash value for a Rogue Wave string versus a Borland string.

```
#include <rw/bench.h>                    /* Benchmark software */
#include <rw/cstring.h>                  /* Rogue Wave string class */
#include <stdlib.h>
#include <iostream.h>
#include <rw/ctoken.h>
#include <rw/regexp.h>

// The string to be hashed:
const char* cs = "A multi-character string with lots of words in it
to be parsed out and searched for.";

class TestBrute : public RWBench {
public:
TestBrute() { }
  virtual void       doLoop(unsigned long n);
  virtual void       idleLoop(unsigned long n);
  virtual void       what(ostream& s) const
    { s << "Brute force string search: \n"; }
};

class TestRW : public RWBench {
public:
TestRW() { }
  virtual void     doLoop(unsigned long n);
  virtual void       idleLoop(unsigned long n);
```

```
  virtual void       what(ostream& s) const
    { s << "Rogue Wave search: \n"; }
};


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

  // Test brute force string search algorithm:
  TestBrute other;
  other.parse(argc, argv);
  other.go();
  other.report(cout);

  // Test RW searching w/regular expressions:
  TestRW rw;
  rw.parse(argc, argv);
  rw.go();
  rw.report(cout);

  return 0;
}

void TestBrute::doLoop(unsigned long n){
  RWCString string(cs);
  RWCTokenizer *tokener;
  RWCString token;

  tokener = new RWCTokenizer(string);

  while(n--){

    if((token = (*tokener)()).isNull())
    {
        delete tokener;
        tokener = new RWCTokenizer(string);
        token = (*tokener)();
    }

    size_t j = 0;

    for(size_t i = 0; i < string.length() && j != token.length();
        i++)
    {
       j = 0;
       while((j < token.length()) && (string[i+j]==token[j]))
          j++;
    }

  }
 delete tokener;
}


void TestRW::doLoop(unsigned long n){
  RWCString string(cs);
  RWCTokenizer *tokener;
  RWCString token, result;
```

```
  RWCRegexp re("");

  tokener = new RWCTokenizer(string);
  while(n--){

   if((token = (*tokener)()).isNull())
    {
        delete tokener;
        tokener = new RWCTokenizer(string);
        token = (*tokener)();
    }
  re = RWCRegexp(token);
  result = string(re);       //Do the search!
  }
 delete tokener;
}
void TestBrute::idleLoop(unsigned long n){
  RWCString string(cs);           // Subtract out the overhead
  RWCTokenizer *tokener;
  RWCString token;

  tokener = new RWCTokenizer(string);

  while(n--){

   if((token = (*tokener)()).isNull())
    {
        delete tokener;
        tokener = new RWCTokenizer(string);
        token = (*tokener)();
    }
  }
 delete tokener;
}

void TestRW::idleLoop(unsigned long n){
  RWCString string(cs);                //Subtract out the overhead
  RWCTokenizer *tokener;
  RWCString token, result;
  RWCRegexp re("");
  tokener = new RWCTokenizer(string);
  while(n--){

   if((token = (*tokener)()).isNull())
    {
        delete tokener;
        tokener = new RWCTokenizer(string);
        token = (*tokener)();
    }
  re = RWCRegexp(token);
  }
 delete tokener;
}
```

Program output:

```
Testing string
"A multi-character string with lots of words in it to be parsed out
and searched for."
Borland C++ V4.0

Brute force string search:

Iterations:                  35
Inner loop operations:       1000
Total operations:            35000
Elapsed (user) time:         4.596
Kilo-operations per second: 7.61532

Borland C++ V4.0

Rogue Wave search:

Iterations:                  53
Inner loop operations:       1000
Total operations:            53000
Elapsed (user) time:         2.824
Kilo-operations per second: 18.7677
```

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 null 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 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 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.

```
const char *
machine();
```

This function accesses the name of the machine which is passed into the benchmark object through `parse()`.

```
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:

Argument	Type	Description
`argv[1]`	`double`	Duration (sec.)
`argv[2]`	`unsigned long`	No. of inner loops
`argv[3]`	`const char*`	Machine

```
void
parse(const char *);
```

This is a non-virtual function which provides the same service as `parse(int argc, char * argv[])`, but is designed for Windows users. It extracts tokens from the null-terminated command argument provided by Windows, then calls the virtual `parse` for ANSI C command arguments.

```
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.

*RW**BinaryTree*** → *RW**Collection*** → *RW**Collectable***

Synopsis

```
typedef RWBinaryTree SortedCollection;   // Smalltalk typedef.
#include <rw/bintree.h>
RWBinaryTree bt;
```

Description

Class *RW**BinaryTree*** represents a group of ordered elements, internally sorted by the `compareTo()` function. Duplicates are allowed. An object stored by an *RW**BinaryTree*** must inherit abstract base class *RW**Collectable***.

Persistence

Polymorphic

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 *RW**Collection***. Calls `clear()`.

Public Member Operators

`void`
`operator=``(const RWBinaryTree& bt);`

Sets self to a shallow copy of `bt`.

`void`
`operator+=``(const RWCollection ct);`

Inserts each element of `.ct` into self. Note that using this operator to insert an already-sorted collection will result in creating a very unbalanced tree, possibly to the point of stack overflow.

`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 a unique 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 a unique item in `bt`.

Public Member Functions

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

Redefined from class *RW**Collection*** 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 balance 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 one node. Since this collection allows duplicate elements, a perfectly balanced tree is not always possible. Preserves the order of duplicate elements.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Redefined from class *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Collectable***.

```
unsigned
height() const;
```

Returns the number of nodes between the root node and the farthest leaf. A *RWBinaryTree* with one entry will have a height of 1. Note that the entire tree is traversed to discover this value.

```
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()`. `insert()` does not automatically balance the *RWBinaryTree.* Be careful not to `insert()` a long sequence of sorted items without calling `balance()` since the result will be very unbalanced (and therefore inefficient).

```
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;
```

Inherited from class *RWCollectable*.

```
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 *RW**Collection*** to store objects by level, rather than in order. This results in the tree maintaining its morphology.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW**BinaryTreeIterator*** → *RW**Iterator***

Synopsis

```
// Smalltalk typedef:
typedef RWBinaryTreeIterator SortedCollectionIterator;
#include <rw/bintree.h>
RWBinaryTree bt;
RWBinaryTreeIterator iterate(bt);
```

Description

Iterator for class *RW**BinaryTree***. Traverses the tree from the "smallest" to "largest" element, where "smallest" and "largest" are defined by the virtual function `compareTo()`. Note that this approach is generally less efficient than using the member function `RWBinaryTree::apply()`.

Like all Rogue Wave 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.

Persistence

None

Public Constructor

```
RWBinaryTreeIterator(const RWBinaryTree&);
```

Constructs an iterator for an *RW**BinaryTree***. Immediately after construction, the position of the iterator is undefined until positioned.

Public Member Operator

```
virtual RWCollectable*
operator()();
```

Redefined from class *RW**Iterator***. 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 *RW**Iterator***. 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 *RW**Iterator***. Resets iterator to its state at construction.

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

Redefined from class *RW**Iterator***. Returns the item at the current iterator position.

*RW***bistream** → *RW***vistream** → *RW***ios** → *RW***vios**
→ **ios**

Synopsis

```
#include <rw/bstream.h>
RWbistream bstr(cin);        // Construct an RWbistream,
                             // using cin's streambuf
```

Description

Class *RW***bistream** specializes the abstract base class *RW***vistream** to restore variables stored in binary format by *RW***bostream**.

You can think of it as a binary veneer over an associated ***streambuf***. Because the *RW***bistream** 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***).

*RW***bistream** can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, *etc.*

Persistence

None

Example

See *RW***bostream** for an example of how the file "`data.dat`" might be created.

```
#include <rw/bstream.h>
#include <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 an *RW***bistream** from the `streambuf s`. For DOS, this `streambuf` must have been opened in binary mode.

```
RWbistream(istream& str);
```

Construct an *RW***bistream** using the `streambuf` associated with the `istream str`. For DOS, the `streambuf` must have been opened in binary mode. This can be done by specifying `ios::binary` as part of the second argument to the constructor for an ***ifstream***. Using the example above, the line to create the ***ifstream*** would read, `ifstream fstr("data.dat", ios::in | ios::binary);` where the "`|`" is the binary OR operator.

Public Operators

```
virtual RWvistream&
operator>>(char& c);
```

Redefined from class *RW****vistream***. Get the next `char` from the input stream and store it in `c`.

```
virtual RWvistream&
operator>>(wchar_t& wc);
```

Redefined from class *RW****vistream***. Get the next wide `char` from the input stream and store it in `wc`.

```
virtual RWvistream&
operator>>(double& d);
```

Redefined from class *RW****vistream***. Get the next `double` from the input stream and store it in `d`.

```
virtual RWvistream&
operator>>(float& f);
```

Redefined from class *RW****vistream***. Get the next `float` from the input stream and store it in `f`.

```
virtual RWvistream&
operator>>(int& i);
```

Redefined from class *RW****vistream***. Get the next `int` from the input stream and store it in `i`.

```
virtual RWvistream&
operator>>(long& l);
```

Redefined from class *RW****vistream***. Get the next `long` from the input stream and store it in `l`.

```
virtual RWvistream&
operator>>(short& s);
```

Redefined from class *RW****vistream***. Get the next `short` from the input stream and store it in `s`.

```
virtual RWvistream&
operator>>(unsigned char& c);
```

Redefined from class *RW****vistream***. Get the next `unsigned char` from the input stream and store it in `c`.

```
virtual RWvistream&
operator>>(unsigned short& s);
```

Redefined from class *RW***vistream**. Get the next `unsigned short` from the input stream and store it in `s`.

```
virtual RWvistream&
operator>>(unsigned int& i);
```

Redefined from class *RW***vistream**. Get the next `unsigned int` from the input stream and store it in `i`.

```
virtual RWvistream&
operator>>(unsigned long& l);
```

Redefined from class *RW***vistream**. Get the next `unsigned long` from the input stream and store it in `l`.

```
operator void*();
```

Inherited via *RW***vistream** from *RW***vios**.

Public Member Functions

```
virtual int
get();
```

Redefined from class *RW***vistream**. 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 *RW***vistream**. Get the next `char` and store it in `c`.

```
virtual RWvistream&
get(wchar_t& wc);
```

Redefined from class *RW***vistream**. Get the next wide `char` and store it in `wc`.

```
virtual RWvistream&
get(unsigned char& c);
```

Redefined from class *RW***vistream**. Get the next `unsigned char` and store it in `c`.

```
virtual RWvistream&
get(char* v, size_t N);
```

Redefined from class *RW***vistream**. Get a vector of `char`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(wchar_t* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of wide `char`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(double* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `double`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(float* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `float`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(int* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `int`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(long* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `long`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(short* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `short`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(unsigned char* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `unsigned char`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(unsigned short* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `unsigned short`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit..

```
virtual RWvistream&
get(unsigned int* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `unsigned int`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(unsigned long* v, size_t N);
```

Redefined from class *RW**vistream***. Get a vector of `unsigned long`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
getString(char* s, size_t N);
```

Redefined from class *RW**vistream***. 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 `N-1` characters have been read and the `Nth` character is not the string terminator, 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 *RW**vistream***. 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 `N-1` characters have been read and the `Nth` character is not the string terminator, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte.

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.

Persistence

Simple

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 *RW**InternalErr*** will occur.

Indexing Operators

```
RWBitRef
operator[](size_t i);
```
Returns a reference to bit `i` of self. A helper class, *RW**BitRef***, 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 *RW**BoundsErr*** will occur.

```
RWBitRef
operator()(size_t i);
```
Returns a reference to bit `i` of self. A helper class, *RW**BitRef***, 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 `<rw/bitvec.h>`. If so, and if the index is out of range, then an exception of type *RW**BoundsErr*** 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 *RW**BoundsErr*** 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 *RW**BoundsErr*** 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 FALSE bit in self. Returns RW_NPOS if there is no FALSE bit.

```
size_t
firstTrue() const;
```
Returns the index of the first `TRUE` bit in self. Returns `RW_NPOS` if there is no `TRUE` 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);
```
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 *RW**BitVec*** `vec` to a virtual stream or *RW**File***, respectively.

```
RWvistream&
operator>>(RWvistream&, RWBitVec& vec);
RWFile&
operator>>(RWFile&,     RWBitVec& vec);
```

Restores an *RW**BitVec*** into `vec` from a virtual stream or *RW**File***, respectively, replacing the previous contents of `vec`.

```
size_t
sum(const RWBitVec& v);
```

Returns the total number of bits set in the vector `v`.

*RW***bostream**

*RW***bostream** → *RW***vostream** → *RW***ios** → *RW***vios**

*RW***bostream** → ***ios***

Synopsis

```
#include <rw/bstream.h>
// Construct an RWbostream, using cout's streambuf:
RWbostream bstr(cout);
```

Description

Class *RW***bostream** specializes the abstract base class *RW***vostream** to store variables in binary format. The results can be restored by using its counterpart *RW***bistream**.

You can think of it as a binary veneer over an associated ***streambuf***. Because the *RW***bostream** 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 that variables should not be separated with white space. Such white space would be interpreted literally and would have to be read back in as a character string.

*RW***bostream** can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, *etc.*

Persistence

None

Example

See *RW***bistream** 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 an *RW**bostream*** from the `streambuf s`. For DOS, the `streambuf` must have been opened in binary mode.

```
RWbostream(ostream& str);
```

Construct an *RW**bostream*** from the `streambuf` associated with the output stream `str`. For DOS, the `streambuf` must have been opened in binary mode. This can be done by specifying `ios::binary` as part of the second argument to the constructor for an ***ofstream***. Using the example above, the line to create the ***ofstream*** would read, `ofstream fstr("data.dat", ios::out | ios::binary);` where the "`|`" is the binary `OR` operator.

Public Destructor

```
virtual ~RWvostream();
```

This virtual destructor allows specializing classes to deallocate any resources that they may have allocated.

Public Operators

```
virtual RWvostream&
operator<<(const char* s);
```

Redefined from class *RW**vostream***. 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 *RW**vostream***. Store the wide character string starting at `ws` to the output stream in binary. The wide character string is expected to be null terminated.

```
virtual RWvostream&
operator<<(char c);
```

Redefined from class *RW**vostream***. Store the `char c` to the output stream in binary.

```
virtual RWvostream&
operator<<(wchar_t wc);
```

Redefined from class *RW**vostream***. Store the wide `char wc` to the output stream in binary.

```
virtual RWvostream&
operator<<(unsigned char c);
```

Redefined from class *RW**vostream***. Store the `unsigned char c` to the output stream in binary.

```
virtual RWvostream&
operator<<(double d);
```
Redefined from class *RW**vostream***. Store the `double d` to the output stream in binary.

```
virtual RWvostream&
operator<<(float f);
```
Redefined from class *RW**vostream***. Store the `float f` to the output stream in binary.

```
virtual RWvostream&
operator<<(int i);
```
Redefined from class *RW**vostream***. Store the `int i` to the output stream in binary.

```
virtual RWvostream&
operator<<(unsigned int i);
```
Redefined from class *RW**vostream***. Store the `unsigned int i` to the output stream in binary.

```
virtual RWvostream&
operator<<(long l);
```
Redefined from class *RW**vostream***. Store the `long l` to the output stream in binary.

```
virtual RWvostream&
operator<<(unsigned long l);
```
Redefined from class *RW**vostream***. Store the `unsigned long l` to the output stream in binary.

```
virtual RWvostream&
operator<<(short s);
```
Redefined from class *RW**vostream***. Store the `short s` to the output stream in binary.

```
virtual RWvostream&
operator<<(unsigned short s);
```
Redefined from class *RW**vostream***. Store the `unsigned short s` to the output stream in binary.

```
operator void*();
```
Inherited via *RW**vostream*** from *RW**vios.***

Public Member Functions

```
virtual RWvostream&
flush();
```
Send the contents of the stream buffer to output immediately.

```
virtual RWvostream&
put(char c);
```
Redefined from class *RW***vostream**. Store the `char c` to the output stream.

```
virtual RWvostream&
put(wchar_t wc);
```
Redefined from class *RW***vostream**. Store the wide character `wc` to the output stream.

```
virtual RWvostream&
put(unsigned char c);
```
Redefined from class *RW***vostream**. Store the `unsigned char c` to the output stream.

```
virtual RWvostream&
put(const char* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of `char`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
put(const wchar_t* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of wide `char`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
put(const unsigned char* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of `unsigned char`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
put(const short* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of `short`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
put(const unsigned short* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of `unsigned short`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
put(const int* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of `int`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
put(const unsigned int* p, size_t N);
```
Redefined from class *RW***vostream**. Store the vector of `unsigned int`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
```
put`(const long* p, size_t N);`

Redefined from class *RW***vostream**. Store the vector of `long`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
```
put`(const unsigned long* p, size_t N);`

Redefined from class *RW***vostream**. Store the vector of `unsigned long`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
```
put`(const float* p, size_t N);`

Redefined from class *RW***vostream**. Store the vector of `float`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
```
put`(const double* p, size_t N);`

Redefined from class *RW***vostream**. Store the vector of `double`s starting at `p` to the output stream in binary.

```
virtual RWvostream&
```
putString`(const char* p, size_t N);`

Redefined from class *RW***vostream**. Data is formatted as a string containing `N` characters.

```
virtual RWvostream&
```
putString`(const char*s, size_t N);`

Store the character string, *including embedded nulls*, starting at s to the output string.

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.

Persistence

Polymorphic

Public Constructors

RWBTree();

Construct an empty B-tree.

RWBTree(const RWBTree& btr);

Construct self as a shallow copy of `btr`.

Public Destructor

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. **Note**: If you inherit from ***RWBTree*** in the presence of the Standard C++ Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

```
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 *RW**Collection*** 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 *RW**Collection***.

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

Redefined from class *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Redefined from class *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Collectable***.

```
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 *RW**Collection***. Inserts the item `c` into the collection and returns it. The item `c` is inserted according to the value returned by `compareTo()`. 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. Otherwise returns `nil` if the insertion was unsuccessful.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW**Collectable*** to return `__RWBTREE`.

```
virtual RWBoolean
isEmpty() const;
```

Redefined from class *RW**Collection***.

```
virtual RWBoolean
isEqual(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Collection***. 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 *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

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.

Persistence

Polymorphic

Public Constructors

`RWBTreeDictionary();`
Constructs an empty B-tree dictionary.

Public Member Operators

`RWBoolean`
`operator<=``(const RWBTreeDictionary& 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. This operator is not explicitly present unless you are compiling with an implementation of the C++ Standard Library. Normally it is inherited from *RWBTree*.

Note: If you inherit from *RWBTreeDictionary* in the presence of the C++ Standard Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

Public Member Functions

`void`
`applyToKeyAndValue``(RWapplyKeyAndValue ap,void*);`
Redefined from class *RWCollection*. Applies the user-supplied function pointed to by `ap` to each key-value pair of the collection, in order, from smallest to largest.

```
RWBinaryTree
asBinaryTree();
RWBag
asBag() const;
RWSet
asSet() const;
RWOrdered
asOrderedCollection() const;
RWBinaryTree
asSortedCollection() const:
```

Converts the *RW**BTreeDictionary*** to an *RW**Bag***, *RW**Set***, *RW**Ordered***, or an *RW**BinaryTree***. Note that since a dictionary contains pairs of keys and values, the result of this call will be a container holding *RW**CollectableAssociations***. Note also that the return value is a *copy* of the data. This can be very expensive for large collections. Consider using `operator+=()` to insert each *RW**CollectableAssociation*** from this dictionary into a collection of your choice.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***. Removes all key-value pairs from the collection.

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

Redefined from class *RW**Collection***. 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 *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Redefined from class *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Collectable***.

```
unsigned
height() const;
```

Inherited from class *RW**BTree***.

```
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 *RW**Collectable*** to return `__RWBTREEDICTIONARY`.

```
virtual RWBoolean
isEmpty() const;
```

Inherited from class *RW**BTree***.

```
virtual RWBoolean
isEqual(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Collection***. 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 *RW**Collection***. Removes *and* deletes the key and value pair for which the key compares equal to the object pointed to by `target`. Note that 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 *RW**Collection***.

```
virtual RWCollection*
select(RWtestCollectable testfunc, void* x) const;
```

Evaluates the function pointed to by `tst` for the key of each item in the *RW**BTreeDictionary***. It inserts keys and values for which the function returns `TRUE` into a new *RW**BTreeDictionary*** allocated off the heap and returns a pointer to this new collection. Because the new dictionary is allocated *off the heap*, you are responsible for deleting it when done. This is *not* a virtual function.

```
virtual RWCollection*
select(RWtestCollectablePair testfunc, void* x) const;
```

Evaluates the function pointed to by `tst` for both the key and the value of each item in the *RW**BTreeDictionary***. It inserts keys and values for which the function returns `TRUE` into a new *RW**BTreeDictionary*** allocated off the heap and returns a pointer to this new collection. Because the new dictionary is allocated *off the heap*, you are responsible for deleting it when done. This is *not* a virtual function.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

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 `baseLocation()`.

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 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 may result in less time spent balancing the tree, but less efficient use of disk space.

Persistence

None

Enumerations

```
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 *RW**FileManager'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 the prototype:

```
void yourApplyFunction(const char* ky,
                       RWstoredValue val,void* x);
```

The function `yourApplyFunction` *may not* change the key. The value `x` can be anything and is passed through from the call to `applyToKeyAndValue()`. This member function may throw an *RW**FileErr*** exception.

```
RWoffset
baseLocation() const;
```

Returns the offset of this tree's starting location within the *RW**FileManager***. This is the value you will pass to a constructor as the `start` argument when you want to open one of several trees stored in one managed file.

```
unsigned
cacheCount() const;
```

Returns the maximum number of nodes that may currently be cached.

```
unsigned
cacheCount(unsigned newcount);
```

Sets the number of nodes that should be cached to `newcount`. Returns the old number.

```
void
clear();
```

Removes all items from the collection.This member function may throw an *RW**FileErr*** exception.

```
RWBoolean
contains(const char* ky) const;
```

Returns `TRUE` if the tree contains a key that is equal to the string pointed to by `ky`, and `FALSE` otherwise. This member function may throw an *RW**FileErr*** exception.

```
size_t
entries();
```

Returns the number of items in the *RW**BTreeOnDisk***. This member function may throw an *RW**FileErr*** exception.

```
RWoffset
extraLocation(RWoffset newlocation);
```

Sets the location where this *RW**BTreeOnDisk*** keeps your own application-specific information to `newlocation`. Returns the previous value.

```
RWBoolean
findKey( const char* ky, RWCString& foundKy)const ;
```

Returns `TRUE` if `ky` is found, otherwise `FALSE.` If successful, the found key is returned as a reference in `foundKy`. This member function may throw an *RW**FileErr*** exception.

```
RWBoolean
findKeyAndValue( const char* ky,
                 RWCString& foundKy,
                 RWStoredValue& foundVal)const ;
```

Returns `TRUE` if `ky` is found, otherwise `FALSE.` If successful, the found key is returned as a reference in `foundKy`, and the value is returned as a reference in `foundVal`. This member function may throw an *RW**FileErr*** exception.

```
RWstoredValue
findValue(const char* ky)const;
```

Returns the value for the key that compares equal to the string pointed to by `ky`. Returns `RWNIL` if no key is found. This member function may throw an *RW**FileErr*** exception.

```
int
height();
```

Returns the height of the *RW**BTreeOnDisk***. A possible exception is *RW**FileErr***.

```
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 is *RW**FileErr***.

```
unsigned
keyLength() const;
```

Return the length of the keys for this *RW**BtreeOnDisk***. This number is set when the tree is first constructed and cannot be changed.

```
unsigned
minOrder()const;
```

Return the minimum number of items that may be found in any non-root node in this *RW**BtreeOnDisk***. This number is set when the tree is first constructed and cannot be changed.

```
unsigned
nodeSize() const;
```

Returns the number of bytes used by each node of this *RW**BTreeOnDisk***. This number is calculated from the length of the keys and the order of the tree, and cannot be changed. We make it available to you for your calculations about how many nodes to cache.

```
unsigned
order()const;
```

Return half the maximum number of items that may be stored in any node in this *RW**BtreeOnDisk***. This number is set when the tree is first constructed and cannot be changed. This method should have been renamed “`halfOrder`” but is still called “`order`” for backward compatibility.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if the *RW**BTreeOnDisk*** is empty, otherwise `FALSE`.

```
void
remove(const char* ky);
```

Removes the key and value pair that has a key which matches `ky`. This member function may throw an *RW**FileErr*** exception.

```
RWBoolean
replaceValue(const RWCString& key,
             const RWstoredValue newval,
             RWstoredValue& oldVal);
```

Attempts to replace the `RWstoredValue` now associated with `key` by the value `newval`. If successful, the previous value is returned by reference in `oldVal`; and the methed returns `TRUE`. Otherwise, returns `FALSE`.

```
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).

*RW***BufferedPageHeap** → *RW***VirtualPageHeap**

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 *RW***VirtualPageHeap**, which represents an abstract page heap.

*RW***BufferedPageHeap** 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 *RW***DiskPageHeap**.

Persistence

None

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 *RW**VirtualPageHeap***. 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 *RW**VirtualPageHeap***.

```
virtual void*
lock(RWHandle h);
```

Redefined from class *RW**VirtualPageHeap***.

```
virtual void
unlock(RWHandle h);
```

Redefined from class *RW**VirtualPageHeap***.

Synopsis

```
#include <rw/cacheman.h>
RWFile f("file.dat");        // Construct a file
RWCacheManager(&f, 100);    // Cache 100 byte blocks to file.dat
```

Description

Class *RW**CacheManager*** caches fixed length blocks to and from an associated *RW**File***. 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.

Persistence

None

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 *RW**File*** pointed to by `file`. The length of the fixed-size blocks is given by `blocksz`. The number of cached blocks is given by `mxblks`. If the total number of bytes cached would exceed the maximum value of an unsigned int, then *RW**CacheManager*** will quietly decide to cache a smaller number of blocks.

Public Destructor

```
~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`.

*RW***CLIPstreambuf** → ***streambuf***

Synopsis

```
#include <rw/winstrea.h>
#include <iostream.h>
iostream str( new RWCLIPstreambuf() );
```

Description

Class *RW***CLIPstreambuf** is a specialized ***streambuf*** that gets and puts sequences of characters to Microsoft Windows global memory. It can be used to exchange data through Windows clipboard facility.

The class has two modes of operation: dynamic and static. In dynamic mode, memory is allocated and reallocated as needed. If too many characters are inserted into the internal buffer for its present size, then it will be resized and old characters copied over into any new memory as necessary. This is transparent to the user. It is expected that this mode would be used primarily for "insertions," *i.e.*, clipboard "cuts" and "copies." In static mode, the buffer streambuf is constructed from a specific piece of memory. No reallocations will be done. It is expected that this mode would be used primarily for "extractions," *i.e.*, clipboard "pastes."

In dynamic mode, the *RW***CLIPstreambuf** "owns" any allocated memory until the member function `str()` is called, which "freezes" the buffer and returns an unlocked Windows handle to it. The effect of any further insertions is undefined. Until `str()` has been called, it is the responsibility of the *RW***CLIPstreambuf** destructor to free any allocated memory. After the call to `str()`, it becomes the user's responsibility.

In static mode, the user has the responsibility for freeing the memory handle. However, because the constructor locks and dereferences the handle, you should not free the memory until either the destructor or `str()` has been called, either of which will unlock the handle.

Persistence

None

Example

```
//Instructions:  compile as a Windows program.
//Run this program, then using your favorite text editor or word
//processor, select paste and see the result!

#include <rw/winstrea.h>

#include <stdlib.h>
#include <iostream.h>
#include <windows.h>


void postToClipboard(HWND owner);
```

```
main()
{
   postToClipboard(NULL);

   return 0;
}


// PASS YOUR WINDOW HANDLE TO THIS FUNCTION THEN PASS YOUR VALUES
// TO THE CLIPBOARD USING ostr.

void postToClipboard(HWND owner)
{
   //Build the clipstream buffer on the heap
   RWCLIPstreambuf* buf = new
   RWCLIPstreambuf();

   ostream ostr(buf);

   double d = 12.34;

   ostr << "Some text to be exchanged through the clipboard.\n";
   ostr << "Might as well add a double: " << d << endl;
   ostr.put('\0');        // Include the terminating null

   // Lock the streambuf, get its handle:
   HANDLE hMem = buf->str();

   OpenClipboard(owner);

   EmptyClipboard();
   SetClipboardData(CF_TEXT, hMem);
   CloseClipboard();

   // Don't delete the buffer!.  Windows is now responsible for it.
}
```

The owner of the clipboard is passed in as parameter "`owner`". A conventional ***ostream*** is created, except that it uses an *RW**CLIPstreambuf*** as its associated ***streambuf***. It can be used much like any other ***ostream***, such as `cout`, except that characters will be inserted into Windows global memory.

Some text and a double is inserted into the ***ostream***. Finally, member function `str()` is called which returns a Windows `HANDLE`. The clipboard is then opened, emptied, and the new data put into it with format `CF_TEXT` which, in this case, is appropriate because a simple ***ostream*** was used to format the output. If a specializing virtual streams class such as *RW**bostream*** or *RW**postream*** had been used instead, the format is not so simple. In this case, the user might want to register his or her own format, using the Windows function `RegisterClipboardFormat()`.

Public Constructors

`RWCLIPstreambuf``();`

Constructs an empty *RW**CLIPstreambuf*** in dynamic mode. The results can be used anywhere any other ***streambuf*** can be used. Memory to accomodate new characters will be allocated as needed.

`RWCLIPstreambuf``(HANDLE hMem);`

Constructs an *RW**CLIPstreambuf*** in static mode, using the memory block with global handle `hMem`. The effect of `gets` and `puts` beyond the size of this memory block is unspecified.

Public Destructor

`~`**`RWCLIPstreambuf`**`();`

If member function `str()` has not been called, the destructor unlocks the handle and, if in dynamic mode, also frees it.

Public Member Functions

Because *RW**CLIPstreambuf*** inherits from ***streambuf***, any of the latter's member functions can be used. Furthermore, *RW**CLIPstreambuf*** has been designed to be analogous to ***strstreambuf***. However, note that the return type of `str()` is a `HANDLE`, rather than a `char*`.

`HANDLE`
`str``();`

Returns an (unlocked) `HANDLE` to the global memory being used. The *RW**CLIPstreambuf*** should now be regarded as "frozen": the effect of inserting any more characters is undefined. If the *RW**CLIPstreambuf*** was constructed in dynamic mode, and nothing has been inserted, then the returned `HANDLE` may be `NULL`. If it was constructed in static mode, then the returned handle will be the handle used to construct the *RW**CLIPstreambuf***.

Synopsis

```
typedef RWCollectable Object;  // Smalltalk typedef
#include <rw/collect.h>
```

Description

Class *RW***Collectable** 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.

Persistence

Polymorphic

Virtual Functions

```
virtual
~RWCollectable();
```

All functions that inherit class *RW***Collectable** 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 *RW***Collectable**. See the *Tool.h++ User's Guide* Section entitled "Virtual Function binaryStoreSize" 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 *RW***Collectable** 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 that 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;

It is important that the hash value be the same for all objects which return `TRUE` to `isEqual()`.

```
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 Rogue Wave 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 *RW**File***, 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 *RW**File***.

```
virtual void
saveGuts(RWvostream&) const;
```

Write an object's state to an output stream.

```
RWStringID
stringID();
```

Returns the identification string for the class. Acts virtual, although it is not.[1]

```
RWspace
recursiveStoreSize() const;
```

Returns the number of bytes required to store the object using the global operator

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

Recursively calls `binaryStoreSize()`, taking duplicate objects into account.

Static Public Member Functions

```
static RWClassID
classID(const RWStringID& name);
```

Returns the result of looking up the `RWClassID` associated with `name` in the global RWFactory.

```
static RWClassID
classIsA();
```

Returns the `RWClassID` of this class.

```
static RWBoolean
isAtom(RWClassID id);
```

Returns `TRUE` if `id` is the *RW**ClassID*** that is associated with an *RW**Collectable*** class that has a programmer-chosen *RW**StringID***.

```
static RWspace
nilStoreSize();
```

Returns the number of bytes required to store a `rwnil` pointer in an *RW**File***.

---

1 See the section in the User's Guide entitled "RWStringID" for more information on how to make a non-virtual function act like a virtual function.

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 the *Tools.h++ User's Guide* section entitled "Persistence" for more information.

```
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 the *Tools.h++ User's Guide* section entitled "Persistence" for more information. 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 the *Tools.h++ User's Guide* section entitled "Persistence" for more information. 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). In case an exception is thrown during this call, the pointer to the partly restored object will probably be lost, and memory will leak. For this reason, you may prefer to use the static methods `tryRecursiveRestore()` documented above.

*RW***CollectableAssociation** → *RW***Collectable**

Synopsis

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

Description

*RW***CollectableAssociation** inherits class *RW***Collectable**. Used internally to associate a key with a value in the *Tools.h++* "dictionary" collection classes. Comparison and equality testing are forwarded to the key part of the association.

Persistence

Polymorphic

Related Classes

The "dictionary containers" *RW***BTreeDictionary**, *RW***HashDictionary**, and *RW***IdentityDictionary** make use of *RW***CollectableAssociation**. When any of their contents is dealt with as an *RW***Collectable**, as when `operator+=()` or `asBag()` etc. is used, the *RW***CollectableAssociation** will be exposed.

Public Constructors

```
RWCollectableAssociation();
RWCollectableAssociation(RWCollectable* k, RWCollectable* v);
```

Construct an *RW***CollectableAssociation** with the given key and value.

Public Destructor

```
virtual ~RWCollectableAssociation();
virtual RWspace
binaryStoreSize() const;
```

Redefined from class *RW***Collectable**.

Public Member Functions

```
virtual int
compareTo(const RWCollectable* c) const;
```

Redefined from class *RW***Collectable**. Returns the results of calling `key()->compareTo(c)`.

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

Redefined from class *RW***Collectable**. Returns the results of calling `key()->hash()`.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW***Collectable** to return `__RWCOLLECTABLEASSOCIATION`.

```
virtual RWBoolean
isEqual(const RWCollectable* c) const;
```

Redefined from class *RW**Collectable***. Returns the results of calling `key()->isEqual(c)`.

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

Returns the key part of the association.

```
RWCollectable*
value() const;
```

Returns the value part of the association.

```
RWCollectable*
value(RWCollectable* ct);
```

Sets the value to `ct` and returns the old value.

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

Redefined from class *RW**Collectable***.

*RW**CollectableDate*** → *RW**Collectable***
*RW**CollectableDate*** → *RW**Date***

Synopsis

```
typedef RWCollectableDate Date;  // Smalltalk typedef
#include <rw/colldate.h>
RWCollectableDate  d;
```

Description

Collectable Dates. Inherits classes *RW**Date*** and *RW**Collectable***. This class is useful when dates are used as keys in the "dictionary" collection classes, or if dates are stored and retrieved as *RW**Collectables***. The virtual functions of the base class *RW**Collectable*** have been redefined.

Persistence

Polymorphic

Public Constructors

```
RWCollectableDate();
RWCollectableDate(unsigned long julianDate);
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 *RW**Date***.

Public Member Functions

```
virtual RWspace
binaryStoreSize() const;
```

Redefined from class *RW**Collectable***.

```
virtual int
compareTo(const RWCollectable* c) const;
```

Redefined from class *RW**Collectable***. Returns the results of calling `RWDate::compareTo`.

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

Redefined from class *RW**Collectable***. 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*.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RWCollectable*.

*RW***CollectableInt** → *RW***Collectable**, *RW***Integer**

Synopsis

```
typedef RWCollectableInt Integer;  // Smalltalk typedef
#include <rw/collint.h>
RWCollectableInt  i;
```

Description

Collectable integers. Inherits classes *RW***Integer** and *RW***Collectable**. This class is useful when integers are used as keys in the "dictionary" collection classes, or if integers are stored and retrieved as *RW***Collectables**. The virtual functions of the base class *RW***Collectable** have been redefined.

Persistence

Polymorphic

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 *RW***Collectable**.

```
virtual int
compareTo(const RWCollectable* c) const;
```

Redefined from class *RW***Collectable**. Returns the difference between self and the *RW***CollectableInt** pointed to by `c`.

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

Redefined from class *RW***Collectable**. Returns the *RW***CollectableInt**'s value as an `unsigned`, to be used as a hash value.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW***Collectable** to return `__RWCOLLECTABLEINT`.

```
virtual RWBoolean
isEqual(const RWCollectable* c) const;
```

Redefined from class *RW**Collectable***. Returns `TRUE` if self has the same value as the *RW**CollectableInt*** at address `c`.

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

Redefined from class *RW**Collectable***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW***CollectableString** → *RW***Collectable**
*RW***CollectableString** → *RW***CString**

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 *RW***Collectables**, or when they are used as keys in the "dictionary" collection classes. Class *RW***CollectableString** inherits from both class *RW***CString** and class *RW***Collectable**. The virtual functions of the base class *RW***Collectable** have been redefined.

Persistence

Polymorphic

Public Constructors

`RWCollectableString();`
Construct an *RW***CollectableString** with zero characters.

`RWCollectableString(const RWCString& s);`
Construct an *RW***CollectableString** from the *RW***CString** `s`.

`RWCollectableString(const char* c);`
Conversion from character string.

`RWCollectableString(const RWCSubString&);`
Conversion from sub-string.

`RWCollectableString(char c, size_t N);`
Construct an *RW***CollectableString** with `N` characters (default blanks).

Public Member Functions

`virtual RWspace`
`binaryStoreSize() const;`
Redefined from class *RW***Collectable**.

`virtual int`
`compareTo(const RWCollectable* c) const;`
Redefined from class *RW***Collectable**. returns the result of `RWCString::compareTo(*(const String*)c, RWCString::exact)`. This compares strings lexicographically, with case considered. It would be possible to define , for instance, `CaseFoldedString` which did comparisons ignoring case. We have deliberately left this as an exercise for two reasons: Because it is both easy to do and not universally needed; and because the presence of both *RW***CollectableStrings** and such a

CaseFoldedString in any kind of sorted collection has the potential for very confusing behavior, since the result of a comparison would depend on the order in which the comparison was done.

```
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*.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RWCollectable*.

*RW***CollectableTime** → *RW***Collectable**, *RW***Time**

Synopsis

```
typedef RWCollectableTime Time;  // Smalltalk typedef
#include <rw/colltime.h>
RWCollectableTime  t;
```

Description

Inherits classes *RW***Time** and *RW***Collectable**. This class is useful when times are used as keys in the "dictionary" collection classes, or if times are stored and retrieved as *RW***Collectables**. The virtual functions of the base class *RW***Collectable** have been redefined.

Persistence

Polymorphic

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 *RW***Time**.

Public Member Functions

```
virtual RWspace
binaryStoreSize() const;
```

Redefined from class *RW***Collectable**.

```
virtual int
compareTo(const RWCollectable* c) const;
```

Redefined from class *RW***Collectable**. Returns the results of calling `RWTime::compareTo`.

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

Redefined from class *RW***Collectable**. Returns the results of calling `RWTime::hash()`.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW***Collectable** to return `__RWCOLLECTABLETIME`.

```
virtual RWBoolean
isEqual(const RWCollectable* c) const;
```

Redefined from class *RW**Collectable***. Returns the results of calling `operator==()` for the base class *RW**Time*** 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 *RW**Collectable***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW***Collection** → *RW***Collectable**

Synopsis

```
#include <rw/colclass.h>
typedef RWCollection Collection;   // Smalltalk typedef
```

Description

Class *RW***Collection** is an abstract base class for the Smalltalk-like collection classes. The class contains virtual functions for inserting and retrieving pointers to *RW***Collectable** 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 *RW***Collection** and a deriving class may not need to redefine the function. Examples are `contains()` or `restoreGuts()`.

Persistence

Polymorphic

Public Member Operators

```
void
operator+=(const RWCollection&);
void
operator-=(const RWCollection&);
```

Adds or removes, respectively, each item in the argument to or from self. Using `operator+=(somePreSortedCollection)` on an *RW***BinaryTree** can cause that tree to become unbalanced; possibly to the point of stack overflow.

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* ctp, void*);
```

The function `yourApplyFunction()` can perform any operation on the item at address `ctp` that *does not change* the hash value or sorting order of the item. Client data may be passed to this function through the second argument.

```
RWBag
asBag() const;
RWSet
asSet() const;
RWOrdered
asOrderedCollection() const;
RWBinaryTree
asSortedCollection() const;
```

Allows any collection to be converted to an *RW***Bag**, *RW***Set**, *RW***Ordered**, or an *RW***BinaryTree**. Note that the return value is a *copy* of the data. This can be very expensive for large collections. You should consider using `operator+=()` to insert each item from this collection into a collection of your choice. Also note that converting a collection containing data which is already sorted to a *RW***BinaryTree** via the `asSortedCollection()` or `asBinaryTree()` methods will build a very unbalanced tree.

```
virtual RWspace
binaryStoreSize() const;
```

Redefined from class *RW***Collectable**.

```
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 *RW***Collectable**.

```
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.*, *RW**IdentitySet*** and *RW**IdentityDictionary***) look for an item with the same address (*i.e.*, "is identical to").

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

Inherited from class *RW**Collectable***.

```
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 collections derived from *RW**Collection*** return the old instance, others return `nil`.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW**Collectable*** 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 *RW**Collectable***.

```
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 void
restoreGuts(RWFile&);
```

Redefined to repeatedly call the global operator

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

followed by `insert(RWCollectable*)` for each item in the collection.

```
virtual void
restoreGuts(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(RWtestCollectable 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. This is *not* a virtual function.

```
virtual void
saveGuts(RWFile&);
```

Redefined to call the global operator

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

for each object in the collection.

```
virtual void
saveGuts(RWvostream&);
```

Redefined to call the global operator

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

for each object in the collection.

Synopsis

```
#include <rw/regexp.h>
RWCRegexp re(".*\\.doc");// Matches filename with suffix ".doc"
```

Description

Class *RW**CRegexp*** 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 *RW**CString***.

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. *E.g.*, "*.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. *E.g.*, "*[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. *E.g. xy?z* matches either *xyz* or *xz*.

2.4 The concatenation of REs is a RE that matches the corresponding concatenation of strings. *E.g.*, [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 `0xdd`
`\ddd`	the literal octal number `ddd`
`\^C`	Control code. *E.g.* `\^D` is "control-D"

Persistence

None

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(const char* pat);

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`.

Public Destructor

`~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 that 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 and resets status to `OK`:

statVal	Meaning
`RWCRegexp::OK`	No errors
`RWCRegexp::ILLEGAL`	Pattern was illegal
`RWCRegexp::TOOLONG`	Pattern exceeded maximum length[1]

1 To change the amount of space allocated for a pattern you may edit file `regexp.cpp` to change the value of `RWCRegexp::maxval_`, then recompile and insert the changed object into the appropriate library.

Synopsis

```
#include <rw/re.h>
RWCRExpr re(".*\\.doc");  // Matches filename with suffix ".doc"
```

Description

Class *RW**CRExpr*** represents an **extended** regular expression such as those found in `lex` and `awk`. 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 *RW**CString***. Regular expressions can be of arbitrary size, limited by memory. The extended regular expression features found here are a subset of those found in the POSIX.2 standard (*ANSI/IEEE Std 1003.2, ISO/IEC 9945-2).*

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

The regular expression (RE) is constructed as follows:

The following rules determine one-character REs that match a *single* character:

Any character that is not a special character (to be defined) matches itself.

1. A backslash (\) followed by any special character matches the literal character itself; that is, this "escapes" the special character.
2. The "special characters" are:

   `+ * ?  . [  ] ^ $ ( ) { } | \`

3. The period (**.**) matches any character. *E.g.*, "*.umpty*" matches either "*Humpty*" or "*Dumpty*."
4. A set of characters enclosed in brackets (**[ ]**) is a one-character RE that matches any of the characters in that set. *E.g.*, "*[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:

5. Parentheses (**( )**) group parts of regular expressions together into subexpressions that can be treated as a single unit. For example, *(ha)+* matches one or more "ha"'s.

6. 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.

7. 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.

8. A question mark (**?**) is an optional element. The preceeding RE can occur zero or once in the string — no more. *E.g. xy?z* matches either *xyz* or *xz*.

9. The concatenation of REs is a RE that matches the corresponding concatenation of strings. *E.g.*, [A-Z][a-z]* matches any capitalized word.

10. The OR character (**|**) allows a choice between two regular expressions. For example, *jell(y | ies)* matches either "jelly" or "jellies".

11. Braces (**{ }**) are reserved for future use.

12. All or part of the regular expression can be "anchored" to either the beginning or end of the string being searched:

13. If the caret (**^**) is at the beginning of the (sub)expression, then the matched string must be at the beginning of the string being searched.

14. If the dollar sign (**$**) is at the end of the (sub)expression, then the matched string must be at the end of the string being searched.

Persistence None

Example

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

main(){
  RWCString aString("Hark! Hark! the lark");

  // A regular expression matching any lowercase word or end of a
  //word starting with "l":
     RWCRExpr re("l[a-z]*");

  cout << aString(re) << endl;  // Prints "lark"
}
```

Public Constructors

```
RWCRExpr(const char* pat);
RWCRExpr(const RWCString& pat);
```

Construct a regular expression from the pattern given by `pat`. The status of the results can be found by using member function `status()`.

```
RWCRExpr(const RWCRExpr& r);
```

Copy constructor. Uses value semantics — self will be a copy of `r`.

```
RWCRExpr();
```

Default constructor. You must assign a pattern to the regular expression before you use it.

Public Destructor

```
~RWCRExpr();
```

Destructor. Releases any allocated memory.

Assignment Operators

```
RWCRExpr&
operator=(const RWCRExpr& r);
```

Recompiles self to pattern found in `r`.

```
RWCRExpr&
operator=(const char* pat);
RWCRExpr&
operator=(const RWCString& 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 = NULL,
      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 that this member function is relatively clumsy to use — class *RWCString* offers a better interface to regular expression searches.

```
statusType
status() const;
```

Returns the status of the regular expression:

statusType	Meaning
`RWCRExpr::OK`	No errors
`RWCRExpr::NOT_SUPPORTED`	POSIX.2 feature not yet supported.
`RWCRExpr::NO_MATCH`	Tried to find a match but failed
`RWCRExpr::BAD_PATTERN`	Pattern was illegal

statusType	Meaning
`RWCRExpr::BAD_COLLATING_ELEMENT`	Invalid collating element referenced
`RWCRExpr::BAD_CHAR_CLASS_TYPE`	Invalid character class type referenced
`RWCRExpr::TRAILING_BACKSLASH`	Trailing \ in pattern
`RWCRExpr::UNMATCHED_BRACKET`	[] imbalance
`RWCRExpr::UNMATCHED_PARENTHESIS`	() imbalance
`RWCRExpr::UNMATCHED_BRACE`	{} imbalance
`RWCRExpr::BAD_BRACE`	Content of {} invalid.
`RWCRExpr::BAD_CHAR_RANGE`	Invalid endpoint in [a-z] expression
`RWCRExpr::OUT_OF_MEMORY`	Out of memory
`RWCRExpr::BAD_REPEAT`	?,* or + not preceded by valid regular expression

Synopsis

```
#include <rw/cstring.h>
RWCString a;
```

Description

Class *RW**CString*** 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.
- 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. Nevertheless, you should be aware of this and program defensively. In any case, class *RW**CString*** can handle embedded nulls.

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 *RW**CSubString*** supports substring extraction and modification operations.

Persistence

Simple

Example

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

main(){
  RWCString a("There is no joy in Beantown.");

  cout << a << endl << "becomes...." << endl;

  RWCRExpr re("[A-Z][a-z]*town");  // Any capitalized "town"
  a.replace(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.

`enum RWCString::scopeType { one, all }`

Used to specify whether regular expression `replace` replaces the first `one` substring matched by the regular expression or replaces `all` substrings matched by the regular expression.

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. *This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

`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` bytes 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 string's *capacity* (that is, the size it can grow to without resizing) is given by the parameter `ic`. We recommend creating an `RWSize_T` value from a numerical constant to pass into this constructor. While `RWSize_T` knows how to convert size_t's to itself, conforming compilers will chose the conversion to char instead.

`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 *RW**CString***'s data as a null terminated string. This data is owned by the *RW**CString*** and may not be deleted or changed. If the *RW**CString*** 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. *This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
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. *This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
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 byte. 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 exceptionof type *RW**BoundsErr*** will occur.

```
char&
operator()(size_t i);
char
operator()(size_t i) const;
```

Return the `i`th byte. 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 *RW**BoundsErr*** will occur.

```
RWCSubString
operator()(size_t start, size_t len);
const RWCSubString
operator()(size_t start, size_t len) const;
```

Substring operator. Returns an *RW**CSubString*** 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 *RW**BoundsErr*** will occur.

```
RWCSubString
operator()(const RWCRExpr& re, size_t start=0);
const RWCSubString
operator()(const RWCRExpr& re, size_t start=0) const;
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`.

Note that if you wish to use `operator()(const RWCRExpr&...)` you must instead use `match(const RWCRExpr&...)` described below. The reason for this is that we are presently retaining *RW**CRegexp*** but `operator(const RWCRExpr&...)` and `operator(const RWCRegexp)` are ambiguous in the case of `RWCString::operator("string").` In addition, operator`(const char *)` and operator(`size_t`) are ambiguous in the case of `RWCString::operator(0)`. *This function maybe incompatible with strings with embedded nulls. This function is incompatible with MBCS strings.*

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. *This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
RWCString&
append(const char* cs, size_t N);
```

Append a copy of the character string `cs` to self. Exactly `N` bytes 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(char c, size_t N);
```

Append `N` copies of the character `c` to self. 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` bytes 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 bytes 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 char* str) const;
int
collate(const RWCString& 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. Provided only on platforms that provide `::strcoll()`. *This function is incompatible with strings with embedded nulls.*

```
int
compareTo(const char* str, caseCompare = RWCString::exact)
const;
int
compareTo(const RWCString& str,
          caseCompare = RWCString::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`. If `caseCompare` is `RWCString::exact`, then this function works for all string types. *Otherwise, this function is incompatible with MBCS strings.* This *function is incompatible with* `const char*` *strings with embedded nulls. This function may be incompatible with* `const char*` *MBCS strings.*

```
RWBoolean
contains(const char* str, caseCompare = RWCString::exact)
         const;
RWBoolean
contains(const RWCString& cs,
         caseCompare = RWCString::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`. If `caseCompare` is `RWCString::exact`, then this function works for all string types. *Otherwise, this function is incompatible with MBCS strings.* This *function is incompatible with* `const char*` *strings with embedded nulls. This function may be incompatible with* `const char*` *MBCS strings.*

```
const char*
data() const;
```

Access to the *RW**CString***'s data as a null terminated string. This datum is owned by the *RW**CString*** and may not be deleted or changed. If the *RW**CString*** object itself changes or goes out of scope, the pointer value previously returned 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 or if there is an embedded null prior to finding `c`. *This function is incompatible with strings with embedded nulls. This function is incompatible with MBCS strings.*

```
size_t
first(char c, size_t) const;
```

Returns the index of the first occurence of the character `c` in self. Continues to search past embedded nulls. Returns `RW_NPOS` if there is no such character. *This function is incompatible with MBCS strings.*

```
size_t
first(const char* str) const;
```

Returns the index of the first occurence in self of any character in `str`. Returns `RW_NPOS` if there is no match or if there is an embedded null prior to finding any character from `str`. *This function is incompatible with strings with embedded nulls. This function may be incompatible with MBCS strings.*

```
size_t
first(const char* str, size_t N) const;
```

Returns the index of the first occurence in self of any character in `str`. Exactly `N` bytes in `str` are checked *including any embedded nulls* so `str` must point to a buffer containing at least `N` bytes. Returns `RW_NPOS` if there is no match.

```
unsigned
hash(caseCompare = RWCString::exact) const;
```

Returns a suitable hash value. *If* `caseCompare` *is* `RWCString::ignoreCase` *then this function will be incompatible with MBCS strings.*

```
size_t
index(const char* pat,size_t i=0,
      caseCompare = RWCString::exact) const;
size_t
index(const RWCString& pat,size_t i=0,
      caseCompare = RWCString::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`. If `caseCompare` is `RWCString::exact`, then this function works for all string types. *Otherwise, this function is incompatible with MBCS strings.*

```
size_t
index(const char* pat, size_t patlen,size_t i,
      caseCompare cmp) const;
size_t
index(const RWCString& pat, size_t patlen,size_t i,
      caseCompare cmp) const;
```

Pattern matching. Starting with index `i`, searches for the first occurrence of the first `patlen` bytes 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. If `caseCompare` is

`RWCString::exact`, then this function works for all string types. *Otherwise, this function is incompatible with MBCS strings.*

```
size_t
index(const RWCRExpr& re, size_t i=0) const;
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. *This function is incompatible with MBCS strings.*

```
size_t
index(const RWCRExpr& re,size_t* ext,size_t i=0) const;
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`. *This function is incompatible with strings with embedded nulls. This function may be incompatible with MBCS strings.*

```
RWCString&
insert(size_t pos, const char* cs);
```

Insert a copy of the null-terminated string `cs` into self at byte position `pos`, thus expanding the string. Returns a reference to self. *This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
RWCString&
insert(size_t pos, const char* cs, size_t N);
```

Insert a copy of the first `N` bytes of `cs` into self at byte position `pos`, thus expanding the string. Exactly `N` bytes 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&
insert(size_t pos, const RWCString& str);
```

Insert a copy of the string `str` into self at byte 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` bytes or the length of `str` (whichever is less) of `str` into self at byte position `pos`. Returns a reference to self.

```
RWBoolean
isAscii() const;
```

Returns `TRUE` if self contains no bytes 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 or if there is an embedded null to the right of `c` in self. *This function is incompatible with strings with embedded nulls. This function may be incompatible with MBCS strings.*

```
size_t
last(char c, size_t N) const;
```

Returns the index of the last occurrence in the string of the character `c`. Continues to search past embedded nulls. Returns `RW_NPOS` if there is no such character. *This function is incompatible with MBCS strings.*

```
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.*

```
RWCSubString
match(const RWCRExpr& re, size_t start=0);
const RWCSubString
match(const RWCRExpr& 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. Note that this is used in place of `operator()(const RWCRegexp&...)` if you want to use extended regular expressions.

```
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()` ≤ `length()`. Provided only on platforms that provide `::mblen()`.

```
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. *This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
RWCString&
prepend(const char* cs, size_t N);
```

Prepend a copy of the character string `cs` to self. Exactly `N` bytes 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(char c, size_t N);
```

Prepend `N` copies of character `c` to self. 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` bytes 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. If the number of bytes remaining in the stream is large, you should resize the ***RWCString*** to approximately the number of bytes to be read prior to using this method. See “Implementation Details” in the User’s Guide for more information. *This function is incompatible with strings with embedded nulls. This function may be incompatible with MBCS strings.*

```
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. *If* `delim` *is* `'\0'` *then this function is incompatible with strings with embedded nulls. If* `delim` *is* `'\0'` *then this function may be incompatible with MBCS strings.*

```
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()`. T*his function is incompatible with MBCS strings.*

```
RWCString&
remove(size_t pos);
```

Removes the bytes from the byte position `pos`, which must be no greater than `length()`, to the end of string. Returns a reference to self.

```
RWCString&
remove(size_t pos, size_t N);
```

Removes `N` bytes or to the end of string (whichever comes first) starting at the byte position `pos`, which must be no greater than `length()`. Returns a reference to self.

```
RWCString&
replace(size_t pos, size_t N, const char* cs);
```

Replaces `N` bytes or to the end of string (whichever comes first) starting at byte position `pos`, which must be no greater than `length()`, with a copy of the null-terminated string `cs`. Returns a reference to self. T*his function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
RWCString&
replace(size_t pos, size_t N1,const char* cs, size_t N2);
```

Replaces `N1` bytes or to the end of string (whichever comes first) starting at byte position `pos`, which must be no greater than `length()`, with a copy of the string `cs`. Exactly `N2` bytes 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` bytes or to the end of string (whichever comes first) starting at byte position `pos`, which must be no greater than `length()`, 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` bytes or to the end of string (whichever comes first) starting at position `pos`, which must be no greater than `length()`, with a copy of the first `N2` bytes, or the length of `str` (whichever is less), from `str`. Returns a reference to self.

```
replace(const RWCRExpr& pattern, const char* replacement,
        scopeType scope=one);
replace(const RWCRExpr& pattern,
        const RWCString& replacement,scopeType scope=one);
```

Replaces substring matched by `pattern` with replacement string. `pattern` is the new extended regular expression. `scope` is one of {`one`, `all`} and controls whether all matches of pattern are replaced with `replacement` or just the first one match is replaced. `replacement` is the replacement pattern for the string. Here's an example:

```
RWCString s("hahahohoheehee");
s.replace(RWCRExpr("(ho)+","HAR"); // s == "hahaHARheehee"
```

This function is incompatible with `const char* replacement` *strings with embedded nulls. This function may be incompatible with* `const char* replacement` *MBCS strings.*

```
void
resize(size_t n);
```

Changes the length of self to `n` bytes, adding blanks or truncating as necessary.

```
RWCSubString
strip(stripType s = RWCString::trailing, char c = ‘ ‘);
const RWCSubString
strip(stripType s = RWCString::trailing, char c = ‘ ‘)
const;
```

Returns a substring of self where the character `c` has been stripped off the beginning, end, or both ends of the string. The first variant can be used as an lvalue. 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 = RWCString::exact);
const RWCSubString
subString(const char* cs, size_t start=0,
          caseCompare = RWCString::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`. *If* `caseCompare` *is* `RWCString::ignoreCase` *then this function is incompatible with MBCS strings. This function is incompatible with* `cs` *strings with embedded nulls. This function may be incompatible with* `cs` *MBCS strings.*

```
void
toLower();
```

Changes all upper-case letters in self to lower-case, using the standard C library facilities declared in `<ctype.h>`. *This function is incompatible with MBCS strings.*

```
void
toUpper();
```

Changes all lower-case letters in self to upper-case, using the standard C library facilities declared in `<ctype.h>`. *This function is incompatible with MBCS strings.*

Static Public Member Functions

```
static unsigned
hash(const RWCString& str);
```

Returns the hash value of `str` as returned by `str.hash(RWCString::exact)`.

```
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 bytes. 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 bytes. If more than `mw` bytes 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 bytes.

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*. *This function is incompatible with* `const char*` *strings with embedded nulls. This function may be incompatible with* `const char*` *MBCS strings.*

```
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. *This function is incompatible with* `const char*` *strings with embedded nulls. This function may be incompatible with* `const char*` *MBCS strings.*

```
RWCString
operator+(const RWCString&, const RWCString&);
RWCString
operator+(const char*,      const RWCString&);
RWCString
operator+(const RWCString&, const char*     );
```

Concatenation operators. *This function is incompatible with* `const char*` *strings with embedded nulls. This function may be incompatible with* `const char*` *MBCS strings.*

```
ostream&
operator<<(ostream& s, const RWCString&);
```

Output an *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`. *This function is incompatible with MBCS strings.*

```
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()`. Provided only on platforms that provide `::strxfrm()`. *This function is incompatible with strings with embedded nulls.*

```
RWCString
toLower(const RWCString& str);
```

Returns a version of `str` where all upper-case characters have been replaced with lower-case characters. Uses the standard C library function `tolower()`. *This function is incompatible with MBCS strings.*

```
RWCString
toUpper(const RWCString& str);
```

Returns a version of `str` where all lower-case characters have been replaced with upper-case characters. Uses the standard C library function `toupper()`. *This function is incompatible with MBCS strings.*

Synopsis

```
#include <rw/cstring.h>
RWCString s("test string");
s(6,3);     // "tri"
```

Description

The class *RWCSubString* allows some subsection of an *RWCString* to be addressed by defining a *starting position* and an *extent*. For example the 7th through the 11th 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 — *RWCSubString*s are constructed by various functions of the *RWCString* class and then destroyed immediately.

A *zero length* 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.

Persistence

None

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

```
RWCSubString&
operator=(const RWCString&);
```

Assignment from an *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. Sets self's extent to be the length of the assigned *RW**CString.*** If self is the null substring, then the statement has no effect. Returns a reference to self.

```
RWCSubString&
operator=(const RWCSubString&);
```

Assignment from an *RW**CSubString***. The statements:

```
RWCString a;
RWCString b;
...
b(2, 3) = a(5,5);
```

will copy 5 characters of `a`'s data into the substring `b(2,3)`. The number of elements need not match: if they differ, `b` will be resized appropriately. Sets self's extent to be the extent of the assigned *RW**CSubString***. If self is the null substring, then the statement has no effect. Returns a reference to self.

```
RWCSubString&
operator=(const char*);
```

Assignment from a character string. Example:

```
RWCString str("Mary had a lamb");
char dat[] = "Perrier";
str(11,4) = dat;  // "Mary had a Perrier"
```

Note that the number of characters selected need not match: if they differ, `str` will be resized appropriately. Sets self's extent to be the `strlen()` of the assigned character string. If self is the null substring, then the statement has no effect. Returns a reference to self.

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 *RW**BoundsErr*** 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 *RW**BoundsErr*** 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 *RW**CSubString***.

```
RWBoolean
operator!() const;
```

Returns `TRUE` if this is a null substring.

```
size_t
start() const;
```

Returns the starting element of the *RW**CSubString***.

```
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 *RW**CString*** 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==()`.

Synopsis

```
#include <rw/ctoken.h>
RWCString str("a string of tokens");
RWCTokenizer(str);  // Lex the above string
```

Description

Class *RW**CTokenizer*** 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.

Persistence

None

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 Operators

`RWCSubString`
`operator()``;`

Advance to the next token and return it as a substring. The tokens are delimited by any of the four characters in `" \t\n\0"`. (space, tab, newline and null).

`RWCSubString`
`operator()``(const char* s);`

Advance to the next token and return it as a substring. The tokens are delimited by any character in `s`, or any embedded null.

```
RWCSubString
operator()(const char* s,size_t num);
```

Advance to the next token and return it as a substring. The tokens are delimited by any of the first num characters in s. Buffer s may contain nulls, and must contain at least num characters. Tokens will not be delimited by nulls unless s contains nulls.

Synopsis

```
#include <rw/rwdate.h>RWDate a;   // Construct today's date
```

Description

Class *RW**Date*** represents a date, stored as a Julian day number. The member function `isValid()` can be used to determine whether an *RW**Date*** is a valid date. For example, `isValid()` would return `FALSE` for the date 29 February 1991 because 1991 is not a leap year. See "Using Class *RW**Date***" in the *Tools.h++ User's Guide*.

*RW**Date***'s can be converted to and from *RW**Time***'s, and to and from the Standard C library type ***struct tm*** defined in `<time.h>`.

Note that using a 2-digit year specifier in your code may lead to less-than-perfect behavior at the turn of the century. We urge you to create programs that are "millenially correct" by using 4-digit year specifiers.

Note that because the default constructor for this class creates an instance holding the current date, constructing a large array of *RW**Date*** may be slow.

```
RWDate v[5000];     // Figures out the current date 5000 times
```

Those with access to the Standard C++ Library-based versions of the *Tools.h++* template collections should consider the following:

```
// Figures out the current date just once:
RWTValOrderedVector<RWDate> v(5000, RWDate());
```

Thanks to the smart allocation scheme of the standard collections, the above declaration will result in only one call to the default constructor followed by 5000 invocations of the copy constructor. In the case of *RW**Date,*** the copy constructor amounts to an assignment of one `long` to another, resulting in faster creation than the simple array.

Persistence

Simple

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 *RW**Date*** with the present date.

```
RWDate(const RWDate&);
```

Copy constructor.

```
RWDate(unsigned day, unsigned year);
```

Constructs an *RW**Date*** 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 *RW**Date*** 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 *RW**Date*** 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. 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 *RW**Locale*** 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 *RW**Locale*** 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 *RW**Date*** from an *RW**Time***. 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 *RW**Date*** 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 *RW**Date*** and *RW**Time*** 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`. Note that it is possible to construct a valid *RW**Date*** which represents a day previous to the beginning of the Gregorian calendar for some locality. Rogue Wave doesn't know the specifics for your locality, so will not enforce an arbitrary cutoff for "validity."

Public Member Operators

RWDate&
operator=(const RWDate&);

Assignment operator.

RWDate
operator++();

Prefix increment operator. Adds one day to self, then return the result.

RWDate
operator--();

Prefix decrement operator. Subtracts one day from self, then returns the result.

RWDate
operator++(int);

Postfix increment operator. Adds one day to self, returning the initial value.

RWDate
operator--(int);

Postfix decrement operator. Subtracts one day from self, returning the initial value.

RWDate&
operator+=(unsigned long s);

Adds `s` days to self, returning self.

```
RWDate&
operator-=(unsigned long s);
```
Substracts 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 in the standard C library function `strftime()`.

```
RWCString
asString(const char* format,
         const RWLocale& = RWLocale::global()) const;
```
Returns the date as a string, formatted by the *RWLocale* argument. Formats are as defined in 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 that the encoding for

months and days of the week used in ***struct tm*** differs from that used elsewhere in *RW**Date***. If the date is invalid, all fields are set to -1.

```
unsigned
firstDayOfMonth() const;
```

Returns the day of the year (1-366) corresponding to the first day of this *RW**Date***'s month and year.

```
unsigned
firstDayOfMonth(unsigned month) const;
```

Returns the day of the year (1-366) corresponding to the first day of the month `month` (1–12) in this *RW**Date***'s year.

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

Returns a suitable hashing value.

```
RWBoolean
isValid() const;
```

Returns `TRUE` if this is a valid date, `FALSE` otherwise.

The following two functions are provided as a service to users who need to manipulate the date representation directly. *The julian day number is not the Julian date!*. The julian day number is calculated using Algorithm 199 from *Communications of the ACM*, Volume 6, No. 8, (Aug. 1963), p. 444 and is valid for any valid Gregorian date in the Gregorian calendar. The Gregorian calendar was first introduced on Sep. 14, 1752, and was adopted at various times in various places.

```
unsigned long
julian() const;
```

Returns the value of the julian day number..

```
void
julian(unsigned long j);
```

Changes the value of the julian day number to j.

```
RWBoolean
leap() const;
```

Returns `TRUE` if the year of this *RW**Date*** 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 *RW**Locale*** argument.

```
RWDate
next(unsigned dayNum) const;
```
Returns the date of the next numbered day of the week, where *Monday* = 1, ..., *Sunday* = 7. The variable `dayNum` must be between 1 and 7, inclusive.

```
RWDate
next(const char* dayName,
     const RWLocale& = RWLocale::global()) const;
```
Returns the date of the next `dayName` (for example, the date of the previous Monday) The weekday name is interpreted according to the *RW**Locale*** argument.

```
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 *RW**Locale*** argument.

```
RWCString
weekDayName(const RWLocale& = RWLocale::global()) const;
```
Returns the name of the day of the week for this date, according to the optional *RW**Locale*** 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 Public 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 *RW***Locale** argument. Returns 0 if no match is found.

```
static unsigned
daysInMonthYear(unsigned month, unsigned year);
```

Returns the number of days in a given month and year. Returns 0 if `month` is not between 1 and 12 inclusive.

```
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
hash(const RWDate& d);
```

Returns the hash value of `d` as returned by `d.hash()`.

```
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 RWDate
now();
```

Returns today's date.

```
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, unsigned long s);
RWDate
operator+(unsigned long 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);
```
If d1>d2, returns the number of days between `d1` and `d2`. Otherwise, the result is implementation defined.

```
RWDate
operator-(const RWDate& d, unsigned long 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 *RW**Locale***), 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 *RW**File***, respectively.

```
RWvistream&
operator>>(RWvistream&, RWDate& date);
RWFile&
operator>>(RWFile&,     RWDate& date);
```

Restores the date into `date` from a virtual stream or *RW**File***, respectively, replacing the previous contents of `date`.

*RW**DDEstreambuf*** ⟶ *RW**CLIPstreambuf*** ⟶ ***streambuf***

Synopsis

```
#include <rw/winstrea.h>
#include <iostream.h>
iostream str( new RWDDEstreambuf(CF_TEXT, TRUE, TRUE, TRUE) ) ;
```

Description

Class *RW**DDEstreambuf*** is a specialized ***streambuf*** that gets and puts sequences of characters to Microsoft Windows global memory that has been allocated with the `GMEM_DDESHARE` flag. It can be used to exchange data through the Windows *Dynamic Data Exchange* (DDE) facility.

The class has two modes of operation: dynamic and static. In dynamic mode, memory is allocated and reallocated on an as-needed basis. If too many characters are inserted into the internal buffer for its present size, then it will be resized and old characters copied over into any new memory as necessary. This is transparent to the user. It is expected that this mode would be used primarily by the DDE server. In static mode, the buffer streambuf is constructed from a specific piece of memory. No reallocations will be done. It is expected that this mode would be used primarily by the DDE client.

In dynamic mode, the *RW**DDEstreambuf*** "owns" any allocated memory until the member function `str()` is called, which "freezes" the buffer and returns an unlocked Windows handle to it. The effect of any further insertions is undefined. Until `str()` has been called, it is the responsibility of the *RW**DDEstreambuf*** destructor to free any allocated memory. After the call to `str()`, it becomes the user's responsibility.

In static mode, the user always has the responsibility for freeing the memory handle. However, because the constructor locks and dereferences the handle, you should not free the memory until either the destructor or `str()` has been called, either of which will unlock the handle.

Note that although the user may have the "responsibility" for freeing the memory, whether it is the client or the server that actually does the call to `GlobalFree()` will depend on the DDE "release" flag.

Persistence

None

Example

This is an example of how the class might be used by a DDE server.

```
#include <rw/winstrea.h>
#include <iostream.h>
#include <windows.h>
```

```
#include <dde.h>

BOOL
postToDDE(HWND hwndServer, HWND hwndClient) {
  RWDDEstreambuf* buf =
  new RWDDEstreambuf(CF_TEXT, TRUE, TRUE, TRUE);
  ostream ostr(buf);
  double d = 12.34;
  ostr << "Some text to be exchanged through the DDE.\n";
  ostr << "The double you requested is: " << d << endl;
  ostr.put(0);// Include the terminating null
  // Lock the streambuf, get its handle:
  HANDLE hMem = buf->str();
  // Get an identifying atom:
  ATOM aItem = GlobalAddAtom("YourData");
  if(!PostMessage(hwndClient, WM_DDE_DATA, hwndServer,
                  MAKELONG(hMem, aItem))){
    // Whoops!  The message post failed, perhaps because
    // the client terminated.  Now we are responsible
    // for deallocating the memory:
   if( hMem != NULL )
   GlobalFree(hMem);
   GlobalDeleteAtom(aItem);
   return FALSE;
  }
  return TRUE;
}
```

The handle of the DDE server is passed in as parameter `hwndServer`, the handle of the client as parameter `hwndClient`. An ***ostream*** is created, using an *RW**DDEstreambuf*** as its associated ***streambuf***. The results can be used much like any other ***ostream***, such as ***cout***, except that characters will be inserted into Windows global memory, from where they can be transferred through the DDE. Note the parameters used in the constructor. These should be studied below as they have important ramifications on how memory allocations are handled through the DDE. In particular, parameter `fRelease`, if `TRUE`, states that the *client* will be responsible for deallocating the memory when done. The defaults also specify `fAckReq TRUE`, meaning that the client will acknowledge receiving the message: you must be prepared to receive it.

Some text and a double is inserted into the ***ostream***. Member function `str()` is then called which unlocks and returns a Windows `HANDLE`. Once we have called `str()`, we are responsible for this memory and must either free it when done, or pass on that responsibility to someone else. In this case, it will be passed on to the client.

An atom is then constructed to identify the data. The DDE data, along with its identifying atom, is then posted. If the post fails, then we have been unable to foist our responsbility for the global memory onto someone else and will have to free it (along with the atom) ourselves.

Public Constructors

```
RWDDEstreambuf(WORD cfFormat   = CF_TEXT,
               BOOL fResponse  = TRUE
               BOOL fAckReq    = TRUE
               BOOL fRelease   = TRUE);
```

Constructs an empty *RW**DDEstreambuf*** in dynamic mode. The results can be used anywhere any other ***streambuf*** can be used. Memory to accomodate new characters will be allocated as needed.

The four parameters are as defined by the *Windows Reference, Volume 2* (in particular, see the section *DDE Message Directory*). Parameter `cfFormat` specifies the format of the data being inserted into the ***streambuf***. These formats are the same as used by `SetClipboardData()`. If a specializing virtual streams class such as *RW**bostream*** or *RW**postream*** is used to perform the actual character insertions instead of a simple ***ostream***, the format may not be so simple. In this case, the user might want to register his or her own format, using the Windows function `RegisterClipboardFormat()`.

For the meaning of the other three parameters see below, and / or the *Windows* reference manuals.

```
RWDDEstreambuf(HANDLE hMem);
```

Constructs an *RW**DDEstreambuf*** in static mode, using the memory block with global handle `hMem`. The effect of gets and puts beyond the size of this block is unspecified. The format of the DDE transfer, and the specifics of DDE acknowledgments, memory allocations, *etc.*, can be obtained by using the member functions defined below.

Public Destructor

```
~RWDDEstreambuf();
```

If member function `str()` has not been called, the destructor unlocks the handle and, if in dynamic mode, also frees it.

Public Member Functions

Because *RW**DDEstreambuf*** inherits from ***streambuf***, any of the latter's member functions can be used. Furthermore, *RW**DDEstreambuf*** has been designed to be analogous to ***streambuf***. However, note that the return type of `str()` is a `HANDLE`, rather than a `char*`.

```
BOOL
ackReq() const;
```

Returns whether this DDE exchange requests an acknowledgement. See the *Windows Reference, Volume 2*, for more information.

```
WORD
format() const;
```

Returns the format of this DDE exchange (*e.g.*, `CF_TEXT` for text exchange, *etc.*). See the *Windows Reference, Volume 2*, for more information.

```
BOOL
release() const;
```

Returns `TRUE` if the client is responsible for the release of of the memory returned by `str()`. See the *Windows Reference, Volume 2*, for more information.

```
BOOL
response() const;
```

Returns `TRUE` if this data is in response to a `WM_DDE_REQUEST` message. Otherwise, it is in response to a `WM_DDE_ADVISE` message. See the *Windows Reference, Volume 2*, for more information.

```
HANDLE
str();
```

Returns an (unlocked) `HANDLE` to the global memory being used. The *RW**DDEstreambuf*** should now be regarded as "frozen": the effect of inserting any more characters is undefined. If the *RW**DDEstreambuf*** was constructed in dynamic mode, and nothing has been inserted, then the returned `HANDLE` may be `NULL`. If it was constructed in static mode, then the returned handle will be the handle used to construct the *RW**DDEstreambuf***.

*RW**DiskPageHeap*** → *RW**BufferedPageHeap*** → *RW**VirtualPageHeap***

Synopsis

```
#include <rw/diskpage.h>
unsigned nbufs;
unsigned pagesize;
RWDiskPageHeap heap("filename", nbufs, pagesize) ;
```

Description

Class *RW**DiskPageHeap*** is a specializing type of buffered page heap. It swaps its pages to disk as necessary.

Persistence

None

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.

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

struct Node {
  int key;
  RWHandle next;
};

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<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);
  }

// 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 *RW**DiskPageHeap*** has been constructed, call member function `isValid()`.

Public Destructor

```
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 *RW**VirtualPageHeap***. 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 *RW**BufferedPageHeap***. Deallocate the page associated with handle `h`. It is not an error to deallocate a zero handle.

```
virtual void
dirty(RWHandle h);
```

Inherited from *RW**BufferedPageHeap***.

```
RWBoolean
isValid() const;
```

Returns `TRUE` if this is a valid *RW**DiskPageHeap***.

```
virtual void*
lock(RWHandle h);
```

Inherited from *RW**BufferedPageHeap***.

```
virtual void
unlock(RWHandle h);
```

Inherited from *RW**BufferedPageHeap***.

RWDlistCollectables → *RWSequenceable* → *RWCollection* → *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.

Persistence

Polymorphic

Public Constructors

`RWDlistCollectables();`
Constructs an empty doubly-linked list.

`RWDlistCollectables (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 *RW**BoundsErr*** will occur. Note that for a linked list, these functions must traverse all the links, making them not particularly efficient.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
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 *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Sequenceable***. 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 *RW**Collectable***.

```
virtual size_t
index(const RWCollectable* c) const;
```
Redefined from class *RW**Sequenceable***. 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 *RW**Collection***. 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 *RW**Sequenceable***. 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 *RW**BoundsErr*** will occur.

```
virtual RWClassID
isA() const;
```
Redefined from class *RW**Collectable*** to return `__RWDLISTCOLLECTABLES`.

```
virtual RWBoolean
isEmpty() const;
```
Redefined from class *RW**Collection***.

```
virtual RWCollectable*
last() const;
```
Redefined from class *RW**Sequenceable***. Returns the item at the end of the list.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```
Redefined from class *RW**Collection***. 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 *RW**Sequenceable***. 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 *RW**Collection***. 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 *RW**Collection***.

```
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 *RW**Collection***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW***DlistCollectablesIterator** → *RW***Iterator**

Synopsis

```
#include <rw/dlistcol.h>
RWDlistCollectables d;
RWDlistCollectablesIterator it(d) ;
```

Description

Iterator for class *RW***DlistCollectables**. Traverses the linked-list from the first (head) to the last (tail) item. Functions are provided for moving in *either* direction.

As with all Rogue Wave 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.

Persistence

None

Public Constructor

`RWDlistCollectablesIterator` `(RWDlistCollectables& d);`
Construct an *RW***DlistCollectablesIterator** from an *RW***DlistCollectables**. Immediately after construction, the position of the iterator is undefined.

Public Member Operators

`virtual RWCollectable*`
`operator()``();`
Redefined from class *RW***Iterator**. 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 *RW**Iterator***. 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.

*RW***eistream** → *RW***bistream** ... → ***ios***

... → *RW***vistream** → *RW***vios**

Synopsis

```
#include <rw/estream.h>
RWeistream estr(cin);          // Construct an RWeistream,
                               // using cin's streambuf
```

Description

Class *RW***eistream** specializes the base class *RW***bistream** to restore values previously stored by *RW***eostream**.The endian streams, *RW***eistream** and *RW***eostream**, offer an efficient compromise between the portable streams (*RW***pistream**, *RW***postream**) and the binary streams (*RW***bistream**, *RW***bostream**). By compensating for differences in big-endian vs. little-endian formats, as well as sizes of the various integral types, the endian streams offer portability without incurring the stream-size overhead of translating values into a series of printable characters. For example, data stored in little-endian format by an *RW***eostream** object in a DOS program can be retrieved by an *RW***eistream** object on any of several machines, regardless of its native endian format or the sizes of its integral types. Endian streams will work properly when shared among a group of platforms that:

- Share a common size and representation (apart from endian format) for types `float` and `double`;
- Use two's complement format for negative integral values.

As with the portable streams, care must be taken when storing or retrieving variables of type `char`. Endian stream methods treat `char`s as numbers except where the method description explicitly states that the `char` is being treated, instead, as a character. See the entry for *RW***postream** for an example of this distinction. Data stored in an integral type on one platform may be too large to fit into that type on a receiving platform. If so, the *RW***eistream**'s failbit will be set.

Endian streams can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, *etc.*

Persistence

None.

Example

See *RW***eostream** for an example of how the file “`data.dat`” might be created.

```
#include <rw/estream.h>
#include <fstream.h>
main()
{
 ifstream fstr("data.dat");     // Open an input file
 RWeistream estr(fstr);        // Construct an RWeistream from it
                               // (For DOS: RWeistream estr(fstr, ios::binary)
 int i;
 float f;
 double d;

 estr >> i;                    // Restore an int that was stored in binary,
                              //  without regard to size or endian format.
 estr >> f >> d;              //  Restore a float & double without regard to
                              //  endian formats.
}
```

Public Constructors

RWeistream(streambuf* s);

Construct an *RW**eistream*** from the ***streambuf*** s. For DOS, this ***streambuf*** must have been created in binary mode. Throw exception *RW**StreamErr*** if not a valid endian stream.

RWeistream(istream& str);

Construct an *RW**eistream*** using the ***streambuf*** associated with the ***istream*** str. For DOS, the str must have been opened in binary mode. Throw exception *RW**StreamErr*** if not a valid endian stream.

Public Member Functions

virtual int
get();
virtual RWvistream&
get(char& c);
virtual RWvistream&
get(unsigned char& c);
virtual RWvistream&
get(char* v, size_t N);
virtual RWvistream&
get(unsigned char* v, size_t N);

Inherited from class *RW**bistream***.

virtual RWvistream&
get(wchar_t& wc);

Redefined from class *RW**bistream***. Get the next wchar_t from the input stream and store it in wc, compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in wc.

virtual RWvistream&
get(wchar_t* v, size_t N);

Redefined from class *RW**bistream***. Get a vector of wchar_ts and store it in the array beginning at v, compensating for any differences in size or endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
get(double* v, size_t N);
```

Redefined from class *RW**bistream***. Get a vector of `double`s and store them in the array beginning at `v`, compensating for any difference in endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit.

```
virtual RWvistream&
get(float* v, size_t N);
```

Redefined from class *RW**bistream***. Get a vector of `float`s and store them in the array beginning at `v`, compensating for any difference in endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit.

```
virtual RWvistream&
get(int* v, size_t N);
```

Redefined from class *RW**bistream***. Get a vector of `int`s and store them in the array beginning at `v`, compensating for any differences in size or endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
get(long* v, size_t N);
```

Redefined from class *RW**bistream***. Get a vector of `long`s and store them in the array beginning at `v`, compensating for any differences in size or endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
get(short* v, size_t N);
```

Redefined from class *RW**bistream***. Get a vector of `short`s and store them in the array beginning at `v`, compensating for any differences in size or endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
```
get`(unsigned short* v, size_t N);`

Redefined from class *RW**bistream***. Get a vector of `unsigned short`s and store them in the array beginning at `v`. If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
```
get`(unsigned int* v, size_t N);`

Redefined from class *RW**bistream***. Get a vector of `unsigned int`s and store them in the array beginning at `v`, compensating for any differences in size or endian format between the stream and the current environment. If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
```
get`(unsigned long* v, size_t N);`

Redefined from class *RW**bistream***. Get a vector of `unsigned long`s and store them in the array beginning at `v`, compensating for any differences in size or endian format between the stream and the current environment If the restore stops prematurely, store whatever possible in `v`, and set the failbit. Also set the failbit if any values in the stream are too large to be stored in an element of `v`.

```
virtual RWvistream&
```
getString`(char* s, size_t N);`

Redefined from class *RW**bistream***. 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, and the remaining characters of the string will be extracted from the stream and thrown away. In either case, the string will be terminated with a null byte. If the size of the string is too large to be represented by a variable of type `size_t` in the current environment, the badbit of the stream will be set, and no characters will be extracted. Note that the elements of the string are treated as characters, not numbers.

```
virtual RWvistream&
```
operator>>`(char& c);`

Redefined from class *RW**bistream***. Get the next `char` from the input stream and store it in `c`. Note that `c` is treated as a character, not a number.

```
virtual RWvistream&
operator>>(wchar_t& wc);
```

Redefined from class *RW**bistream***. Get the next `wchar_t` from the input stream and store it in `wc,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `wc`.

```
virtual RWvistream&
operator>>(double& d);
```

Redefined from class *RW**bistream***. Get the next `double` from the input stream and store it in `d,` compensating for any difference in endian format between the stream and the current environment.

```
virtual RWvistream&
operator>>(float& f);
```

Redefined from class *RW**bistream***. Get the next `float` from the input stream and store it in `f,` compensating for any difference in endian format between the stream and the current environment.

```
virtual RWvistream&
operator>>(int& i);
```

Redefined from class *RW**bistream***. Get the next `int` from the input stream and store it in `i,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `i`.

```
virtual RWvistream&
operator>>(long& l);
```

Redefined from class *RW**bistream***. Get the next `long` from the input stream and store it in `l,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `l`.

```
virtual RWvistream&
operator>>(short& s);
```

Redefined from class *RW**bistream***. Get the next `short` from the input stream and store it in `s,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `s`.

```
virtual RWvistream&
operator>>(unsigned char& c);
```

Redefined from class *RW**bistream***. Get the next `unsigned char` from the input stream and store it in `c`. Note that `c` is treated as a character, not a number.

`virtual RWvistream&`
`operator>>``(unsigned short& s);`

Redefined from class *RW**bistream***. Get the next `unsigned short` from the input stream and store it in `s,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `s`.

`virtual RWvistream&`
`operator>>``(unsigned int& i);`

Redefined from class *RW**bistream***. Get the next `unsigned int` from the input stream and store it in `i,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `i`.

`virtual RWvistream&`
`operator>>``(unsigned long& l);`

Redefined from class *RW**bistream***. Get the next `unsigned long` from the input stream and store it in `l,` compensating for any differences in size or endian format between the stream and the current environment. Set the failbit if the value in the stream is too large to be stored in `l`.

`RWeostream::EndianstreamEndian``();`

Return the endian format (`RWeostream::BigEndian` or `RWeostream::LittleEndian`) of numeric values, as represented in the stream.

`size_t`
`streamSizeofInt``();`

Return the size of `int`s, as represented in the stream.

`size_t`
`streamSizeofLong``();`

Return the size of `long`s, as represented in the stream.

`size_t`
`streamSizeofShort``();`

Return the size of `short`s, as represented in the stream.

`size_t`
`streamSizeofSizeT``();`

Return the size of `size_t`s, as represented in the stream.

`size_t`
`streamSizeofWchar``();`

Returns the size of `wchar_t`s, as represented in the stream.

. . . → *ios*

*RW**eostream*** . . .

. . . → *RW**bostream*** → *RW**vostream*** → *RW**vios***

Synopsis

```
#include <rw/estream.h>
// Construct an RWeostream that uses cout's streambuf,
//   and writes out values in little-endian format:
RWeostream estr(cout, RWeostream::LittleEndian);
```

Description

Class *RW**eostream*** specializes the base class *RW**bostream*** to store values in a portable binary format. The results can be restored via its counterpart, *RW**eistream***.See the entry for *RW**eistream*** for a general description of the endian stream classes.

Persistence

None.

Example

See *RW**eistream*** for an example of how the file "`data.dat`" might be read.

```
#include <rw/estream.h>
#include <fstream.h>

main()
{
 ofstream fstr("data.dat");                              // Open an output file
 RWeostream estr(fstr);                     // Construct an RWeostream from it
                                   // (For DOS: RWeistream estr(fstr, ios::binary)
 int i = 5;
 float f = 22.1;
 double d = -0.05;

 estr << i;                                 // Store an int, float, and double
 estr << f << d;                            //  using the native endian format
}
```

Enumeration

```
enum RWeostream::Endian { LittleEndian,
                          BigEndian,
                          HostEndian }
```

Used to specify the format that *RW**eostream***s should use to represent numeric values in the stream. `HostEndian` means to use the native format of the current environment.

Public Constructors

```
RWeostream(streambuf* s, Endian fmt = HostEndian);
```

Construct an *RW**eostream*** from the ***streambuf*** `s`. Values placed into the stream will have an endian format given by `fmt`. For DOS, the ***streambuf*** must have been created in binary mode. Throw exception *RW**StreamErr*** if ***streambuf*** `s` is not empty.

RWeostream(ostream& str, Endian fmt = HostEndian);

Construct an *RW**eostream*** from the ***streambuf*** associated with the output stream `str`. Values placed into the stream will have an endian format given by `fmt`. For DOS, the `str` must have been opened in binary mode. Throw exception *RW**StreamErr*** if ***streambuf*** `s` is not empty.

Public Destructor

virtual **~RWvostream**();

This virtual destructor allows specializing classes to deallocate any resources that they may have allocated.

Public Member Functions

virtual RWvostream&
flush();

Send the contents of the stream buffer to output immediately.

virtual RWvostream&
operator<<(const char* s);

Redefined from class *RW**bostream***. Store the character string starting at `s` to the output stream. The character string is expected to be null terminated. Note that the elements of s are treated as characters, not as numbers.

virtual RWvostream&
operator<<(char c);

Redefined from class *RW**bostream***. Store the `char c` to the output stream. Note that `c` is treated as a character, not a number.

virtual RWvostream&
operator<<(wchar_t wc);

Redefined from class RWbostream. Store the `wchar_t wc` to the output stream in binary, using the appropriate endian representation.

virtual RWvostream&
operator<<(unsigned char c);

Redefined from class *RW**bostream***. Store the `unsigned char c` to the output stream. Note that `c` is treated as a character, not a number.

virtual RWvostream&
operator<<(double d);

Redefined from class *RW**bostream***. Store the `double d` to the output stream in binary, using the appropriate endian representation.

virtual RWvostream&
operator<<(float f);

Redefined from class *RW**bostream***. Store the `float f` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
operator<<(int i);
```

Redefined from class *RWbostream*. Store the `int i` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
operator<<(unsigned int i);
```

Redefined from class *RWbostream*. Store the `unsigned int i` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
operator<<(long l);
```

Redefined from class *RWbostream*. Store the `long l` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
operator<<(unsigned long l);
```

Redefined from class *RWbostream*. Store the `unsigned long l` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
operator<<(short s);
```

Redefined from class *RWbostream*. Store the `short s` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
operator<<(unsigned short s);
```

Redefined from class *RWbostream*. Store the `unsigned short s` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(char c);
virtual RWvostream&
put(unsigned char c);
virtual RWvostream&
put(const char* p, size_t N);
```

Inherited from class *RWbostream*.

```
virtual RWvostream&
put(wchar_t wc);
```

Redefined from class *RWbostream*. Store the w`char_t wc` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const wchar_t* p, size_t N);
```

Redefined from class *RWbostream*. Store the vector of `wchar_t`s starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const unsigned char* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `unsigned chars` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const short* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `shorts` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const unsigned short* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `unsigned shorts` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const int* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `ints` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const unsigned int* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `unsigned ints` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const long* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `longs` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const unsigned long* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `unsigned longs` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const float* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `floats` starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
put(const double* p, size_t N);
```

Redefined from class *RW**bostream***. Store the vector of `double`s starting at `p` to the output stream in binary, using the appropriate endian representation.

```
virtual RWvostream&
putString(const char*s, size_t N);
```

Store the character string, *including embedded nulls*, starting at s to the output string.

Synopsis

```
typedef unsigned short  RWClassID;
typedef RWCollectable*  (*RWuserCreator)();
#include <rw/factory.h>

RWFactory* theFactory;
```

Description

Class *RW**Factory*** can create an instance of an *RW**Collectable*** object, given a class ID. It does this by maintaining a table of class IDs and associated "creator functions." 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 *RW**Factory*** which can be accessed using `getRWFactory`. It is guaranteed to have creator functions in it for all of the classes referenced by your program. See also the section in the User's Guide about *RW**Factory***.

Persistence

None

Example

```
#include <rw/factory.h>
#include <rw/rwbag.h>
#include <rw/colldate.h>
#include <rw/rstream.h>

main(){
 // Create new RWBag off the heap, using Class ID __RWBAG.

 RWBag* b = (RWBag*)getRWFactory ()->create(__RWBAG);

 b->insert( new RWCollectableDate ); // Insert today's date
 // ...
 b->clearAndDestroy();     // Cleanup: first delete members,
 delete b;                 // then the bag itself
}
END FILE
```

Public Constructors

`RWFactory``();`

Construct an *RW**Factory***.

Public Operator

```
RWBoolean
operator<=(const RWFactory& h);
```

Returns `TRUE` if self is a subset of `h`, that is, every element of self has a counterpart in `h` which `isEqual`. This operator is included to fix an inconsistency in the C++ language. It is not explicitly present unless you are compiling with an implementation of the Standard C++ Library. It would normally be inherited from *RWSet*

Note: If you inherit from *RWFactory* in the presence of the Standard C++ Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

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`.

```
void
addFunction(RWuserCreator uc, RWClassID id, RWStringID sid);
```

Adds to the *RWFactory* the global function pointed to by `uc`, which creates an instance of an object with `RWClassID id` and `RWStringID sid`.

```
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.

```
RWCollectable*
create(RWString sid) const;
```

Allocates a new instance of the class with `RWStringID sid` off the heap and returns a pointer to it. Returns `nil` if `sid` 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.

```
RWuserCreator
getFunction(RWStringID sid) const;
```

Returns from the *RWFactory* a pointer to the global function associated with `RWStringID sid`. Returns `nil` if `sid` does not exist.

`void`
`removeFunction``(RWClassID id);`

Removes from the *RW***Factory** the global function associated with `RWClassID id`. If `id` does not exist in the factory, no action is taken.

`void`
`removeFunction``(RWStringID sid);`

Removes from the *RW***Factory** the global function associated with `RWStringID sid`. If `sid` does not exist in the factory, no action is taken.

`RWStringID`
`stringID``(RWClassID id) const;`

Looks up the *RW***StringID** associated with `id` and returns it. If there is no such association, returns `RWStringID("NoID")`.

`RWClassID`
`classID``(RWStringID) const;`

Looks up the *RW***ClassID** associated with `sid` and returns it. If there is no such association, returns `__RWUNKNOWN`.

Synopsis

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

RWFile f("filename");
```

Description

Class *RW**File*** 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).

Persistence

None

Public Constructors

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

Construct an *RW**File*** 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

`const char*`
`Access``();`

Returns the access mode with which the underlying `FILE*` was opened.

`void`
`ClearErr``();`

Reset error state so that neither `Eof()` nor `Error()` returns `TRUE`. Calls C library function `clearerr()`.

`RWoffset`
`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 as determined by a call to the C library function `ferror()`.

`RWBoolean`
`Exists``();`
Returns `TRUE` if the file exists.

`RWBoolean`
`Flush``();`
Perform any pending I/O operations. Returns `TRUE` if successful.

`const char*`
`GetName``();`
Returns the file name.

`FILE*`
`GetStream``();`
Returns the `FILE*` that underlies the *RW**File*** interface. Provided for users who need to “get under the hood” for system-dependent inquiries, etc. *Do not use to alter the state of the file!*

`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 short* i,size_t count);
RWBoolean
Read(unsigned int* i,  size_t count);
RWBoolean
Read(unsigned long* 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 that 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 that you are responsible for declaring `string` and for allocating the necessary storage before calling this function. Beware of overflow when using this function.

```
RWBoolean
SeekTo(RWoffset 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 short* i,size_t count);
RWBoolean
Write(const unsigned int* i,  size_t count);
RWBoolean
Write(const unsigned long* 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. Beware of non-terminated strings when using this function.

Static Public Member Functions

```
static RWBoolean
Exists(const char* filename, int mode = F_OK);
```

Returns `TRUE` if a file with name `filename` exists and may be accessed according to the `mode` specified. The `mode` may be `OR`ed together from one or more of:

- `F_OK`: "Exists" (Implied by any of the others)
- `X_OK`: "Executable or searchable"
- `W_OK`: "Writable"
- `R_OK`: "Readable"

If your compiler or operating system does not support the POSIX `access()` function, then mode `X_OK` will always return `FALSE`.

RW**FileManager** → RW**File**

Synopsis

```
typedef long      RWoffset ;
typedef unsigned long   RWspace;  // (typically)
#include <rw/filemgr.h>
RWFileManager f("file.dat");
```

Description

Class *RW**FileManager*** 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 *RW**FileManager*** inherits class *RW**File*** as a public base class; hence all the public member functions of *RW**File*** are visible to *RW**FileManager***. They *are not* listed here.

If a file is managed by an *RW**FileManager*** then reading or writing to unallocated space in the file will have undefined results. In particular, overwriting the end of allocated space is a common problem which usually results in corrupted data. One way to encounter this problem is to use `binaryStoreSize()` to discover the amount of space needed to store an *RW**Collection***. For most purposes, the storage size of an *RW**Collection*** is found using the *RW**Collectable*** method `recursiveStoreSize()`.

Persistence

None

Public Constructor

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

Constructs an *RW**FileManager*** for the file with path name `filename` using 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+`"). If the file exists and is not empty, then the constructor assumes it contains an existing file manager; other contents will cause an exception of type *RW**ExternalErr*** to be thrown. If no file exists or if an existing file is empty, then the constructor will attempt to create the file (if necessary) and initialize it with a new file manager. The resultant object should be checked for validity using function `isValid()`. A possible exception that could occur is *RW**FileErr***.

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 an offset just past the end of the file.

```
RWoffset
start();
```

Returns the offset of the first space ever allocated for data in this file. If no space has ever been allocated, returns `RWNIL`. This is typically used to "get started" and find the rest of the data in the file.

Synopsis

```
#include <rw/gbitvec.h>
declare(RWGBitVec,size)
RWGBitVec(size) a;
```

Description

*RW***GBitVec(size)** is a bit vector of fixed length `size`. The length cannot be changed dynamically (see class *RW***BitVec** if you need a bit vector whose length can be changed at run time). Objects of type *RW***GBitVec(size)** are declared with macros defined in the standard C++ header file `<generic.h>`. Bits are numbered from 0 through *size-1*, inclusive.

Persistence

None

Example

In this example, a bit vector 24 bits long is declared and exercised:

```
#include "rw/gbitvec.h"
#include <iostream.h>

const int VECSIZE = 8;

declare(RWGBitVec, VECSIZE)   // declare a 24 bit long vector
implement(RWGBitVec, VECSIZE) // implement the vector

main()
{
  RWGBitVec(VECSIZE) 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(VECSIZE) c = a ^ b;  // Set c to the XOR of a and b.

  cout << "Vector 1" << "\t" << "Vector 2" << "\t"
       << "Vector 1 xor Vector 2" << endl;

  for(int i = 0; i < VECSIZE; i++)
    cout << a[i] << "\t\t" << b[i] << "\t\t" << c[i] << endl;

  return 0;
}
```

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`.

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`.

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 documentation below, an imaginary typedef

```
typedef RWBoolean (*yourTester)(const type*, const void*);
```

has been used for this tester function.

Persistence

None

Example

```
#include <rw/gdlist.h>
#include <rw/rstream.h>
declare(RWGDlist,int)    /* Declare a list of ints */

main()  {
  RWGDlist(int) list;    // Define a list of ints
  int *ip;

  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
```

```
  while(!list.isEmpty())
    delete list.get();      // Remove & delete list items

  return 0;
}
END FILE
```

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 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(*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.

Synopsis

```
#include <rw/gdlist.h>
declare(RWGDlist, type)

RWGDlist(type) a;
RWGDlistIterator(type) I(a) ;
```

Description

Iterator for class *RW***GDlist(type),** which allows sequential access to all the elements of a doubly-linked list. Elements are accessed in order, in either direction.As with all Rogue Wave 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 *RW***GDlist(type)** for an explanation of this function.

Persistence

None

Example

See class *RW***GDlist(type)**

Public Constructor

`RWGDlistIterator(type)( RWGDlist(type)& list);`

Construct an iterator for the *RW***GDlist(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.

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 val val.
```

Description

Class *RW***GOrderedVector(val)** represents an ordered collection of objects of val `val`. Objects are ordered by the order of insertion and are accessible by index. Duplicates are allowed. *RW***GOrderedVector(val)** is implemented as a vector, using macros defined in the standard C++ header file `<generic.h>`.Note that 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 .cpp file you must put the following:
implement(RWGVector,double)           // Implement base class
implement(RWGOrderedVector,double)    // Implement ordered vector
```

For each val of *RW***GOrderedVector** you must include one (and only one) call to the macro `implement` somewhere in your code for both the *RW***GOrderedVector** itself and for its base class *RW***GVector**.

Persistence

None

Example

Here's an example that uses an ordered vector of *RW***CString**s.

```
#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 val conversion occurs
  vec.insert("Last");     // Automatic val 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 val `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 one less than the number of items in the vector. 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 one less than the number of items in the vector. 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.

```
const val*
data() const;
```
Returns a pointer to the raw data of self. Should be used with care.

```
size_t
entries() const;
```
Return the number of items currently in the collection.

```
size_t
index(val item) const;
```
Perform 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;
```
Returns `TRUE` if the collection has no entries. `FALSE` otherwise.

```
void
size_t
length() const;
```
Synonym for `entries()`.

```
val
pop();
```
Removes and returns the last item in the vector.

```
void
push(val);
```
Synonym for `insert()`.

```
removeAt(size_t indx);
```
Removes the item at position `indx` from the collection. The value of `indx` must be between zero and one less than the length of the collection. 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 that the actual number of items in the collection does not change, just the capacity.

Synopsis

```
#include <rw/gqueue.h>
declare(RWGQueue, type)

RWGQueue(type) a ;
```

Description

Class *RW**GQueue(type)*** represents a group of ordered elements, not accessible by an external key. A *RW**GQueue(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 *RW**GQueue(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.

Persistence

None

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`.

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.

Persistence

None

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.

Synopsis

```
#include <rw/gslist.h>
declare(RWGSlist, type)

RWGSlist(type) a ;
RWGSlistIterator(type) I(a);
```

Description

Iterator for class *RW**GSlist(type)**,* which allows sequential access to all the elements of a singly-linked list. Elements are accessed in order, first to last.As with all Rogue Wave 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 *RW**GSlist(type)*** for an explanation of this function.

Persistence

None

Public Constructor

`RWGSlistIterator``(type)( RWGSlist(type)& list);`
Constructs an iterator for the *RW**GSlist(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 it is 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.

*RW**GSortedVector(val)*** → *RW**GVector(val)***

Synopsis

```
#include <rw/gsortvec.h>
declare(RWGSortedVector,val)
implement(RWGSortedVector, val)
RWGSortedVector(val) v;    // A sorted vector of vals .
```

Description

Class *RW**GSortedVector(val)*** represents a vector of elements of val *val*, sorted using an insertion sort. The elements can be retrieved using an index or a search. Duplicates are allowed. Objects of val *RW**GSortedVector(val)*** are declared with macros defined in the standard C++ header file `<generic.h>`.Note that 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 .cpp file you must put the following:
implement(RWGVector,double)          // Implement base class
implement(RWGSortedVector,double)   // Implement sorted vector
```

For each ***val*** of *RW**GSortedVector*** you must include one (and only one) call to the macro `implement` somewhere in your code for both the *RW**GSortedVector*** itself and for its base class *RW**GVector***.

Insertions and retrievals are done using a binary search. Note that the constructor of an *RW**GSortedVector(val)*** requires a pointer to a "comparison function." This function should have protoval:

```
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`.

Persistence None

Example Here's an example of a sorted vector of `int`s:

```
#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 val `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 val `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 ith 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 ith 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
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 that the actual number of items in the collection does not change, just the capacity.

Synopsis

```
#include <rw/gstack.h>
declare(RWGStack,type)

RWGStack(type) a ;
```

Description

Class *RW***GStack(type)** represents a group of ordered elements, not accessible by an external key. A *RW***GStack(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 *RW***GStack(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.

Persistence

None

Public Constructors

`RWGStack(type)``();`
Constructs an empty stack.

`RWGStack(type)``(type* a);`
Constructs 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.

Synopsis

```
#include <rw/gvector.h>
declare(RWGVector,val)
implement(RWGVector,val)

RWGVector(val) a;   // A Vector of val's.
```

Description

Class *RW**GVector(val)*** represents a group of ordered elements, accessible by an index. Duplicates are allowed. This class is implemented as an array. Objects of type *RW**GVector(val)*** are declared with macros defined in the standard C++ header file `<generic.h>`. Note that 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 equivalent).

For each type of *RW**GVector***, you must include one (and only one) call to the macro `implement`, somewhere in your code.

Persistence

None

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 embedded 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) const;
val&
operator()(size_t i);
```
Return the `i`'th element in the vector. The index `i` must be between zero and the length of the vector less one. No bounds checking is performed. The second variant can be used as an lvalue.

```
val
operator[](size_t i) const;
val&
operator[](size_t i);
```
Return the `i`th element in the vector. The index `i` must be between zero and the length of the vector less one. Bounds checking is performed.

Public Member Functions

```
const val*
data() const;
```
Returns a pointer to the raw data of self. Should be used with care.

```
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.

*RW**HashDictionary*** → *RW**Set*** → *RW**HashTable*** → *RW**Collection*** → *RW**Collectable***

Synopsis

```
typedef RWHashDictionary Dictionary;  // Smalltalk typedef.
#include <rw/hashdict.h>
RWHashDictionary  a ;
```

Description

An *RW**HashDictionary*** represents a group of unordered values, accessible by external keys. Duplicate keys are not allowed. *RW**HashDictionary*** is implemented as a hash table of associations of keys and values. Both the key and the value must inherit from the abstract base class *RW**Collectable***, with a suitable definition of the virtual function `hash()` and `isEqual()` for the key.

*This class corresponds to the Smalltalk class **Dictionary**.*

Persistence

None

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.

Note: If you inherit from *RW**HashDictionary*** in the presence of the Standard C++ Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

`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.

Public Member Functions

```
void
applyToKeyAndValue(RWapplyKeyAndValue ap, void* x);
```

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. An untyped argument may be passed to the `ap` function through `x`.

```
RWBinaryTree
asBinaryTree();
RWBag
asBag() const;
RWSet
asOrderedCollection() const;
asSet() const;
RWOrdered
RWBinaryTree
asSortedCollection() const;
```

Converts the *RW**HashDictionary*** to an *RW**Bag***, *RW**Set***, *RW**Ordered***, or an *RW**BinaryTree***. Note that since a dictionary contains pairs of keys and values, the result of this call will be a container holding *RW**CollectableAssociations***. Note also that the return value is a *copy* of the data. This can be very expensive for large collections. Consider using `operator+=()` to insert each *RW**CollectableAssociation*** from this dictionary into a collection of your choice.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***. Removes all key-value pairs in the collection.

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

Redefined from class *RW**Collection***. 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 *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Inherited from class *RW**Set***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW***Collection**. 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 *RW***Collectable**.

```
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 *RW***Collectable** to return `__RWHASHDICTIONARY`.

```
virtual RWBoolean
isEmpty() const;
```

Inherited from class *RW***Set**.

```
virtual RWBoolean
isEqual(const RWCollectable* a) const;
```

Inherited from class *RW***Collectable**.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```

Inherited from class *RW**Set***. 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 *RW**Collection***. 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 *RW**Collection***. Removes *and* deletes the key and value pair where the key `isEqual` to the item pointed to by `target`. Note that 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 part of the removed pair is put in `v`. You are responsible for defining `v` before calling this function.

```
void
resize(size_t n = 0);
```

Inherited from class *RW**Set***.

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

Inherited from class *RW**Collection***.

```
virtual RWCollection*
select(RWtestCollectable testfunc, void* x) const;
```

Evaluates the function pointed to by `tst` for the key of each item in the *RW**HashDictionary***. It inserts keys and values for which the function returns `TRUE` into a new *RW**HashDictionary*** allocated off the heap and returns a pointer to this new collection. Because the new dictionary is allocated *off the heap*, you are responsible for deleting it when done. This is a `virtual` function which hides the non-virtual function inherited from *RW**Collection***.

```
virtual RWCollection*
select(RWtestCollectablePair testfunc, void* x) const;
```

Evaluates the function pointed to by `tst` for both the key and the value of each item in the *RW**HashDictionary***. It inserts keys and values for which the function returns `TRUE` into a new *RW**HashDictionary*** allocated off the heap and returns a pointer to this new collection. Because the new dictionary is allocated *off the heap*, you are responsible for deleting it when done. This is a `virtual` function which hides the non-virtual function inherited from *RW**Collection***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW***HashDictionaryIterator** → *RW***Iterator**

Synopsis

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

RWHashDictionary hd;
RWHashDictionaryIterator  iter(hd);
```

Description

Iterator for class *RW***HashDictionary**, allowing sequential access to all the elements of *RW***HashDictionary**. Since *RW***HashDictionary** is unordered, elements are not accessed in any particular order.

Like all Rogue Wave 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.

Persistence

None

Public Constructor

`RWHashDictionaryIterator(RWHashDictionary&);`

Construct an iterator for an *RW***HashDictionary** collection. Immediately after construction, the position of the iterator is undefined until positioned.

Public Member Operator

```
virtual RWCollectable*
operator()();
```

Redefined from class *RW***Iterator**. 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 *RW***Iterator**. 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 *RW***Iterator**. 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 *RW**Iterator***. Inherited from class *RW**SetIterator***. 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.

Synopsis

```
#include <rw/rwstl/hashmap.h>
rw_hashmap<K,V,Hash,EQ> map;
```

Description

Class *rw_hashmap<K,V,Hash,EQ>* maintains a collection of mappings between `K` and `V`, implemented as a hash table of `pair<const K,V>` . Pairs with duplicate keys are not allowed. Two pairs having duplicate keys is the result of the `EQ` comparison, applied to the first element of each, is `TRUE`. Since this is a *value* based collection, objects are *copied* into and out of the collection. As with all classes that meet the ANSI *associative container* specification, *rw_hashmap* provides for iterators that reference its elements. Operations that alter the contents of *rw_hashmap* may invalidate other iterators that reference the container. Since the contents of *rw_hashmap* are in pseudo-random order, the only iterator ranges that will usually make sense are the results of calling `equal_range(key)`, and the entire range from `begin()` to `end()`.

Persistence

None

Public Typedefs

```
typedef K                    key_type;
typedef Hash                 key_hash;
typedef EQ                   key_equal;
typedef pair<K,V>            value_type; // or ... "const K"
typedef (unsigned)           size_type; //from rw_slist
typedef (int)                difference_type; // from rw_slist
typedef (value_type&)        reference;
typedef (const value_type&) const_reference; //from rw_slist
```

Iterators over *rw_hashmap<K,V,Hash,EQ>* are forward iterators.

```
typedef (scoped Iterator)      iterator;
typedef (scoped ConstIterator)  const_iterator;
```

Public Constructors

```
rw_hashmap<K,V,Hash,EQ>(size_type sz = 1024,
                        const Hash& h = Hash(),
                        const EQ& eq = EQ());
```

Construct an empty *rw_hashmap<K,V,Hash,EQ>* with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator.

```
rw_hashmap<K,V,Hash,EQ>(const rw_hashmap<K,V,Hash,EQ>& map);
```

Construct an *rw_hashmap<K,V,Hash,EQ>* which is a copy of `map`. Each element from `map` will be copied into self.

```
rw_hashmap<K,V,Hash,EQ>(const_iterator first,
                        const_iterator bound
                        size_type sz=1024,
                        const Hash& h = Hash(),
                        const EQ& eq = EQ());
```

Construct an *rw_hashmap<K,V,Hash,EQ>* with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each pair referenced by the range starting with `first` and bounded by `bound`.

```
rw_hashmap<K,V,Hash,EQ>(const value_type* first,
                        const value_type* bound
                        size_type sz=1024,
                        const Hash& h = Hash(),
                        const EQ& eq = EQ());
```

Construct an *rw_hashmap<K,V,Hash,EQ>* with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each pair referenced by the range starting with `first` and bounded by `bound`. If there are items in the range for which the `K` parts of the pairs match `EQ`, then only the first such item will be inserted into self.

Public Destructor

```
~rw_hashmap<K,V,Hash,EQ>();
```

The destructor releases the memory used by the container's implementation.

Public Operators

```
rw_hashmap<K,V,Hash,EQ>&
operator=(const rw_hashmap<K,V,Hash,EQ>& rhs);
```

Sets self to have the same capacity, `Hash` and `EQ` as `rhs`, removes all self's current contents, and replaces them with copies of the elements in `rhs`.

```
bool
operator==(const rw_hashmap<K,V,Hash,EQ> & rhs) const;
```

Returns true if self and `rhs` have the same number of elements, and for each `value_type` in self, there is a `value_type` in `rhs` that has a first part for which the `EQ` object in self returns true, and a second part for which `operator==()` returns true. The need to test both parts means that this operator is slightly slower than the method `equal_by_keys()` described below.

```
V&
operator[](const key_type& key);
```

Returns a reference to the `V` part of a pair held in self which has a part `EQ` to `key`, either by finding such a pair, or inserting one (in which case the reference is to an instance of `V` created by its default constructor).

Accessors

```
iterator
begin();
```

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

`const_iterator`
`begin``() const;`

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

`iterator`
`end``();`

The iterator returned marks the location "off the end" of self. It may not be dereferenced.

`const_iterator`
`end``() const;`

The iterator returned marks the location "off the end" of self. It may not be dereferenced.

`pair<const_iterator, const_iterator>`
`equal_range``(const key_type key) const;`

Returns `pair<const_iterator, const_iterator>(lower_bound(key), upper_bound(key))`. Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

`pair<iterator, iterator>`
`equal_range``(const key_type key);`

Returns `pair<iterator, iterator>(lower_bound(key), upper_bound(key))`. Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

`const_iterator`
`lower_bound``(const key_type& key) const;`

Returns the lower bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

`iterator`
`lower_bound``(const key_type& key);`

Returns the lower bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

`const_iterator`
`upper_bound``(const key_type& key) const;`

Returns the upper bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

```
iterator
upper_bound(const key_type& key);
```
Returns the upper bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

Const Public Member Functions

```
size_type
capacity() const;
```
Returns the number of slots in the hash table that self uses.

```
bool
empty() const;
```
Returns `true` if self is empty.

```
float
fill_ratio() const;
```
Returns the result of calculating `size()/capacity()`.

```
size_type
size() const;
```
Returns the number of pairs currently held in self.

Mutators

```
void
clear();
```
A synonym for `erase(begin(),end());`

```
size_type
erase(const key_type& key);
```
If there is a pair in self for which the first part is `EQ` to `key`, that pair is removed, and `1` is returned. Otherwise, `0` is returned.

```
iterator
erase(iterator iter);
```
Removes the element referenced by `iter` and returns an iterator referencing the "next" element. If `iter` does not reference an item in self, the result is undefined.

```
iterator
erase(iterator first, iterator bound);
```
Removes each element in the range which begins with `first` and is bound by `bound`. Returns an iterator referencing `bound`. If `first` does not reference an item in self (and if `first` and `bound` are not equal), the effect is undefined.

```
pair<iterator,bool>
insert(const value_type& val);
```
If there is no pair in self with first part `EQ` to the first part of `val` then inserts `val`, returning a pair with an iterator referencing the new element and true. Otherwise, returns a pair with an iterator referencing the matching `value_type` and false.

`size_type`
`insert``(iterator ignore, const value_type& val);`

If there is no pair in self with first part `EQ` to the first part of `val` then inserts `val`, returning `1`. Otherwise, does nothing and returns `0`. Note that the first argument is provided only for conformance with the ANSI *associative container* specification, and is ignored by the method, since hash table look up can be done in constant time.

`size_type`
`insert``(const value_type* first, const value_type* bound);`

For each element in the range beginning with `first` and bounded by `bound`, if there is no pair in self with first part `EQ` to the first part of that element, the element is copied into self, or if there is such a pair, the element is skipped. Returns the number of elements inserted.

`size_type`
`insert``(const_iterator first, const_iterator bound);`

For each element in the range beginning with `first` and bounded by `bound`, if there is no pair in self with first part `EQ` to the first part of that element, the element is copied into self, or if there is such a pair, the element is skipped. Returns the number of elements inserted.

`void`
`swap``(rw_hashmap<K,V,Hash,EQ>& other);`

Exchanges the contents of self with `other` including the `Hash` and `EQ` objects. This method does not copy or destroy any of the items exchanged but exchanges the underlying hash tables.

Special Methods for Maps

`size_type`
`count``(const key_type& key) const;`

Returns `1` if self contains a pair with its first element `EQ` to `key`, else `0`.

`bool`
`equal_by_keys``(const rw_hashmap<K,V,Hash,EQ>& rhs) const;`

Returns true if self and `rhs` have the same size, and if for each `value_type` in self, there is a `value_type` in `rhs` such that the `EQ` object in self returns true when called for the first parts of those pairs. Note that this method does not compare the `V` (second) part of the pair of the items, so it will run slightly faster than `operator==()`.

`const_iterator`
`find``(const key_type& key) const;`

Returns a const_iterator referencing the pair with `key` as its first element if such a pair is contained in self, else returns `end()`.

```
iterator
find(const key_type& key);
```

Returns an iterator referencing the pair with `key` as its first element, if such a pair is contained in self, else returns `end()`.

```
void
resize(size_type sz);
```

Resizes self's hash table to have `sz` slots; and re-hashes all self's elements into the new table. Can be very expensive if self holds many elements.

Synopsis

```
#include <rw/rwstl/hashmmap.h>
rw_hashmultimap<K,V,Hash,EQ> mmap;
```

Description

Class *rw_**hashmultimap<K,V,Hash,EQ>*** maintains a collection of mappings between `K` and `V`, implemented as a hash table of `pair<const K,V>` in which there may be many pairs with the same `K` instance. Since this is a *value* based collection, objects are *copied* into and out of the collection. As with all classes that meet the ANSI *associative container* specification, *rw_**hashmap*** provides for iterators that reference its elements. Operations that alter the contents of *rw_**hashmap*** may invalidate other iterators that reference the container. Since the contents of *rw_**hashmap*** are in pseudo-random order, the only iterator ranges that will usually make sense are the results of calling `equal_range(key)`, and the entire range from `begin()` to `end()`.

Persistence

None

Public Typedefs

```
typedef K                      key_type;
typedef Hash                   key_hash;
typedef EQ                     key_equal;
typedef pair<K,V>              value_type; // or ... "const K"
typedef (unsigned)             size_type; //from rw_slist
typedef (int)                  difference_type; // from rw_slist
typedef (value_type&)          reference;
typedef (const value_type&)    const_reference; //from rw_slist
```

Iterators over *rw_**hashmultimap<K,V,Hash,EQ>*** are forward iterators.

```
typedef (scoped Iterator)      iterator;
typedef (scoped ConsIterator)  const_iterator;
```

Public Constructors

```
rw_hashmultimap<K,V,Hash,EQ>(size_type sz = 1024,
                             const Hash& h = Hash(),
                             const EQ& eq = EQ());
```

Construct an empty *rw_**hashmultimap<K,V,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator.

```
rw_hashmultimap<K,V,Hash,EQ>(const
                         rw_hashmultimap<K,V,Hash,EQ>& mmap);
```

Construct an *rw_**hashmultimap<K,V,Hash,EQ>*** which is a copy of `mmap`. Each element from `mmap` will be copied into self.

```
rw_hashmultimap<K,V,Hash,EQ>(const_iterator first,
                             const_iterator bound
                             size_type sz=1024,
                             const Hash& h = Hash(),
                             const EQ& eq = EQ());
```

Construct an *rw_**hashmultimap<K,V,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each pair referenced by the range starting with `first` and bounded by `bound`.

```
rw_hashmultimap<K,V,Hash,EQ>(const value_type* first,
                             const value_type* bound
                             size_type sz=1024,
                             const Hash& h = Hash(),
                             const EQ& eq = EQ());
```

Construct an *rw_**hashmultimap<K,V,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each pair referenced by the range starting with `first` and bounded by `bound`.

Public Destructor

```
~rw_hashmultimap<K,V,Hash,EQ>();
```

The destructor releases the memory used by the container's implementation.

Public Operators

```
rw_hashmultimap<K,V,Hash,EQ>&
operator=(const rw_hashmultimap<K,V,Hash,EQ>& rhs);
```

Sets self to have the same capacity, `Hash` and `EQ` as `rhs`, removes all self's current contents, and replaces them with copies of the elements in `rhs`.

```
bool
operator==(const rw_hashmultimap<K,V,Hash,EQ> & rhs) const;
```

Returns true if self and `rhs` have the same number of elements, and for each `value_type` in self, there is exactly one corresponding `value_type` in `rhs` that has a first part for which the `EQ` object in self returns true, and a second part for which `operator==()` returns true. The need to test both parts, and ensure that the matches are one-to-one means that this operator may be significantly slower than the method `equal_by_keys()` described below.

Accessors

```
iterator
begin();
```

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

const_iterator
begin() const;

The iterator returned references the first item in self. If self is empty, the iterator is equal to end(). Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

iterator
end();

The iterator returned marks the location “off the end” of self. It may not be dereferenced.

const_iterator
end() const;

The iterator returned marks the location “off the end” of self. It may not be dereferenced.

pair<const_iterator, const_iterator>
equal_range(const key_type key) const;

Returns pair<const_iterator,const_iterator>(lower_bound(key), upper_bound(key)). Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

pair<iterator, iterator>
equal_range(const key_type key);

Returns pair<iterator,iterator>(lower_bound(key), upper_bound(key)). Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

const_iterator
lower_bound(const key_type& key) const;

Returns the lower bound of key in self. This has a special meaning for hash-based collections. See discussion elsewhere.

iterator
lower_bound(const key_type& key);

Returns the lower bound of key in self. This has a special meaning for hash-based collections. See discussion elsewhere.

const_iterator
upper_bound(const key_type& key) const;

Returns the upper bound of key in self. This has a special meaning for hash-based collections. See discussion elsewhere.

iterator
upper_bound(const key_type& key);

Returns the upper bound of key in self. This has a special meaning for hash-based collections. See discussion elsewhere.

Const Public Member Functions

```
size_type
capacity() const;
```
Returns the number of slots in the hash table that self uses.

```
bool
empty() const;
```
Returns `true` if self is empty.

```
float
fill_ratio() const;
```
Returns the result of calculating `size()/capacity()`.

```
size_type
size() const;
```
Returns the number of items currently held in self.

Mutators

```
void
clear();
```
A synonym for `erase(begin(),end());`

```
size_type
erase(const key_type& key);
```
Removes all pairs in self for which the first part is `EQ` to `key`, and returns the number of removed elements.

```
iterator
erase(iterator iter);
```
Removes the element referenced by `iter` and returns an iterator referencing the “next” element. If `iter` does not reference an item in self, the result is undefined.

```
iterator
erase(iterator first, iterator bound);
```
Removes each element in the range which begins with `first` and is bound by `bound`. Returns an iterator referencing `bound`. If `first` does not reference an item in self (and if `first` and `bound` are not equal), the effect is undefined.

```
pair<iterator,bool>
insert(const value_type& val);
```
Inserts the pair, `val`, and returns a pair with an iterator referencing the new element and `true`.

```
size_type
insert(iterator ignore, const value_type& val);
```
Inserts the pair, `val`, returning `1`. Note that the first argument is provided only for conformance with the ANSI *associative container* specification, and is ignored by the method, since hash table look up can be done in constant time.

```
size_type
insert(const value_type* first, const value_type* bound);
```

For each element in the range beginning with `first` and bounded by `bound`, the element is copied into self. Returns the number of elements inserted.

```
size_type
insert(const_iterator first, const_iterator bound);
```

For each element in the range beginning with `first` and bounded by `bound`, the element is copied into self. Returns the number of elements inserted.

```
void
swap(rw_hashmultimap<K,V,Hash,EQ>& other);
```

Exchanges the contents of self with `other` including the `Hash` and `EQ` objects. This method does not copy or destroy any of the items exchanged but exchanges the underlying hash tables.

Special Methods for Multimaps

```
size_type
count(const key_type& key) const;
```

Returns the number of pairs in self which have `key` `EQ` to their first element.

```
bool
equal_by_keys(const rw_hashmultimap<K,V,Hash,EQ>& rhs) const;
```

Returns true if self and `rhs` have the same size, and if for each distinct `key_type` in self, self and `rhs` have the same number of pairs with first parts that test `EQ` to that instance. Note that this method does not compare the `V` (second) part of the pair of the items, so it will run slightly faster than `operator==()`.

```
const_iterator
find(const key_type& key) const;
```

Returns a const_iterator referencing some pair with `key` as its first element, if such a pair is contained in self, else returns `end()`.

```
iterator
find(const key_type& key);
```

Returns an iterator referencing some pair with `key` as its first element, if such a pair is contained in self, else returns `end()`.

```
void
resize(size_type sz);
```

Resizes self's hash table to have `sz` slots; and re-hashes all self's elements into the new table. Can be very expensive if self holds many elements.

Synopsis

```
#include <rw/rwstl/hashmset.h>
rw_hashmultiset<T,Hash,EQ> mset;
```

Description

Class *rw_hashmultiset<T,Hash,EQ>* maintains a collection of `T`, implemented as a hash table in which there may be many `EQ` instances of `T`. Since this is a *value* based collection, objects are *copied* into and out of the collection. As with all classes that meet the ANSI *associative container* specification, *rw_hashmap* provides for iterators that reference its elements. Operations that alter the contents of *rw_hashmap* may invalidate other iterators that reference the container. Since the contents of *rw_hashmap* are in pseudo-random order, the only iterator ranges that will usually make sense are the results of calling `equal_range(key)`, and the entire range from `begin()` to `end()`.

Persistence

None

Public Typedefs

```
typedef T                     key_type;
typedef T                     value_type; // or ... "const K"
typedef Hash                  key_hash;
typedef EQ                    key_equal;
typedef (unsigned)            size_type; //from rw_slist
typedef (int)                 difference_type; // from rw_slist
typedef (value_type&)         reference;
typedef (const value_type&)   const_reference; //from rw_slist
```

Iterators over *rw_hashmultiset<T,Hash,EQ>* are forward iterators.

```
typedef (scoped Iterator)      iterator;
typedef (scoped ConsIterator)  const_iterator;
```

Public Constructors

```
rw_hashmultiset<T,Hash,EQ>(size_type sz = 1024,
                           const Hash& h = Hash(),
                           const EQ& eq = EQ());
```

Construct an empty *rw_hashmultiset<T,Hash,EQ>* with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator.

```
rw_hashmultiset<T,Hash,EQ>(const rw_hashmultiset<T,Hash,EQ>&
                           mset);
```

Construct an *rw_hashmultiset<T,Hash,EQ>* which is a copy of `mset`. Each element from `mset` will be copied into self.

```
rw_hashmultiset<T,Hash,EQ>(const_iterator first,
                           const_iterator bound
                           size_type sz=1024,
                           const Hash& h = Hash(),
                           const EQ& eq = EQ());
```

Construct an *rw_**hashmultiset<T,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each item referenced by the range starting with `first` and bounded by `bound`.

```
rw_hashmultiset<T,Hash,EQ>(const value_type* first,
                           const value_type* bound
                           size_type sz=1024,
                           const Hash& h = Hash(),
                           const EQ& eq = EQ());
```

Construct an *rw_**hashmultiset<T,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equals object, containing a copy of each item referenced by the range including `first` and bounded by `bound`.

Public Destructor

```
~rw_hashmultiset<T,Hash,EQ>();
```

The destructor releases the memory used by the container's implementation.

Public Operators

```
rw_hashmultiset<T,Hash,EQ>&
operator=(const rw_hashmultiset<T,Hash,EQ>& rhs);
```

Sets self to have the same capacity, `Hash` and `EQ` as `rhs`, removes all self's current contents, and replaces them with copies of the elements in `rhs`.

```
bool
operator==(const rw_hashmultiset<T,Hash,EQ> & rhs) const;
```

Returns `true` if self and `rhs` have the same number of elements, and for each distinct instance of `T` in self, both self and `rhs` have the same count of instances.

Accessors

```
iterator
begin();
```

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

```
const_iterator
begin() const;
```

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

iterator
end();
The iterator returned marks the location "off the end" of self. It may not be dereferenced.

const_iterator
end() const;
The iterator returned marks the location "off the end" of self. It may not be dereferenced.

pair<const_iterator, const_iterator>
equal_range(const key_type key) const;
Returns `pair<const_iterator, const_iterator>(lower_bound(key), upper_bound(key))`. Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

pair<iterator, iterator>
equal_range(const key_type key);
Returns `pair<iterator, iterator>(lower_bound(key), upper_bound(key))`. Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

const_iterator
lower_bound(const key_type& key) const;
Returns the lower bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

iterator
lower_bound(const key_type& key);
Returns the lower bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

const_iterator
upper_bound(const key_type& key) const;
Returns the upper bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

iterator
upper_bound(const key_type& key);
Returns the upper bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

Const Public Member Functions

size_type
capacity() const;
Returns the number of slots in the hash table that self uses.

bool
empty() const;
Returns `true` if self is empty.

```
float
fill_ratio() const;
```
Returns the result of calculating `size()/capacity()`.

```
size_type
size() const;
```
Returns the number of items currently held in self.

Mutators

```
void
clear();
```
A synonym for `erase(begin(),end());`

```
size_type
erase(const key_type& key);
```
Removes all items in self which are `EQ` to `key`, and returns the number of removed elements.

```
iterator
erase(iterator iter);
```
Removes the element referenced by `iter` and returns an iterator referencing the "next" element. If `iter` does not reference an item in self, the result is undefined.

```
iterator
erase(iterator first, iterator bound);
```
Removes each element in the range which begins with `first` and is bound by `bound`. Returns an iterator referencing `bound`. If `first` does not reference an item in self (and if `first` and `bound` are not equal), the effect is undefined.

```
pair<iterator,bool>
insert(const value_type& val);
```
Inserts `val`, returning a pair with an iterator referencing the new element and true.

```
size_type
insert(iterator ignore, const value_type& val);
```
Inserts `val`, returning `1`. Note that the first argument is provided only for conformance with the ANSI *associative container* specification, and is ignored by the method, since hash table look up can be done in constant time.

```
size_type
insert(const value_type* first, const value_type* bound);
```
For each element in the range beginning with `first` and bounded by `bound`, the element is copied into self. Returns the number of elements inserted.

```
size_type
insert(const_iterator first, const_iterator bound);
```

For each element in the range beginning with `first` and bounded by `bound`, the element is copied into self. Returns the number of elements inserted.

```
void
swap(rw_hashmultiset<T,Hash,EQ>& other);
```

Exchanges the contents of self with `other` including the `Hash` and `EQ` objects. This method does not copy or destroy any of the items exchanged but exchanges the underlying hash tables.

Special Methods for Multisets

```
size_type
count(const key_type& key) const;
```

Returns the number of items in self which are `EQ` to `key`.

```
const_iterator
find(const key_type& key) const;
```

Returns a `const_iterator` referencing some item `EQ` to `key` if such an item is contained in self, else returns `end()`.

```
iterator
find(const key_type& key);
```

Returns an iterator referencing some item `EQ` to `key` if such a item is contained in self, else returns `end()`.

```
void
resize(size_type sz);
```

Resizes self's hash table to have `sz` slots; and re-hashes all self's elements into the new table. Can be very expensive if self holds many elements.

Synopsis

```
#include <rw/rwstl/hashset.h>
rw_hashset<T,Hash,EQ> set;
```

Description

Class *rw_**hashset<T,Hash,EQ>*** maintains a collection of `T`, implemented as a hash table in which there may not be more than one instance of any given `T`. Since this is a *value* based collection, objects are *copied* into and out of the collection. As with all classes that meet the ANSI *associative container* specification, *rw_**hashset*** provides for iterators that reference its elements. Operations that alter the contents of *rw_**hashset*** may invalidate other iterators that reference the container. Since the contents of *rw_**hashset*** are in pseudo-random order, the only iterator ranges that will usually make sense are the results of calling `equal_range(key)`, and the entire range from `begin()` to `end()`.

Persistence

None

Public Typedefs

```
typedef T                      key_type;
typedef T                      value_type; // or ... “const K”
typedef Hash                   key_hash;
typedef EQ                     key_equal;
typedef (unsigned)             size_type; //from rw_slist
typedef (int)                  difference_type; // from rw_slist
typedef (value_type&)          reference;
typedef (const value_type&) const_reference; //from rw_slist
```

Iterators over *rw_**hashset<T,Hash,EQ>*** are forward iterators.

```
typedef (scoped Iterator)     iterator;
typedef (scoped ConsIterator) const_iterator;
```

Public Constructors

```
rw_hashset<T,Hash,EQ>(size_type sz = 1024,
                      const Hash& h = Hash(),
                      const EQ& eq = EQ());
```

Construct an empty *rw_**hashset<T,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator.

```
rw_hashset<T,Hash,EQ>(const rw_hashset<T,Hash,EQ>& set);
```

Construct an *rw_**hashset<T,Hash,EQ>*** which is a copy of `set`. Each element from `set` will be copied into self.

```
rw_hashset<T,Hash,EQ>(const_iterator first,
                      const_iterator bound
                      size_type sz=1024,
                      const Hash& h = Hash(),
                      const EQ& eq = EQ());
```

Construct an *rw_**hashset<T,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each item referenced by the range starting with `first` and bounded by `bound`.

```
rw_hashset<T,Hash,EQ>(const value_type* first,
                      const value_type* bound
                      size_type sz=1024,
                      const Hash& h = Hash(),
                      const EQ& eq = EQ());
```

Construct an *rw_**hashset<T,Hash,EQ>*** with `sz` slots, using `h` as the hash object, and `eq` as the equality comparator, containing a copy of each item referenced by the range starting with `first` and bounded by `bound`. If there are items in the range which test `EQ`, then only the first such item will be inserted into self.

Public Destructor

```
~rw_hashset<T,Hash,EQ>();
```

The destructor releases the memory used by the container's implementation.

Public Operators

```
rw_hashset<T,Hash,EQ>&
operator=(const rw_hashset<T,Hash,EQ>& rhs);
```

Sets self to have the same capacity, `Hash` and `EQ` as `rhs`, removes all self's current contents, and replaces them with copies of the elements in `rhs`.

```
bool
operator==(const rw_hashset<T,Hash,EQ> & rhs) const;
```

Returns true if self and `rhs` have the same number of elements, and for each item in self there is an item in `rhs` which tests `EQ`.

Accessors

```
iterator
begin();
```

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

```
const_iterator
begin() const;
```

The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`. Note that because items are stored in pseudo-random order, this iterator might reference any item that has been stored in self.

```
iterator
end();
```
The iterator returned marks the location “off the end” of self. It may not be dereferenced.

```
const_iterator
end() const;
```
The iterator returned marks the location “off the end” of self. It may not be dereferenced.

```
pair<const_iterator, const_iterator>
equal_range(const key_type key) const;
```
Returns `pair<const_iterator, const_iterator>(lower_bound(key), upper_bound(key))`. Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

```
pair<iterator, iterator>
equal_range(const key_type key);
```
Returns `pair<iterator,iterator>(lower_bound(key), upper_bound(key))`. Upper and lower bound have special meaning for hash-based collections. See discussion elsewhere.

```
onst_iterator
lower_bound(const key_type& key) const;
```
Returns the lower bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

```
iterator
lower_bound(const key_type& key);
```
Returns the lower bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

```
const_iterator
upper_bound(const key_type& key) const;
```
Returns the upper bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

```
iterator
upper_bound(const key_type& key);
```
Returns the upper bound of `key` in self. This has a special meaning for hash-based collections. See discussion elsewhere.

Const Public Member Functions

```
size_type
capacity() const;
```
Returns the number of slots in the hash table that self uses.

```
bool
empty() const;
```
Returns `true` if self is empty.

```
float
fill_ratio() const;
```
Returns the result of calculating `size()/capacity()`.

```
size_type
size() const;
```
Returns the number of items currently held in self.

Mutators

```
void
clear();
```
A synonym for `erase(begin(),end());`

```
size_type
erase(const key_type& key);
```
If there is an item `EQ` to `key`, it is removed, and `1` is returned. Otherwise, `0` is returned.

```
iterator
erase(iterator iter);
```
Removes the element referenced by `iter` and returns an iterator referencing the "next" element. If `iter` does not reference an item in self, the result is undefined.

```
iterator
erase(iterator first, iterator bound);
```
Removes each element in the range which begins with `first` and is bounded by `bound`. Returns an iterator referencing `bound`. If `first` does not reference an item in self (and if `first` and `bound` are not equal), the effect is undefined.

```
pair<iterator,bool>
insert(const value_type& val);
```
If there is no item in self `EQ` to `val` then inserts `val`, returning a pair with an iterator referencing the new element and true. Otherwise, returns a pair with an iterator referencing the matching `value_type` and false.

```
size_type
insert(iterator ignore, const value_type& val);
```
If there is no item in self `EQ` to `val` then inserts `val`, returning `1`. Otherwise, does nothing and returns `0`. Note that the first argument is provided only for conformance with the ANSI *associative container* specification, and is ignored by the method, since hash table look up can be done in constant time.

```
size_type
insert(const value_type* first, const value_type* bound);
```
For each element in the range beginning with `first` and bounded by `bound`, if there is no item in self `EQ` to that element, the element is copied

into self, or if there is such an element, it is skipped. Returns the number of elements inserted.

```
size_type
insert(const_iterator first, const_iterator bound);
```

For each element in the range beginning with `first` and bounded by `bound,` if there is no item in self `EQ` to that element, the element is copied into self, or if there is such an element, it is skipped. Returns the number of elements inserted.

```
void
swap(rw_hashset<T,Hash,EQ>& other);
```

Exchanges the contents of self with `other` including the `Hash` and `EQ` objects. This method does not copy or destroy any of the items exchanged but exchanges the underlying hash tables.

Special Methods for Sets

```
size_type
count(const key_type& key) const;
```

Returns `1` if self contains `key`, else `0`.

```
const_iterator
find(const key_type& key) const;
```

Returns a const_iterator referencing `key`, if it is contained in self, else returns `end()`.

```
iterator
find(const key_type& key);
```

Returns an iterator referencing `key`, if it is contained in self, else returns `end()`.

```
void
resize(size_type sz);
```

Resizes self's hash table to have `sz` slots; and re-hashes all self's elements into the new table. Can be very expensive if self holds many elements.

*RW**HashTable*** → *RW**Collection*** → *RW**Collectable***

Synopsis

```
#include <rw/hashtab.h>
RWHashTable h ;
```

Description

This class is a simple hash table for objects inheriting from *RW**Collectable***. It uses chaining (as implemented by class *RW**SlistCollectables***) to resolve hash collisions. Duplicate objects are allowed.

An object stored by *RW**HashTable*** must inherit from the abstract base class *RW**Collectable***, with suitable definition for virtual functions `hash()` and `isEqual()` (see class *RW**Collectable***).

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 *RW**HashTableIterator***.

Persistence

None

Example

```
hashtab.cpp
#include <rw/hashtab.h>
#include <rw/colldate.h>
#include <rw/rstream.h>

main(){
 RWHashTable table;
 RWCollectableDate *july
      = new RWCollectableDate(7, "July", 1990);
 RWCollectableDate *may
      = new RWCollectableDate (1, "May", 1977);
 RWCollectableDate *feb
      = new RWCollectableDate (22, "Feb", 1983);
 RWCollectableDate *aug
      = new RWCollectableDate (2, "Aug", 1966);

 table.insert(july);
 table.insert(may);
 table.insert(feb);
```

```
 table.insert(aug);

 cout << "Table contains " << table.entries() << " entries.\n";
 RWCollectableDate key(22, "Feb", 1983);
 cout << "It does ";
 if (!table.contains(&key)) cout << "not ";
 cout << "contain the key " << key << endl;

 delete july;
 delete may;
 delete feb;
 delete aug;
 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`. **Note**: If you inherit from *RW**HashTable*** in the presence of the Standard C++ Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

```
RWBoolean
operator!=(const RWHashTable&) const;
```
Returns the negation of `operator==()`, above.

Member Functions

```
virtual void
apply(RWapplyCollectable ap, void*);
```
Redefined from *RW**Collection***. 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 *RW**Collection***.

```
virtual void
clear();
```
Redefined from *RW**Collection***.

```
virtual void
clearAndDestroy();
```
Inherited from *RW**Collection***.

```
virtual int
compareTo(const RWCollectable*) const;
```
Inherited from *RW**Collection***.

```
virtual RWBoolean
contains(const RWCollectable*) const;
```
Inherited from *RW**Collection***.

```
virtual size_t
entries() const;
```
Redefined from *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable*) const;
```
Redefined from *RW**Collection***.

```
virtual unsigned
hash() const;
```
Inherited from *RW**Collection***.

```
virtual RWCollectable*
insert(RWCollectable* a);
```
Redefined from *RW**Collection***. Returns `a` if successful, `nil` otherwise.

```
virtual RWClassID
isA() const;
```

Redefined from *RW**Collection*** to return `__RWHASHTABLE`.

```
virtual RWBoolean
isEmpty() const;
```

Redefined from *RW**Collection***.

```
virtual RWBoolean
isEqual(const RWCollectable*) const;
```

Redefined from *RW**Collection***.

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

Redefined from *RW**Collection***.

```
virtual size_t
occurrencesOf(const RWCollectable*) const;
```

Redefined from *RW**Collection***.

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

Redefined from *RW**Collection***.

```
virtual void
removeAndDestroy(const RWCollectable*);
```

Inherited from *RW**Collection***.

```
virtual void
resize(size_t n = 0);
```

Resizes the internal hash table to have `n` buckets. This causes 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 *RW**Collection***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW**HashTableIterator*** → *RW**Iterator***

Synopsis

```
#include <rw/hashtab.h>
RWHashTable h;
RWHashTableIterator it(h);
```

Description

Iterator for class *RW**HashTable***, which allows sequential access to all the elements of *RW**HashTable***. Note that because an *RW**HashTable*** is unordered, elements are not accessed in any particular order.

As with all Rogue Wave 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.

Persistence

None

Public Constructor

`**RWHashTableIterator**(RWHashTable&);`

Construct an iterator for an *RW**HashTable***. After construction, the position of the iterator is undefined.

Public Member Operator

```
virtual RWCollectable*
operator()();
```

Redefined from class *RW**Iterator***. 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 *RW**Iterator***. 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 *RW**Iterator***. 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 *RW**Iterator***. Resets the iterator to its starting state.

*RW***IdentityDictionary** → *RW***HashDictionary** → *RW***Set** → *RW***HashTable** → ...
... *RW***Collection** → *RW***Collectable**

Synopsis

```
#include <rw/idendict.h>
// Smalltalk typedef:
typedef RWIdentityDictionary IdentityDictionary;
RWIdentityDictionary a;
```

Description

The class *RW***IdentityDictionary** is implemented as a hash table, for the storage and retrieval of key-value pairs. Class *RW***IdentityDictionary** is similar to class *RW***HashDictionary** 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 RW***Collectable .**

The iterator for this class is *RW***HashDictionaryIterator**.

Persistence

None

Public Constructor

`RWIdentityDictionary(size_t n = RWDEFAULT_CAPACITY);`

Construct an empty identity dictionary with `n` hashing buckets.

Public Operator

```
RWBoolean
operator<=(const RWIdentityDictionary& t) const;
```

Returns `TRUE` if self is a subset of `t`, that is, every element of self has a counterpart in `t` which `isEqual`. This operator is not explicitly present unless you are compiling with an implementation of the Standard C++ Library. It is normally inherited from *RW***HashDictionary**.

Note: If you inherit from *RW***IdentityDictionary** in the presence of the Standard C++ Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

Public Member Functions

The user interface to this class is identical to class *RW***HashDictionary** 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 *RW***IdentityDictionary**.

*RW**IdentitySet*** → *RW**Set*** → *RW**HashTable*** → *RW**Collection*** → *RW**Collectable***

Synopsis

```
#include <rw/idenset.h>
typedef RWIdentitySet IdentitySet; // Smalltalk typedef
RWIdentitySet a;
```

Description

The class *RW**IdentitySet*** is similar to class *RW**Set*** 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 RW**SetIterator**.*

Persistence

Polymorphic

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 *RW**Set*** 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 *RW**IdentitySet***.

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*.

Persistence

Isomorphic

Public Constructors

`RWInteger();`

Construct an *RW**Integer*** with value zero (0).

`RWInteger(int i);`

Construct an *RW**Integer*** 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:

```
RWFile& operator<<(RWFile&, const RWInteger&);
```

```
int
value() const;
```

Returns the value of the *RW**Integer***.

```
int
value(int newval);
```

Changes the value of the *RW**Integer*** 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 *RW**Integer*** `x` to a virtual stream or *RW**File***, respectively.

```
RWvistream&
operator>>(RWvistream&, RWInteger& x);
RWFile&
operator>>(RWFile&,     RWInteger& x);
```

Restores an *RW**Integer*** into `x` from a virtual stream or *RW**File***, respectively, replacing the previous contents of `x`.

Synopsis

```
#include <rw/iterator.h>
typedef RWIterator Iterator;  // "Smalltalk" typedef
```

Description

Class *RW**Iterator*** 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.

Persistence

None

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()` indicate equivalence, whichever is appropriate for the actual collection type. However, when an iterator is used with an "identity collection" (*i.e.*, *RW**IdentitySet*** and *RW**IdentityDictionary***), 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.

Synopsis

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

(Abstract base class)

Description

*RW***Locale** 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 *RW***Locale** should act.

There are three ways to use an *RW***Locale** object:

- By passing the object to functions which expect one, such as `RWDate::asString().`
- By specifying a "global" locale using the static member function `RWLocale::global(RWLocale*).` This locale is passed as the default argument to functions that use a locale.
- By "imbuing" a stream with the object, so that when an *RW***Date** or *RW***Time** is written to a stream using `operator<<()`, the appropriate formatting will be used automatically.

Two implementations of *RW***Locale** are provided with the library:

- Class *RW***LocaleSnapshot** 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.
- There is also an internal class that mimics `RWLocaleSnapshot("C")`. If your compiler does not have built-in support for locales, one is constructed automatically at program startup to be used as the default value of `RWLocale::global()`. If your compiler does support locales, `RWLocale::global()` returns a const reference to an instance of `RWLocaleSnapshot("C")`.

Persistence

None

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(const 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 1 for a summary of `strftime()` formatting characters.

```
RWCString
asString(const struct tm* tmbuf,const char* format,
         const RWZone& zone) const;
```

Converts components of the ***struct tm*** object to a string, according to the format string. Each format character in the format string must be preceded by `%`. Any characters not preceded by `%` are treated as ordinary characters which are returned unchanged. You may represent the special character `%` with "`%%`". 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 1 for a summary of `strftime()` formatting characters. This function is not virtual in order to maintain link-compatibility with the previous version of the library.

```
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 int
monthIndex(const RWCString&) const = 0;
```

Interprets its argument as a full or abbreviated month name, returning values 1 through 12 to represent (respectively) January through December, or 0 for an error. Leading white space is ignored.

```
virtual RWBoolean
stringToNum(const RWCString&, double* fp) const = 0;
```

Interprets the *RW**CString*** 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 RWBoolean
stringToNum(const RWCString&, long* ip) const = 0;
```

Interprets the *RW**CString*** 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 1. 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].

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

Format character	Meaning	Example
B	Full month name	February
c	Date and time [may use all members]	Feb 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
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 (deprecated)	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 that 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 RWBoolean
stringToTime(const RWCString&, struct tm*) const = 0;
```

Interprets the *RW**CString*** 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 *RW**CString*** 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 *RW**Date*** or *RW**Time***). 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.

```
virtual int
weekdayIndex(const RWCString&) const = 0;
```

Interprets its argument as a full or abbreviated weekday name, returning values 1 through 7 to represent (respectively) Monday through Sunday, or 0 for an error.

Static Public 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 a class that provides the functionality of `RWLocaleSnapshot("C")`.

```
static const RWLocale&
global();
```

Returns a reference to the present global "default" locale.

```
const RWLocale*
defaultLocale();
```

Returns a pointer to a new instance of either `RWLocaleSnapshot("C")`; or another class that provides the same behavior for compilers that don't fully support Standard C locales.

*RW**LocaleSnapshot*** → *RW**Locale***

Synopsis

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

RWLocaleSnapshot ourLocale("");  // encapsulate user's formats
```

Description

The class *RW**LocaleSnapshot*** implements the *RW**Locale*** interface using Standard C library facilities. To use it, the program creates an *RW**LocaleSnapshot*** 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 *RW**LocaleSnapshot*** created. By creating multiple instances of *RW**LocaleSnapshot***, 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: *RW**LocaleSnapshot*** does not encapsulate character set, collation, or message information.

Class *RW**LocaleSnapshot*** has a set of public data members initialized by its constructor with information extracted from its execution environment.

Persistence

None

Example

Try this program with the environmental variable `LANG` set to various locales:

```
#include <rw/rwdate.h>
#include <rw/locale.h>
#include <iostream.h>

main(){
 RWLocaleSnapshot *userLocale = new RWLocaleSnapshot("");
 RWLocale::global(userLocale);
 // Print a number using the global locale:
 cout << RWLocale::global().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;
 delete userLocale;
 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
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>`.

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, *RW**ModelClient***, supplies the “View” leg.

It maintains a list of dependent *RW**ModelClient*** 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. Subclasses of *RW**ModelClient*** should be prepared to accept such a call.

Persistence

None

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.

```
#include <rw/model.h>
class Dial : public RWModelClient {
public:
  virtual void updateFrom(RWModel* m, void* d);
};
class Thermostat : public RWModel {
  double setting;
public:
  Thermostat( Dial* d )
    { addDependent(d); }
  double temperature() const
    { return setting; }
  void setTemperature(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.

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 *RW**Model***, supplies the “Model” leg. See class *RW**Model*** for details.

Persistence

None

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.

*RW**Ordered*** → *RW**Sequenceable*** → *RW**Collection*** → *RW**Collectable***

Synopsis

```
#include <rw/ordcltn.h>
RWOrdered a;
```

Description

Class *RW**Ordered*** 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 *RW**Ordered*** must inherit from the abstract base class *RW**Collectable***.

Class *RW**Ordered*** is implemented as a vector of pointers, allowing for more efficient traversing of the collection than the linked list classes. *RW**SlistCollectables*** and *RW**DlistCollectables***, but slower insertion in the center of the collection.

Persistence

Polymorphic

Public Constructors

```
RWOrdered(size_t size = RWDEFAULT_CAPACITY);
```

Construct an *RW**Ordered*** 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 *RW**BoundsErr*** 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 *RW**BoundsErr*** will occur. The results of this function can be used as an lvalue.

Public Member Functions

```
virtual RWCollectable*
append(RWCollectable*);
```

Redefined from class *RW**Sequenceable***. 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 *RW**Collection***. 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 *RW**Sequenceable***.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Redefined from class *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW**Sequenceable***. Returns the first item in the collection.

```
virtual unsigned
hash() const;
```
Inherited from class *RW**Collectable***.

```
virtual size_t
index(const RWCollectable*) const;
```
Redefined from class *RW**Sequenceable***.

```
virtual RWCollectable*
insert(RWCollectable* c);
```
Redefined from class *RW**Collection***. 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 *RW**Sequenceable***. 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 *RW**BoundsErr*** will be thrown.

```
virtual RWClassID
isA() const;
```
Redefined from class *RW**Collectable*** to return `__RWORDERED`.

```
virtual RWBoolean
isEmpty() const;
```
Redefined from class *RW**Collection***.

```
virtual RWBoolean
isEqual(const RWCollectable* a) const;
```
Inherited from class *RW**Collectable***.

```
virtual RWCollectable*
last() const;
```
Redefined from class *RW**Sequenceable***. Returns the last item in the collection.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```
Redefined from class *RW**Collection***. Returns the number of items that compare `isEqual` to the item pointed to by `target`.

```
RWCollectable*
prepend(RWCollectable*);
```
Redefined from class *RW**Sequenceable***. 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 *RW**SlistCollectablesStack***. 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 *RW**SlistCollectablesStack***. 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 *RW**Collection***. Removes the first item that `isEqual` to the item pointed to by `target` and returns it. Returns `nil` if no item was found.

```
RWCollectable*
removeAt(size_t index);
```
Removes the item at the position `index` in the collection and returns it.

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

```
RWCollectable*
top() const;
```
This is an alternative implementation of a stack to class *RW**SlistCollectablesStack***. The last item in the collection is returned. If there are no items in the collection, `nil` is returned.

*RW***OrderedIterator** → *RW***Iterator**

Synopsis

```
#include <rw/ordcltn.h>
RWOrdered a ;
RWOrderedIterator iter(a);
```

Description

Iterator for class *RW***Ordered**. Traverses the collection from the first to the last item.

Like all Rogue Wave 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.

Persistence

None

Public Constructors

`RWOrderedIterator(const RWOrdered& a);`

Construct an *RW***OrderedIterator** from an *RW***Ordered**. Immediately after construction the position of the iterator is undefined.

Public Member Operator

```
virtual RWCollectable*
operator()();
```

Redefined from class *RW***Iterator**. 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 *RW***Iterator**. 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 *RW***Iterator**. Returns the item at the current iterator position.

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

Redefined from class *RW***Iterator**. Resets the iterator to its starting state.

*RW**pistream*** → *RW**vistream*** → *RW**vios***

Synopsis

```
#include <rw/pstream.h>
RWpistream pstr(cin); // Construct an RWpistream, using cin's
                      // streambuf
```

Description

Class *RW**pistream*** specializes the abstract base class *RW**vistream*** to restore variables stored in a portable ASCII format by *RW**postream***.

You can think of *RW**pistream*** and *RW**postream*** as an ASCII veneer over an associated ***streambuf*** which are responsible 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 *RW**bistream*** and *RW**bostream*** which are more machine dependent. Because *RW**pistream*** and *RW**postream*** retain no information about the state of their associated ***streambuf***s, their use can be freely exchanged with other users of the ***streambuf*** (such as ***istream*** or ***ifstream***).

*RW**pistream*** can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, *etc.*

Persistence

None

Example

See *RW**postream*** for an example of how to create an input stream for this program.

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

main(){
   // Construct an 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 an *RW**pistream*** from the streambuf `s`.

RWpistream(istream& str);

Initialize an *RWpistream* using the ***streambuf*** associated with the istream str.

Public Operators

virtual RWvistream&
operator>>(char& c);

Redefined from class *RWvistream*. Get the next character from the input stream and store it in c. This member attempts to preserve the symbolic characters values transmitted over the stream.

virtual RWvistream&
operator>>(wchar_t& wc);

Redefined from class *RWvistream*. Get the next wide char from the input stream and store it in wc.

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 *RW***vistream**. Get the next `unsigned short` from the input stream and store it in `s`.

```
virtual RWvistream&
operator>>(unsigned int& i);
```
Redefined from class *RW***vistream**. Get the next `unsigned int` from the input stream and store it in `i`.

```
virtual RWvistream&
operator>>(unsigned long& l);
```
Redefined from class *RW***vistream**. Get the next `unsigned long` from the input stream and store it in `l`.

```
operator void*();
```
Inherited via *RW***vistream** from *RW***vios**.

Public Member Functions

```
virtual int
get();
```
Redefined from class *RW***vistream**. 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 *RW***vistream**. Get the next char and store it in `c`. This member only preserves ASCII numerical codes, not the coresponding character symbol.

```
virtual RWvistream&
get(wchar_t& wc);
```
Redefined from class *RW***vistream**. Get the next `wide char` and store it in `wc`.

```
virtual RWvistream&
get(unsigned char& c);
```
Redefined from class *RW***vistream**. Get the next `unsigned char` and store it in `c`.

```
virtual RWvistream&
get(char* v, size_t N);
```
Redefined from class *RW***vistream**. Get a vector of `char`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit. Note that this member preserves ASCII numerical codes, not their corresponding

character values. If you wish to restore a character string, use the function `getString(char*, size_t).`

```
virtual RWvistream&
get(wchar_t* v, size_t N);
```

Redefined from class *RW***vistream**. Get a vector of wide `char`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit. Note that this member preserves ASCII numerical codes, not their corresponding character values. If you wish to restore a character string, use the function `getString(char*, size_t).`

```
virtual RWvistream&
get(double* v, size_t N);
```

Redefined from class *RW***vistream**. Get a vector of `double`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(float* v, size_t N);
```

Redefined from class *RW***vistream**. Get a vector of `float`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(int* v, size_t N);
```

Redefined from class *RW***vistream**. Get a vector of `int`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
get(long* v, size_t N);
```

Redefined from class *RW***vistream**. Get a vector of `long`s and store them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `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 them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `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 them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit. Note that this member preserves ASCII numerical codes, not their corresponding character values. If you wish to restore a character string, use the 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 them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `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 them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `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 them in the array beginning at `v`. If the restore operation stops prematurely, because there are no more data available on the stream, because an exception is thrown, or for some other reason; `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

```
virtual RWvistream&
getString(char* s, size_t N);
```

Redefined from class *RW**vistream***. 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 `N-1` characters have been read and the `Nth` character is not the string terminator, 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 *RW**ExternalErr*** will be thrown.

```
virtual RWvistream&
getString(wchar_t* ws, size_t N);
```

Redefined from class *RW**vistream***. 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 `N-1` characters have been read and the `Nth` character is not the string terminator, 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 *RW**ExternalErr*** will be thrown.

*RW***postream** → *RW***vostream** → *RW***vios**

Synopsis

```
#include <rw/pstream.h>
// Construct an RWpostream, using cout's streambuf:
RWpostream pstr(cout) ;
```

Description

Class *RW***postream** specializes the abstract base class *RW***vostream** to store variables in a portable (printable) ASCII format. The results can be restored by using its counterpart *RW***pistream**.

You can think of *RW***pistream** and *RW***postream** as an ASCII veneer over an associated `streambuf` which are responsible 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 *RW***bistream** and *RW***bostream** which are more machine dependent. Because *RW***pistream** and *RW***postream** retain no information about the state of their associated ***streambuf***s, their use can be freely exchanged with other users of the ***streambuf*** (such as ***istream*** or ***ifstream***).

The goal of class *RW***postream** and *RW***pistream** 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©;      // Stores the number 10.
```

The expression "`pstr << c`" treats `c` as a symbol for a newline, an unprintable character. The expression "`pstr.put©`" treats `c` as the literal number "10".

Note that variables should not be separated with white space. Such white space would be interpreted literally and would have to be read back in as a character string.

*RW***postream** can be interrogated as to the stream state using member functions `good()`, `bad()`, `eof()`, `precision()`, *etc.*

Persistence

None

Example

See *RW***pistream** 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 an 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 an *RW***postream** from the `streambuf s`.

RWpostream(ostream& str);

Initialize an *RW***postream** from the ***streambuf*** associated with the output stream `str`.

Public Destructor

virtual **~RWvostream**();

This virtual destructor allows specializing classes to deallocate any resources that they may have allocated.

Public Operators

virtual RWvostream&
operator<<(const char* s);

Redefined from class *RW***vostream**. 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 *RW***vostream**. 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 *RW***vostream**. Store the char `c` to the output stream using a portable format. Note that `c` is treated as a character, not a number. This member attempts to preserve the symbolic characters values transmitted over the stream

```
virtual RWvostream&
operator<<(wchar_t wc);
```

Redefined from class *RW***vostream**. Store the `wide char wc` to the output stream using a portable format. Note that `wc` is treated as a character, not a number.

```
virtual RWvostream&
operator<<(unsigned char c);
```

Redefined from class *RW***vostream**. Store the `unsigned char c` to the output stream using a portable format. Note that `c` is treated as a character, not a number.

```
virtual RWvostream&
operator<<(double d);
```

Redefined from class *RW***vostream**. Store the `double d` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(float f);
```

Redefined from class *RW***vostream**. Store the `float f` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(int i);
```

Redefined from class *RW***vostream**. Store the `int i` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(unsigned int i);
```

Redefined from class *RW***vostream**. Store the `unsigned int i` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(long l);
```

Redefined from class *RW***vostream**. Store the `long l` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(unsigned long l);
```

Redefined from class *RW***vostream**. Store the `unsigned long l` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(short s);
```

Redefined from class *RW***vostream**. Store the `short s` to the output stream using a portable format.

```
virtual RWvostream&
operator<<(unsigned short s);
```

Redefined from class *RW***vostream**. Store the `unsigned short s` to the output stream using a portable format.

```
operator void*();
```

Inherited via *RW***vostream** from *RW***vios**.

Public Member Functions

```
int
precision() const;
```

Returns the currently set precision used for writing `float` and `double` data. At construction, the precision is set to `RW_DEFAULT_PRECISION` (defined in `compiler.h`.)

```
int
precision(int p);
```

Changes the precision used for writing `float` and `double` data. Returns the previously set precision. At construction, the precision is set to `RW_DEFAULT_PRECISION` (defined in `compiler.h`.)

```
virtual RWvostream&
flush();
```

Send the contents of the stream buffer to output immediately.

```
virtual RWvostream&
put(char c);
```

Redefined from class *RW***vostream**. Store the `char c` to the output stream, preserving its value using a portable format. This member only preserves ASCII numerical codes, not the coresponding character symbol.

```
virtual RWvostream&
put(wchar_t wc);
```

Redefined from class *RW***vostream**. Store the wide character `wc` to the output stream, preserving its value using a portable format.

```
virtual RWvostream&
put(unsigned char c);
```

Redefined from class *RW***vostream**. 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 *RW***vostream**. Store the vector of `char`s starting at `p` to the output stream, preserving their values using a portable format.

Note that 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 *RW**vostream***. Store the vector of wide `char`s starting at `p` to the output stream, preserving their values using a portable format. Note that 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 *RW**vostream***. Store the vector of `unsigned char`s 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 *RW**vostream***. Store the vector of `short`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const unsigned short* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `unsigned short`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const int* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `int`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const unsigned int* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `unsigned int`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const long* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `long`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const unsigned long* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `unsigned long`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const float* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `float`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
put(const double* p, size_t N);
```

Redefined from class *RW**vostream***. Store the vector of `double`s starting at `p` to the output stream using a portable format.

```
virtual RWvostream&
putString(const char*s, size_t N);
```

Store the character string, *including embedded nulls,* starting at s to the output string.

*RW***Sequenceable** → *RW***Collection** → *RW***Collectable**

Synopsis

```
#include <rw/seqcltn.h>
typedef RWSequenceable SequenceableCollection;
             // Smalltalk typedef
```

Description

Class *RW***Sequenceable** is an abstract base class for collections that can be accessed by an index. It inherits class *RW***Collection** as a public base class and adds a few extra virtual functions. This documentation only describes these extra functions.

Persistence

Polymorphic

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 *RW***BoundsErr** 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.

```
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 *RW***BoundsErr** 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.

*RW***Set** → *RW***HashTable** → *RW***Collection** → *RW***Collectable**

Synopsis

```
typedef RWSet Set;    // Smalltalk typedef.
#include <rw/rwset.h>

RWSet h ;
```

Description

Class *RW***Set** 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 *RW***Set** must inherit abstract base class *RW***Collectable**, with suitable definition for virtual functions `hash()` and `isEqual()` (see class *RW***Collectable**). 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 with the same has value as `c` for which `isEqual(c)` returns `TRUE`. In this case, method `insert(c)` will not add it, thus insuring that there are no duplicates.

The iterator for this class is *RW***SetIterator**.

Persistence

Polymorphic

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`. **Note**: If you inherit from *RWSet* in the presence of the C++ Standard Library, we recommend that you override this operator and explicitly forward the call. Overload resolution in C++ will choose the Standard Library provided global operators over inherited class members. These global definitions are not appropriate for set-like partial orderings.

```
RWBoolean
operator<(const RWSet& h);
```

Returns `TRUE` if self is a proper subset of `h`, that is, every element of self has a counterpart in `h` which `isEqual`, but where the two sets are not identical.

```
RWSet&
operator*=(const RWSet& h);
```

Sets self to be the intersection of self and `h`. Returns self.

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 a (generally) unpredictable order. This supplied function must 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();
```

Inherited from class *RWCollection*.

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

Redefined 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 *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```
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 *RW**Collectable***.

```
virtual RWCollectable*
insert(RWCollectable* c);
```
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 *RW**Collectable*** to return `__RWSET`.

```
virtual RWBoolean
isEmpty() const;
```
Inherited from class *RW**Collectable***.

```
virtual RWBoolean
isEqual(const RWCollectable* a) const;
```
Redefined from class *RW**Collection***.

```
void
intersectWith(const RWSet& h, RWSet& ret) const;
```
Computes the intersection of self and h, and inserts the result into `ret` (which may be either empty or not, depending on the effect desired). It may be slightly more efficient than `operator*=()`.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```
Redefined from class *RW**Collection***. Returns the count 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 *RW**Collection***. 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 *RW**Collection***.

```
void
resize(size_t n = 0);
```

Resizes the internal hashing table to leave `n` slots. If `n==0`, resizes to `3*entries()/2`.

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

Inherited from class *RW**Collection***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW***SetIterator** → *RW***HashTableIterator** → *RW***Iterator**

Synopsis

```
#include <rw/rwset.h>
RWSet h;
RWSetIterator it(h) ;
```

Description

Iterator for class *RW***Set**, which allows sequential access to all the elements of *RW***Set**. Note that because an *RW***Set** is unordered, elements are not accessed in any particular order.

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.

Persistence

None

Public Constructor

`RWSetIterator(RWSet&);`

Construct an iterator for an *RW***Set**. After construction, the position of the iterator will be undefined.

Public Member Operator

```
virtual RWCollectable*
operator()();
```

Inherited from *RW***HashTableIterator**.

Public Member Functions

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

Inherited from *RW***HashTableIterator**.

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

Inherited from *RW***HashTableIterator**.

```
RWCollectable*
remove();
```

Inherited from *RW***HashTableIterator**.

```
RWCollectable*
removeNext(const RWCollectable*);
```

Inherited from *RW***HashTableIterator**.

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

Inherited from *RW***HashTableIterator**.

Synopsis

```
#include <rw/rwstl/slist.h>
rw_slist<T> list;
```

Description

Class *rw_**slist<T>*** maintains a collection of ***T***, implemented as a singly-linked list. Since this is a *value* based list, objects are copied into and out of the links that make up the list. As with all classes that meet the ANSI *sequence* specification, *rw_**slist*** provides for iterators that reference its elements. Operations that alter the contents of *rw_**slist*** will invalidate iterators that reference items at or after the location of change.

Public Typedefs

```
typedef T                    value_type;
typedef T&                   reference;
typedef const T&             const_reference;
typedef (unsigned)           size_type; //from Allocator<Node>
```

Iterators over *rw_**slist<T>*** are forward iterators.

```
typedef (scoped Iterator)      iterator;
typedef (scoped ConstIterator) const_iterator;
```

Public Constructors

`rw_slist<T>``();`
Construct an empty *rw_**slist<T>***.

`rw_slist<T>``(const rw_slist<T>& list);`
Construct an *rw_**slist<T>*** which is a copy of `list`. Each element from `list` will be copied into self.

`rw_slist<T>``(size_type count, const T& value);`
Construct an *rw_**slist<T>*** containing exactly `count` copies of `value`.

`rw_slist<T>``(const_iterator first, const_iterator bound);`
Construct an *rw_**slist<T>*** containing a copy of each element referenced by the range starting at `first` and bounded by `bound`.

`rw_slist<T>``(const T* first, const T* bound);`
Construct an *rw_**slist<T>*** containing a copy of each element referenced by the range starting at `first` and bounded by `bound`.

Public Destructor

`~rw_slist<T>``();`
The destructor releases the memory used by the links.

Accessors

```
iterator
begin();
```
The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`.

```
const_iterator
begin() const;
```
The iterator returned references the first item in self. If self is empty, the iterator is equal to `end()`.

```
iterator
end();
```
The iterator returned marks the location “off the end” of self. It may not be dereferenced.

```
const_iterator
end() const;
```
The iterator returned marks the location “off the end” of self. It may not be dereferenced.

```
T&
front();
```
References the first item in the list as an L-value. If self is empty, the behavior is undefined.

```
const T&
front();
```
References the first item in the list as an R-value. If self is empty, the behavior is undefined.

Const Public Member Functions

```
bool
empty() const;
```
Returns `true` if self is empty.

```
size_type
size() const;
```
Returns the number of items currently held in self.

Mutators

```
iterator
erase(iterator iter);
```
Removes from self the element referenced by `iter`. If iter does not reference an actual item contained in self, the effect is undefined. Returns an iterator referencing the location just after the erased item.

```
iterator
erase(iterator first, iterator bound);
```
Removes from self the elements referenced by the range beginning at `first` and bounded by `bound`. Returns an iterator referencing a position just after the last erased item. If `first` does not reference an item in self (and if `first` and `bound` are not equal), the effect is undefined.

`iterator`
`insert``(iterator loc, const T& val);`

Insert `val` just prior to the place referenced by `loc`. Returns an iterator referencing the newly inserted element. (Note: `++(list.insert(loc,val))==loc;`)

`iterator`
`insert``(iterator loc, const_iterator first, const_iterator bound);`

Insert a copy of each item in the range beginning at `first` and bounded by `bound` into self at a place just prior to the place referenced by `loc`. Returns an iterator referencing the last newly inserted element. (Note: `++(list.insert(loc,first,bound))==loc;`)

`iterator`
`insert``(iterator loc, const T* first, const T* bound);`

Insert a copy of each item in the range beginning at `first` and bounded by `bound` into self at a place just prior to the place referenced by `loc`. Returns an iterator referencing the last newly inserted element. (Note: `++(list.insert(loc,first,bound))==loc;`)

`void`
`pop_front``();`

Erases the first element of self. If self is empty, the effect is undefined.

`void`
`push_back``(const T& item);`

Inserts `item` as the last element of the list.

`void`
`push_front``(const T& item);`

Inserts `item` as the first element of the list.

`void`
`reverse``();`

Reverses the order of the nodes containing the elements in self.

`void`
`sort``();`

Sorts self according to `T::operator<(T)` or equivalent. Runs in time proportional to `N log(N)` where `N` is the number of elements.This is method does not copy or destroy any of the items exchanged during the sort, but adjusts the order of the links in the list.

`void`
`swap``(rw_slist<T>& other);`

Exchanges the contents of self with `other` retaining the ordering of each. This is method does not copy or destroy any of the items exchanged, but re-links the lists.

```
void
unique();
```

Removes from self all but the first element from each equal range. A precondition is that any duplicate elements are adjacent.

Special Methods for Lists

```
void
merge(rw_slist& donor);
```

Assuming both `donor` and self are sorted, moves every item from `donor` into self, leaving `donor` empty, and self sorted. If either list is unsorted, the move will take place, but the result may not be sorted. This method does not copy or destroy the items in `donor`, but re-links list nodes into self.

```
void
splice(iterator to, rw_slist<T>& donor);
```

Insert the entire contents of `donor` into self, just before the position referenced by `to`, leaving `donor` empty. This method does not copy or destroy any of the items moved, but re-links the list nodes from `donor` into self.

```
void
splice(iterator to, rw_slist<T>& donor, iterator from);
```

Remove from `donor` and insert into self, just before location `to`, the item referenced by `from`. If `from` does not reference an actual item contained in `donor` the effect is undefined. This method does not copy or destroy the item referenced by `from`, but re-links the node containing it from `donor` into self.

```
void
splice(iterator to, rw_slist<T>& donor, iterator from_start,
       iterator from_bound);
```

Remove from `donor` and insert into self just before location `to`, the items referenced by the range beginning with `from_start` and bounded by `from_bound`. If that range does not refer to items contained by `donor`, the effect is undefined. This method does not copy or destroy the items referenced by the range, but re-links those list nodes from `donor` into self.

Related Global Operators

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

Returns true if `lhs` and `rhs` have the same number of elements and each element of `rhs` tests equal (`T::operator==()` or equivalent) to the corresponding element of `lhs`.

```
bool
operator<(const rw_slist<T>& lhs, const rw_slist<T>& rhs);
```

Returns the result of calling

```
lexicographical_compare(lhs.begin(), lhs.end(),
     rhs.begin(), rhs.end());
```

*RW**SlistCollectables*** → *RW**Sequenceable*** → *RW**Collection*** → *RW**Collectable***

Synopsis

```
// Smalltalk typedef:
typedef RWSlistCollectables LinkedList ;
#include <rw/slistcol.h>
RWSlistCollectables a;
```

Description

Class *RW**SlistCollectables*** 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 *RW**SlistCollectables*** must inherit abstract base class *RW**Collectable***.

The virtual function `isEqual()` (see class *RW**Collectable***) is required to find a match between a target and an item in the collection

Class *RW**SlistCollectables*** 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***.

Persistence

Polymorphic

Public Constructors

`RWSlistCollectables();`
Constructs an empty linked list.

`RWSlistCollectables(RWCollectable* a);`
Constructs a linked list 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 *RW**Sequenceable***. 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 *RW**Collection***. 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 *RW**Sequenceable***. The index `i` must be between 0 and the number of items in the collection less one, or an exception of type *RW**BoundsErr*** will be thrown. Note that for a linked list, these functions must traverse all the links, making them not particularly efficient.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Redefined from class *RW**Collection***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
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 *RW**Collection***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Redefined from class *RW**Collection***. 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 *RW***Sequenceable**. 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 *RW***Collectable**.

```
virtual size_t
index(const RWCollectable* c) const;
```

Redefined from class *RW***Sequenceable**. Returns the index of the first item that `isEqual` to the item pointed to by `c`. If there is no such item, returns `RW_NPOS`.

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

Redefined from class *RW***Collection**. 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 *RW***Sequenceable**. 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 *RW***BoundsErr** will be thrown.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW***Collectable** to return `__RWSLISTCOLLECTABLES`.

```
virtual RWBoolean
isEmpty() const;
```

Redefined from class *RW***Collection**.

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

Redefined from class *RW***Sequenceable**. Returns the value at the end of the collection.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```

Redefined from class *RW***Collection**. 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 *RW**Sequenceable***. 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 *RW**Collection***. 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 *RW**Collection***.

```
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 *RW**Collection***.

```
RWStringID
stringID();
```

(acts virtual) Inherited from class *RW**Collectable***.

*RW***SlistCollectablesIterator** → *RW***Iterator**

Synopsis

```
// Smalltalk typedef.
typedef RWSlistCollectablesIterator LinkedListIterator;
#include <rw/slistcol.h>
RWSlistCollectables sc;
RWSlistCollectablesIterator sci(sc) ;
```

Description

Iterator for class *RW***SlistCollectables**. Traverses the linked-list from the first to last item.

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.

Persistence

None

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 *RW***Iterator**. 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. 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.

*RW**SlistCollectablesQueue*** → *RW**SlistCollectables*** → *RW**Sequenceable*** → ...
... *RW**Collection*** → *RW**Collectable***

Synopsis

```
// Smalltalk typedef:
typedef RWSlistCollectablesQueue Queue ;
#include <rw/queuecol.h>
RWSlistCollectablesQueue a;
```

Description

Class *RW**SlistCollectablesQueue*** represents a restricted interface to class *RW**SlistCollectables*** 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 *RW**SlistCollectablesQueue*** must inherit abstract base class *RW**Collectable***. The virtual function `isEqual()` (see class *RW**Collectable***) is required, to find a match between a target and an item in the queue.

This class corresponds to the Smalltalk class ***Queue***.

Persistence

Polymorphic

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 *RW**SlistCollectables***.

```
virtual RWCollectable*
append(RWCollectable*);
```
Inherited from class *RW**SlistCollectables***. Adds an element to the end of the queue.

```
virtual RWspace
binaryStoreSize() const;
```
Inherited from class *RW**Collection***.

```
virtual void
clear();
```
Inherited from class *RW**SlistCollectables***.

```
virtual void
clearAndDestroy();
virtual RWBoolean
contains(const RWCollectable* target) const;
```
Inherited from class *RW**Collection***.

```
RWBoolean
containsReference(const RWCollectable* e) const;
virtual size_t
entries() const;
```
Inherited from class *RW**SlistCollectables***.

```
virtual RWCollectable*
first() const;
```
Inherited from class *RW**SlistCollectables***. 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 *RW**SlistCollectables***. 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 *RW**SlistCollectables*** to call `append()`.

```
virtual RWClassID
isA() const;
```
Redefined from class *RW**Collectable*** to return `__RWSLISTCOLLECTABLESQUEUE`.

```
virtual RWBoolean
isEmpty() const;
```
Inherited from class *RW**SlistCollectables***.

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

Inherited from class *RW**SlistCollectables***. 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 *RW**SlistCollectables***.

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

Redefined from class *RW**SlistCollectables***. Calls `get()`. The argument is ignored.

*RW**SlistCollectablesStack*** → *RW**SlistCollectables*** → *RW**Sequenceable*** → ...
...*RW**Collection*** → *RW**Collectable***

Synopsis

```
// Smalltalk typedef:
typedef RWSlistCollectablesStack Stack;
#include <rw/stackcol.h>
RWSlistCollectablesStack a;
```

Description

Class *RW**SlistCollectablesStack*** represents a restricted interface to class *RW**SlistCollectables*** 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 *RW**SlistCollectablesStack*** must inherit abstract base class *RW**Collectable***. The virtual function `isEqual()` (see class *RW**Collectable***) is required, to find a match between a target and an item in the stack.

This class corresponds to the Smalltalk class ***Stack***.

Persistence

Polymorphic

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 *RW**SlistCollectables***.

```
virtual void
clearAndDestroy();
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
RWBoolean
containsReference(const RWCollectable* e) const;
virtual size_t
entries() const;
```

Inherited from class *RW**SlistCollectables***.

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

Inherited from class *RW**SlistCollectables***. Same as `top()`.

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

Inherited from class *RW**SlistCollectables***. Same as `push()`.

```
virtual RWClassID
isA() const;
```

Redefined from class *RW**Collectable*** to return `__RWSLISTCOLLECTABLESSTACK`.

```
virtual RWBoolean
isEmpty()const;
```

Inherited from class *RW**SlistCollectables***.

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

Inherited from class *RW**SlistCollectables***. 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 *RW**SlistCollectables***.

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

Redefined from class *RW**SlistCollectables***. 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.

*RW**SortedVector*** → *RW**Ordered*** → *RW**Sequenceable*** → ...
... *RW**Collection*** → *RW**Collectable***

Synopsis

```
#include <rw/sortvec.h>
RWSortedVector a;
```

Description

Class *RW**SortedVector*** 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 *RW**SortedVector*** must inherit from the abstract base class *RW**Collectable***. An insertion sort is used to maintain the vector in sorted order.

Because class *RW**SortedVector*** is implemented as a vector of pointers, traversing the collection is more efficient than with class *RW**BinaryTree***. However, insertions are slower in the center of the collection.

Note that because the vector is sorted, you must not modify elements contained in the vector in such a way as to invalidate the ordering.

Persistence

Polymorphic

Example

```
sortvec.cpp
#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 *RW**SortedVector*** that has an initial capacity of size items. The capacity will be increased automatically as needed.

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 ith element in the collection. If `i` is out of range, an exception of type *RW**BoundsErr*** will be thrown. The return value 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 "`rwsortvec.h`". In this case, if i is out of range, an exception of type *RW**BoundsErr*** will be thrown. The return value cannot be used as an lvalue.

Public Member Functions

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

Inherited from class *RW**Ordered***.

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

Inherited from class *RW**Ordered***.

```
virtual RWspace
binaryStoreSize() const;
```

Inherited from class *RW**Collection***.

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

Inherited from class *RW**Ordered***.

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

Inherited from class *RW**Collection***.

```
virtual int
compareTo(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

```
virtual RWBoolean
contains(const RWCollectable* target) const;
```

Inherited from class *RW**Collection***.

```
virtual size_t
entries() const;
```

Inherited from class *RW**Ordered***.

```
virtual RWCollectable*
find(const RWCollectable* target) const;
```

Inherited from class *RW**Ordered***. Note that `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 *RW**Ordered***.

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

Inherited from class *RW**Collectable***.

```
virtual size_t
index(const RWCollectable*) const;
```

Redefined from class *RW**Ordered***. 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 *RW**Ordered***. 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 *RW**Collectable*** to return `__RWSORTEDVECTOR`.

```
virtual RWBoolean
isEmpty() const;
```

Inherited from class *RW**Ordered***.

```
virtual RWBoolean
isEqual(const RWCollectable* a) const;
```

Inherited from class *RW**Collectable***.

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

Inherited from class *RW**Ordered***.

```
virtual size_t
occurrencesOf(const RWCollectable* target) const;
```

Redefined from class *RW**Ordered***. Returns the number of items that compare equal to the item pointed to by target.

```
virtual RWCollectable*
remove(const RWCollectable* target);
```

Inherited from class *RW**Ordered***. Note that `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 *RW**Collection***.

```
RWCollectable*
removeAt(size_t index);
```

Inherited from class *RW**Ordered***. Removes the item at the position index in the collection and returns it.

Synopsis

```
#include <rw/tbitvec.h>
RWTBitVec<22>   // A 22 bit long vector
```

Description

RWTBitVec<size> is a parameterized bit vector of fixed length *size*. Unlike class ***RWBitVec***, its length cannot be changed at run time. The 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.

Persistence

None

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`.

```
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.

Logical Operators

```
RWBoolean
operator==(RWBoolean b) const;
```
Returns `TRUE` if every bit of self is set to the value `b`. Otherwise, returns `FALSE`.

```
RWBoolean
operator!=(RWBoolean b) const;
```
Returns `TRUE` if any bit of self is not set to the value `b`. Otherwise, returns `FALSE`.

```
RWBoolean
operator==(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`.

```
RWBoolean
operator!=(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`.

```
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 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_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.

```
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 two lines 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

```
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`.

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 1, 1901 UTC. See Section 8 for how to set the time zone for your compiler. Failure to do this may result in UTC (GMT) times being wrong.

Output formatting is done using an *RWLocale* object. The default locale formats according to U.S. conventions.

Note that because the default constructor for this class creates an instance holding the current date and time, constructing a large array of *RWTime* may be slow.

```
RWTime v[5000];     // Figures out the current time 5000 times
```

Those with access to the C++ Standard Library-based versions of the *Tools.h++* template collections should consider the following:

```
// Figures out the current time just once:
RWTValOrderedVector<RWTime> v(5000, RWTime());
```

Thanks to the smart allocation scheme of the standard collections, the above declaration will result in only one call to the default constructor followed by 5000 invocations of the copy constructor. In the case of *RWTime,* the copy constructor amounts to an assignment of one `long` to another, resulting in faster creation than the simple array.

Persistence

Simple

Example

This example constructs a current time, and the time when Daylight-Saving Time starts in the year 1990. It then prints them out.

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

```
RWTime(unsigned long s);
```

Constructs a time with `s` seconds since 00:00:00 January 1, 1901 UTC. If `s==0`, an invalid time is constructed. Note that for small `s` this may be prior to January 1, 1901 in your time zone.

```
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. Note that the maximum *RWTime* is much sooner than maximum *RWDate*. (In fact, it is on Feb. 5, 2037 for platforms with 4-byte `long`s.) This is a consequence of the fact that *RWTime* counts seconds while *RWDate* only deals with full days.

```
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 that 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 string `str` should contain only the time. 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.

```
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+=(unsigned long s);
```
Add `s` seconds to self, returning self.

```
RWTime&
operator-=(unsigned 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`". The exact format of the date and time returned is dependent upon the implementation of `strftime()` available. For more information, look under *RWLocale*.

```
RWCString
asString(char* format,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()`.

```
RWBoolean
between(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 that 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-Saving 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).

```
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-Saving 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-Saving Time for the given year, in the given time zone. Returns an “invalid time” if DST is not observed in that year and zone.

```
static unsigned
hash(const RWTime& t);
```
Returns the hash value of `t` as returned by `t.hash()`.

```
static RWTime
now();
```
Returns the present time.

Related Global Operators

```
RWTime
operator+(const RWTime& t, unsigned long s);
RWTime
operator+(unsigned long s, const RWTime& t);
```
Returns an ***RWTime*** `s` seconds greater than `t`.

```
RWTime
operator-(const RWTime& t, unsigned long s);
```
Returns an ***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 *RW**Locale***), or by `RWLocale::global()` if none.

```
RWvostream&
operator<<(RWvostream&, const RWTime& t);
RWFile&
operator<<(RWFile&,     const RWTime& t);
```
Saves *RW**Time*** `t` to a virtual stream or *RW**File***, respectively.

```
RWvistream&
operator>>(RWvistream&, RWTime& t);
RWFile&
operator>>(RWFile&,     RWTime& t);
```
Restores an *RW**Time*** into `t` from a virtual stream or *RW**File***, respectively, replacing the previous contents of `t`.

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()`.

RWTimer uses the system-dpendent function `clock()` which returns the number of “ticks” since it was first called. As a result, ***RWTimer*** will not be able to measure intervals longer than some system-dependent value. (For instance, on several common UNIX systems, this value is just under 36 minutes.)

Persistence

None

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 = RWTime::now())
 {;}

 t.stop();                     // Stop the timer

 cout << t.elapsedTime() << endl;
 return 0;
}
```

Program output (exact value may differ):

```
5.054945
```

Public Constructor

```
RWTimer();
```

Constructs a new timer. The timer will not start running until `start()` is called.

Public Member Functions

```
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.

Synopsis

```
#include <rw/tidlist.h>
RWTIsvDlist<T> 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.

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

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 Member Functions

void
append(T* a);

Appends the item pointed to by `a` to the end of the list.

void
apply(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.

void
clear();

Removes all items from the list.

void
clearAndDestroy();

Removes *and calls delete* for each item in the list. Note that this assumes that each item was allocated off the heap.

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` . 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`.

RWBoolean
containsReference(const T* a) const;

Returns `TRUE` if the list contains an item with the address `a`.

```
size_t
entries() 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_t
index(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`.

```
void
insert(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.

```
void
insertAt(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.

```
RWBoolean
isEmpty() 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_t
occurrencesOf(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_t
occurrencesOfReference(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.

```
void
prepend(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.

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.

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.

Persistence

None

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.

```
void
insertAfterPoint(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.

```
void
reset();
```

Resets the iterator to the state it had immediately after construction.

```
void
reset(RWTIsvDlist<TL>& c);
```

Resets the iterator to iterate over the collection `c`.

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.

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

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.

```
void
apply(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.

```
void
clear();
```
Removes all items from the list.

```
void
clearAndDestroy();
```
Removes *and calls delete* for each item in the list. Note that this assumes that each item was allocated off the heap.

```
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`. 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`.

```
RWBoolean
containsReference(const T* a) const;
```
Returns `TRUE` if the list contains an item with the address `a`.

```
size_t
entries() 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_t
index(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`.

```
void
insert(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.

```
void
insertAt(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.

```
RWBoolean
isEmpty() 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_t
occurrencesOf(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_t
occurrencesOfReference(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.

```
void
prepend(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.

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.

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.

Persistence

None

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.

```
void
insertAfterPoint(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.

```
void
reset();
```

Resets the iterator to the state it had immediately after construction.

```
void
reset(RWTIsvSlist<TL>& c);
```

Resets the iterator to iterate over the collection `c`.

Synopsis

```
#include <rw/tpdeque.h>
RWTPtrDeque<T> deq;
```

Please Note!

***RWTPtrDeque* requires the Standard C++ Library.**

Description

This class maintains a pointer-based collection of values, implemented as a double-ended queue, or *deque*. Class `T` is the type pointed to by the items in the collection.

Persistence

Isomorphic

Example

In this example, a double-ended queue of `int`s is exercised.

```
// tpdeque.cpp
#include <rw/tpdeque.h>
#include <iostream.h>

/*
 * This program partitions integers into even and odd numbers
 */

int main(){
  RWTPtrDeque<int> numbers;

  int n;

  cout << "Input an assortment of integers (EOF to end):"
       << endl;

  while (cin >> n) {
    if (n % 2 == 0)
      numbers.pushFront(new int(n));
    else
      numbers.pushBack(new int(n));
  }

  while (numbers.entries()) {
    cout << *numbers.first() << endl;
    delete numbers.popFront();
  }

  return 0;
}
```

Program Input:

```
1 2 3 4 5
<eof>
```

Program Output:

```
4
2
1
3
5
```

Related Classes

Classes *RWTPtrDlist<T>*, *RWTPtrSlist<T>*, and *RWTPtrOrderedVector<T>* also provide a Rogue Wave pointer-based interface to C++-standard sequence collections.

Class `deque<T*, allocator>` is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef deque<T*, allocator>                     container_type;
typedef container_type::iterator                 iterator;
typedef container_type::const_iterator           const_iterator;
typedef container_type::size_type                size_type;
typedef container_type::difference_type          difference_type;
typedef T*                                       value_type;
typedef T*&                                      reference;
typedef T* const&                                const_reference;
```

Public Constructors

`RWTPtrDeque<T>``();`
Constructs an empty, double-ended queue.

`RWTPtrDeque<T>``(const deque<T*, allocator>& deq);`
Constructs a double-ended queue by copying all elements of `deq`.

`RWTPtrDeque<T>``(const RWTPtrDeque<T>& rwdeq);`
Copy constructor.

`RWTPtrDeque<T>``(size_type n, T* a);`
Constructs a double-ended queue with `n` elements, each initialized to `a`.

`RWTPtrDeque<T>``(T* const* first, T* const* last);`
Constructs a double-ended queue by copying elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTPtrDeque<T>&`
`operator=``(const RWTPtrDeque<T>& deq);`
Clears all elements of self and replaces them by copying all elements of `deq`.

`RWTPtrDeque<T>&`
`operator=``(const deque<T*, allocator>& stddeq);`
Clears all elements of self and replaces them by copying all elements of `stddeq`.

`bool`
`operator<``(const RWTPtrDeque<T>& deq);`
Returns `true` if self compares lexicographically less than `deq`, otherwise returns `false`. Items in each collection are dereferenced before being compared. Assumes that type `T` has well-defined less-than semantics.

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

Returns `true` if self compares equal to `deq`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between `0` and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed.*

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(T* a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(T*,void*), void* d);
void
apply(void (*fn)(const T*,void*), void* d) const;
void
apply(void (*fn)(T*&,void*), void* d);
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(T* a, void* d);
void yourfun(const T* a, void* d);
void yourfun(T*& a, void* d);
```

for reference semantics. Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains (const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is true, returns `true`. Otherwise, returns `false`.

```
bool
contains (bool (*fn)(const T*, void*), void *d) const;
bool
contains(bool (*fn)(T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void *d)
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
T*
find(const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
T*
find(bool (*fn)( T*,void*), void* d) const;
T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
const_reference
first() const;
```

Returns a reference to the first element of self. If the collection is empty, the function throws an exception of type *RW**BoundsErr***.

```
size_type
index(const T* a) const;
```

Returns the position of the first item `t` in self such that `(*t == *a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(T*,void*), void* d) const;
size_type
index(bool (*fn)(const T*,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true,` or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, T* a);
```

Inserts the item `a` in front of the item at position `i` in self. This position must be between zero and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
T*&
last();
```

```
T* const &
last() const;
```

Returns a reference to the last element of self.

```
reference
maxElement();
```

```
const_reference
maxElement() const;
```

```
reference
minElement();
```

```
const_reference
minElement() const;
```

Returns a reference to the maximum or minimum element in self.

```
size_type
occurrencesOf(const T* a) const;
```

Returns the number of elements `t` in self such that the expression `(*t == *a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(T*,void*), void* d) const;
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
popBack();
```

Removes and returns the last item in the collection.

```
T*
popFront();
```

Removes and returns the first item in the collection.

```
void
prepend(T* a);
```

Adds the item a to the beginning of the collection.

```
void
pushBack(T* a);
```

Adds the item a to the end of the collection.

```
void
pushFront(T* a);
```

Adds the item a to the beginning of the collection.

```
T*
remove(const T* a);
```

Removes and returns the first element t in self such that the expression (*t == *a) is true. Returns rwnil if there is no such element.

```
T*
remove(bool (*fn)(T*, void*), void* d);
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element t in self such that the expression ((*fn)(t,d)) is true. Returns rwnil if there is no such element. fn points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter d.

```
size_type const T*
removeAll(const_reference a);
```

Removes all elements t in self such that the expression (*t == *a) is true. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(T*,void*), void* d);
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements t in self such that the expression ((*fn)(t,d))is true. Returns the number of items removed. fn points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter d.

```
T*
removeAt(size_type i);
```
Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RWBoundsErr*.

```
T*
removeFirst();
```
Removes and returns the first item in the collection.

```
T*
removeLast();
```
Removes and returns the first item in the collection.

```
size_type
replaceAll(const T* oldVal, T* newVal);
```
Replaces with `newVal` all elements `t` in self such that the expression `(*t == *oldVal)` is `true`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(T*, void*), void* x, T* newVal);
size_type
replaceAll(bool (*fn)(const T*, void*), void* x,
           const T* newVal);
```
Replaces with `newVal` all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
sort();
```
Sorts the collection using the less-than operator to compare elements. Elements are dereferenced before being compared.

```
deque<T*, allocator>&
std();
const deque<T*, allocator>&
std() const;
```
Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Static Public Data Member

```
size_type  npos;
```
This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrDeque<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrDeque<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrDeque<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrDeque<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrDeque<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrDeque<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tpdlist.h>
RWTPtrDlist<T> dlist;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTPtrDlist* described in Appendix A.

Description

This class maintains a pointer-based collection of values, implemented as a doubly-linked list. Class ***T*** is the type pointed to by the items in the collection.

Persistence

Isomorphic

Example

In this example, a pointer-based doubly-linked list of user type `Dog` is exercised.

```
//
// tpdlist.cpp
//
#include <rw/tpdlist.h>
#include <iostream.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) {
   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; }

  // Order alphabetically by name:
  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
```

Related Classes

Classes *RWTPtrDeque<T>*, *RWTPtrSlist<T>*, and *RWTPtrOrderedVector<T>* also provide a Rogue Wave pointer-based interface to C++-standard sequence collections.

Class ***list<T*, allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef list<T*, allocator>                  container_type;
typedef container_type::size_type            size_type;
typedef container_type::difference_type      difference_type;
typedef container_type::iterator             iterator;
typedef container_type::const_iterator       const_iterator;
typedef T*                                   value_type;
typedef
typedef T*                                   reference;
typedef T* const&                            const_reference;
```

Public Constructors

```
RWTPtrDlist<T>();
```

Constructs an empty, doubly-linked list.

```
RWTPtrDlist<T>(const RWTPtrDlist<T>& rwlst);
```

Copy constructor.

```
RWTPtrDlist<T>(const list<T*, allocator>& lst);
```

Constructs a pointer based doubly linked list by copying all elements of lst.

```
RWTPtrDlist<T>(size_type n, T* a=0);
```

Constructs a doubly-linked list with n elements, each initialized to a.

```
RWTPtrDlist<T>(T*const* first, T*const* last);
```

Constructs a doubly-linked list by copying elements from the array of T*s pointed to by first, up to, but not including, the element pointed to by last.

Public Member Operators

```
RWTPtrDlist<T>&
operator=(const list<T*, allocator>& lst);
RWTPtrDlist<T>&
operator=(const RWTPtrDlist<T>& lst);
```

Clears all elements of self and replaces them by copying all elements of lst.

```
bool
operator<(const RWTPtrDlist<T>& lst);
```

Returns true if self compares lexicographically less than lst, otherwise returns false. Items in each collection are dereferenced before being compared. Assumes that type T has well-defined less-than semantics.

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

Returns true if self compares equal to lst, otherwise returns false. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed.*

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(T* a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(T*,void*), void* d);
void
apply(void (*fn)(T*&,void*), void* d);
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. self function must have one of the prototypes:

```
void yourfun(T* a, void* d);
void yourfun(const T* a, void* d);
void yourfun(reference a, void* d);
```

Client data may be passed through parameter `d`.

```
const
const_reference
at (size_type i);
reference
at(size_type i);
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use self method if multiple pointers to the same object are stored.

```
bool
contains(const T* a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(*t == *a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(T*,void*), void* d) const;
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

for the `const` version. Client data may be passed through parameter `d.`

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
T*
find(const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
T*
find(bool (*fn)(T*,void*), void* d) const;
T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

for the `const` version. Client data may be passed through parameter `d`.

```
reference
first();
```

```
const_reference
first() const;
```

Returns a reference to the first element of self.

```
T*
get();
```

Removes and returns the first element in the collection.

```
size_type
index(const T* a) const;
```

Returns the position of the first item `t` in self such that `(*t == *a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(T*,void*), void* d) const;
size_type
index(bool (*fn)(const T*,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true,` or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

for the `const` version. Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, T* a);
```

Inserts the item `a` in front of the item at position `i` in self. self position must be between zero and the number of entries in the collection, otherwise the function throws an exception of type *RW***BoundsErr**.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
T*&
last();
```

```
T*const&
last() const;
```

Returns a reference to the last item in the collection.

```
reference
maxElement();
```

```
const_reference
maxElement() const;
```

```
reference
minElement();
```

```
const_reference
minElement() const;
```

Returns a reference to the maximum or minimum element in self.

```
size_type
occurrencesOf(const T* a) const;
```

Returns the number of elements `t` in self such that the expression `(*t == *a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)( T*,void*), void* d) const;
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

for the `const` version. Client data may be passed through parameter `d`.

```
void
prepend(T* a);
```

Adds the item `a` to the beginning of the collection.

```
T*
remove(const T* a);
```

Removes and returns the first element `t` in self such that the expression `(*t == *a)` is `true`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)( T*,void*), void* d);
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self such that the expression `(*t == *a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)( T*,void*), void* d);
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

for the `const` version. Client data may be passed through parameter `d`.

```
T*
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. self position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T*
removeFirst();
```

Removes and returns the first item in the collection.

```
T*
removeLast();
```

Removes and returns the first item in the collection.

```
size_type
replaceAll(const T* oldVal,T* newVal);
```

Replaces with `newVal` all elements `t` in self such that the expression `(*t == *oldVal)` is `true`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(T*, void*),void* d,T* newVal);
size_type
replaceAll(bool (*fn)(const T*, void*),void* d,T* newVal);
```

Replaces with `newVal` all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
sort();
```

Sorts the collection using the less-than operator to compare elements. Elements are dereferenced before being compared.

```
list<T*, allocator>&
std();
const list<T*, allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrDlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrDlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrDlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrDlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrDlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrDlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpdlist.h>
RWTPtrDlist<T> dl;
RWTPtrDlistIterator<T> itr(dl);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTPtrDlistIterator* described in Appendix A.

Description

RWTPtrDlistIterator provides an iterator interface to the Tools 7 Standard C++ Library-based collections which is compatible with the iterator interface provided for the *Tools.h++* 6.xcontainers.

The order of iteration over an ***RWTPtrDlist*** is dependent on the order of the values in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called. For `operator--`, decrementing past the first element will return a value equivalent to `false`.

Persistence

None

Examples

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

int main(){
   RWTPtrDlist<RWCString> a;
   RWTPtrDlistIterator<RWCString> itr(a);
   a.insert(new RWCString("John"));
   a.insert(new RWCString("Steve"));
   a.insert(new RWCString("Mark"));
   a.insert(new RWCString("Steve"));
   for(;itr();)
     cout << *itr.key() <<endl;
   return 0;
}
```

Program Output
```
John
Steve
Mark
Steve
```

Public Constructors

`RWTPtrDlistIterator<T>``(RWTPtrDlist<T>& l);`

Creates an iterator for the list `l`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

`T*`
`operator()``();`

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a nil pointer equivalent to boolean `false`.

`RWBoolean`
`operator++``();`

Advances self to the next element. If the iterator has been reset or just created, self will reference the first element. If, before iteration, self referenced the last value in the list, self will now referece an undefined value distinct from the reset value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

`RWBoolean`
`operator+=``(size_type n);`

Behaves as if `operator++()` had been applied `n` times

`RWBoolean`
`operator--``();`

Moves self back to the immediately previous element. If the iterator has been reset or just created, self operator will returna value equivalent to `false`, otherwise it will return a value equivalent to `true`. If self references the the first element, it will now be in the reset state. If self has been iterated past the last value in the list, it will now reference the last item in the list. Note: no post-decrement operator is provided.

`RWBoolean`
`operator-=``(size_type n);`

Behaves as if `operator–()` had been applied `n` times

Public Member Functions

`RWTPtrDlist<T>*`
`container()`` const;`

Returns a pointer to the collection being iterated over.

```
T*
findNext(const T* a);
```

Returns the first element `t` encountered while iterating self forward, such that the expression `(*t == *a)` is `true`. If no such element exists, returns a nil pointer equivalent to false. Leaves self referencing the found item, or “past the end.”

```
T*
findNext(RWBoolean(*fn)(T*, void*), void* d);
```

Returns the first element `t` encountered by iterating self forward such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`. If no such element exists, returns a nil pointer equivalent to false. Leaves self referencing the found item, or “past the end.”

```
void
insertAfterPoint(T* p);
```

Inserts the pointer `p` into the container directly after the element referenced by self.

```
T*
key();
```

Returns the stored value referenced by self. Undefined if self is not referencing a value within the list.

```
T*
remove();
```

Returns the stored value referenced by self and removes it from the collection. Undefined if self is not referencing a value within the list.

```
T*
removeNext(const T*);
```

Returns and removes the first element `t`, encountered by iterating self forward, such that the expression `(*t == *a)` is `true`. If no such element exists, returns `nil`.

```
T*
removeNext(RWBoolean(*fn)(T*, void*), void* d);
```

Returns and removes the first element `t,` encountered by iterating self forward, such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`. If no such element exists, returns `nil`.

```
void
reset();
void
reset(RWTPtrDlist<T>& l*);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset` with no argument will reset the iterator on the current container. Supplying `RWTPtrDlist<T>` to `reset()` will reset the iterator on the new container.

Synopsis

```
#define RWTPtrHashDictionary RWTPtrHashMap
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTPtrHashMap*. Although the old name (*RWTPtrHashDictionary*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTPtrHashDictionary* in Appendix A.

Synopsis

```
#define RWTPtrHashDictionaryIterator RWTPtrHashMapIterator
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTPtrHashMapIterator*. Although the old name (*RWTPtrHashDictionaryIterator*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTPtrHashDictionaryIterator* in Appendix A.

Synopsis

```
#include <rw/tphdict.h>
RWTPtrHashMap<K,T,H,EQ> m;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTPtrHashDictionary* described in Appendix A.

Description

This class maintains a pointer-based collection of associations of type `pair<K* const, T*>`. These pairs are stored according to a hash object of type `H`. `H` must provide a hash function on elements of type `K` via a public member

```
unsigned long operator()(const K& x)
```

Equivalent keys within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const K& x, const K& y)
```

which should return `true` if `x` and `y` are equivalent.

RWTPtrHashMap<K,T,H,EQ> will not accept a key that compares equal to any key already in the collection. (*RWTPtrHashMultiMap<K,T,H,EQ>* may contain multiple keys that compare equal to each other.) Equality is based on the comparison object and *not* on the `==` operator.

Persistence

Isomorphic

Examples

```
//
// tphmap.cpp
//
#include<rw/tphdict.h>
#include<rw/cstring.h>
#include<iostream.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x(0); }
};
int main(){
   RWCString snd = "Second";
   RWTPtrHashMap<RWCString,int,silly_hash,equal_to<RWCString> >
      contest;
   contest.insert(new RWCString("First"), new int(7));
   contest.insert(&snd,new int(3));
```

```
   //duplicate insertion rejected
   contest.insert(&snd,new int(6));
   contest.insert(new RWCString("Third"), new int(2));
   cout << "There was "
        << contest.occurrencesOf(new RWCString("Second"))
        << " second place winner." << endl;
   return 0;
}
```

Program Output:

```
There was 1 second place winner.
```

Related Classes

Class *RWTPtrHashMultiMap<K,T,H,EQ>* offers the same interface to a pointer-based collection that accepts multiple keys that compare equal to each other.

Class *rw_hashmap<K*,T*,rw_deref_hash<H,K>,rw_deref_compare<C,K> >* is the C++-standard library style collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef rw_deref_hash<H,K>                       container_hash;
typedef rw_deref_compare<EQ,K>                   container_eq;
typedef rw_hashmap<K*,T*,container_hash,container_eq >
                                                 container_type;
typedef container_type::size_type                size_type;
typedef container_type::difference_type          difference_type;
typedef container_type::iterator                 iterator;
typedef container_type::const_iterator           const_iterator;
typedef pair <K* const, T*>                      value_type;
typedef pair <K* const, T*>&                     reference;
typedef const pair <K* const, T*>&               const_reference;
typedef K*                                       value_type_key;
typedef T*                                       value_type_data;
typedef K*&                                      reference_key;
typedef T*&                                      reference_data;
typedef const K*const&                           const_reference_key;
typedef const T*const&                           const_reference_data;
```

Public Constructors

RWTPtrHashMap<K,T,H,EQ>`();`

Constructs an empty map.

RWTPtrHashMap<K,T,H,EQ>`(const RWTPtrHashMap<K,T,H,EQ>& rwm);`

Copy constructor.

RWTPtrHashMap<K,T,H,EQ>
`(const container_type & m);`

Constructs a pointer based hash map by copying all elements from `m`.

RWTPtrHashMap<K,T,H,EQ>
`(const H& h, size_type sz = RWDEFAULT_CAPACITY);`

This *Tools.h++* 6.x style constructor creates an empty hashed map which uses the hash object `h` and has an initial capacity of `sz`.

```
RWTPtrHashMap<K,T,H,EQ>
(const value_type* first,value_type* last);
```

Constructs a map by copying elements from the array of `pair`s pointed to by `first`, up to, but not including, the pair pointed to by `last`.

Public Member Operators

```
RWTPtrHashMap<K,T,H,EQ>&
operator=(const container_type& m);
RWTPtrHashMap<K,T,H,EQ>&
operator=(const RWTPtrHashMap<K,T,H,EQ>& m);
```

Destroys all associations in self and replaces them by copying all associations from `m`.

```
bool
operator==(const RWTPtrHashMap<K,T,H,EQ>& m) const;
```

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual keys that compare equal to each other. Keys are dereferenced before being compared.

```
T*&
operator[](K* key);
```

Looks up `key` and returns a reference to its associated item. If the key is not in the dictionary, then it will be added with an associated uninitialized pointer of type `T*`. Because of this, if there is a possibility that a key will not be in the dictionary, then this operator should only be used as an lvalue.

Public Member Functions

```
void
apply(void (*fn)(const K*, T*&,void*),void* d);
void
apply(void (*fn)(const K*,const T*,void*),void* d) const;
```

Applies the user-defined function pointed to by `fn` to every association in the collection. self function must have one of the prototypes:

```
void yourfun(const K* key, T*& a, void* d);
void yourfun(const K* key, const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K*, T*&,void*),void* d);
void
applyToKeyAndValue
(void (*fn)(const K*, const T*, void*), void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply**.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
size_type
capacity() const;
```

Returns the number of buckets(slots) available in the underlying hash representation. See **resize** below.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all associations from the collection *and* uses `operator delete` to destroy the objects pointed to by the keys and their associated items. Do not use self method if multiple pointers to the same object are stored. (If the equality operator is reflexive, the container cannot hold such multiple pointers.)

```
bool
contains(const K* key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `*key`, otherwise returns `false`.

```
bool
contains
(bool (*fn)(value_type,void*),void* d) const;
```

Returns `true` if there exists an association a in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
float
fillRatio() const;
```
Returns the ratio `entries()`/`capacity()`.

```
const K*
find(const K* key) const;
```
If there exists a key `j` in self that compares equal to `*key`, then `j` is returned. Otherwise, returns `rwnil`.

```
value_type
find(bool (*fn)(value_type,void*), void* d) const;
```
If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, then returns `a`. Otherwise, returns `pair<rwnil,rwnil>`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
findValue(const K* key);
const T*
findValue(const K* key) const;
```
If there exists a key `j` in self that compares equal to `*key`, returns the item associated with `j`. Otherwise, returns `rwnil`.

```
const K*
findKeyAndValue(const K* key, T*& tr);
const K*
findKeyAndValue(const K* key, const T*& tr) const;
```
If there exists a key `j` in self that compares equal to `*key`, assigns the item associated with `j` to `tr`, and returns `j`. Otherwise, returns `rwnil` and leaves the value of `tr` unchanged.

```
bool
insert(K* key, T* a);
```
Adds `key` with associated item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an association with the equivalent key.

```
bool
insertKeyAndValue(K* key,T* a);
```
This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert**.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K* key) const;
```
Returns the number of keys `j` in self that compare equal to `*key`.

```
size_type
occurrencesOf
(bool (*fn)(value_type,void*),void* d) const;
```
Returns the number of associations `a` in self such that the expression`((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
K*
remove(const K* key);
```
Removes the first association with key `j` in self that compares equal to `*key` and returns `j`. Returns `rwnil` if there is no such association.

```
K*
remove(bool (*fn)(value_type,void*), void* d);
```
Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns its key. Returns `rwnil` if there is no such association. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const K* key);
```
Removes all associations with key `j` in self that compare equal to `*key`. Returns the number of associations removed.

```
size_type
removeAll(bool (*fn)(value_type,void*), void* d);
```
Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```

Changes the capacity of self by creating a new hashed map with a capacity of `sz`. **resize** copies every element of self into the new container and finally swaps the internal representation of the new container with the internal representation of `self`.

```
rw_hashmap<K*,T*,rw_deref_hash<H,K>,deref_compare<EQ,K>>&
std();
const rw_hashmap<K*,T*,rw_deref_hash<H,K>,deref_compare<EQ,K>>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
          const RWTPtrHashMap<K,T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrHashMap<K,T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrHashMap<K,T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrHashMap<K,T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrHashMap<K,T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrHashMap<K,T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tphdict.h>
RWTPtrHashMap<K,T,H,EQ> m;
RWTPtrHashMap<K,T,H,EQ> itr(m);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTPtrHashDictionaryIterator* described in Appendix A.

Description

RWTPtrHashMapIterator is supplied with *Tools.h++* 7.x to provide an iterator interface to the Standard Library based collections that has backward compatibility with the container iterators provided in *Tools.h++* 6.x.

Iteration over an ***RWTPtrHashMap*** is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Once this state is reached, continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

struct silly_h{
   unsigned long operator()(RWCString x) const
     { return x.length() * (long)x(0); }
};

int main(){
   RWTPtrHashMap
     <RWCString,int,silly_h,equal_to<RWCString> > age;

   RWTPtrHashMapIterator
     <RWCString,int,silly_h,equal_to<RWCString> > itr(age);

   age.insert(new RWCString("John"),new int(30));
   age.insert(new RWCString("Steve"),new int(17));
```

```
   age.insert(new RWCString("Mark"),new int(24));

//Duplicate insertion is rejected
   age.insert(new RWCString("Steve"),new int(24));

   for(;++itr;)
     cout << *itr.key() << "\'s age is " << *itr.value() << endl;

   return 0;
}
```

Program Output (not necessarily in this order)

```
John's age is 30
Mark's age is 24
Steve's age is 17
```

Public Constructors

RWTPtrHashMapIterator<K,T,H,EQ>(RWTPtrHashMap<K,T,H,EQ>&h);

Creates an iterator for the hashed map h. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

K*
operator()();

Advances self to the next element, dereferences the resulting iterator and returns its key. If the iterator has advanced past the last item in the container, the element returned will be a nil pointer equivalent to boolean false.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multi-map, self will now reference an undefined value and a value equivalent to false will be returned. Otherwise, a value equivalent to true is returned. Note: no post-increment operator is provided.

Public Member Functions

RWTPtrHashMap<K,T,H,EQ>*
container() const;

Returns a pointer to the collection being iterated over.

K*
key() const;

Returns the key portion of the association currently referenced by self. Undefined if self is not referencing a value within the map.

```
void
reset();
void
reset(RWTPtrHashMap<K,T,H,EQ>& h);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a hashed map with `reset()` will reset the iterator on that container.

```
T*
value();
```

Returns the value portion of the association pointed to by self. The behavior is undefined if the map is empty.

Synopsis

```
#include <rw/tphmmap.h>
RWTPtrHashMultiMap<K,T,H,EQ> m;
```

Standard C++ Library Dependent!

***RWTPtrHashMultiMap* requires the Standard C++ Library.**

Description

This class maintains a pointer-based collection of associatoins of type `pair<K* const, T*>`. These pairs are stored according to a hash object of type `H`. `H` must provide a hash function on elements of type `K` via a public member

```
unsigned long operator()(const K& x)
```

Equivalent keys within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const K& x, const K& y)
```

which should return `true` if `x` and `y` are equivalent.

RWTPtrHashMultiMap<K,T,H,EQ> may contain multiple keys that compare equal to each other. (***RWTPtrHashMap<K,T,H,EQ>*** will not accept a key that compares equal to any key already in the collection.) Equality is based on the comparison object and *not* on the `==` operator.

Persistence

Isomorphic

Examples

```
//
// tphmap.cpp
//
#include<rw/tphmmap.h>
#include<rw/cstring.h>
#include<iostream.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x[0]; }
};
int main(){
  RWCString snd = "Second";
  RWTPtrHashMultiMap<RWCString,int,silly_hash,equal_to<RWCString> >
      contest;
  contest.insert(new RWCString("First"), new int(7));
  contest.insert(&snd, new int(3));
  contest.insert(&snd, new int(6));      // duplicate key OK
  contest.insert(new RWCString("Third"), new int(2));
```

```
  cout << "There were " << contest.occurrencesOf(&snd)
       << " second place winners." << endl;

  return 0;
}
```

Program Output:

```
There were 2 second place winners.
```

Related Classes

Class *RWTPtrHashMap<K,T,H,EQ>* offers the same interface to a pointer-based collection that will not accept multiple keys that compare equal to each other.

rw_hashmultimap<<K,T*>,rw_deref_hash<H,K>,rw_deref_compare<EQ,K> >* is the C++-standard style collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef rw_deref_hash<H,K>                   container_hash;
typedef rw_deref_compare<EQ,K>               container_eq;
typedef rw_hashmultimap<K*,T*,container_hash,container_eq>
                                             container_type;
typedef container_type::size_type            size_type;
typedef container_type::difference_type      difference_type;
typedef container_type::iterator             iterator;
typedef container_type::const_iterator       const_iterator;
typedef pair <K* const, T*>                  value_type;
typedef pair <K* const, T*>&                 reference;
typedef const pair <K* const, T*>&           const_reference;
typedef K*                                   value_type_key;
typedef T*                                   value_type_data;
typedef K*&                                  reference_key;
typedef T*&                                  reference_data;
typedef const K*const&                       const_reference_key;
typedef const T*const&                       const_reference_data;
```

Public Constructors

RWTPtrHashMultiMap<K,T,H,EQ>();

Constructs an empty map.

RWTPtrHashMultiMap<K,T,H,EQ>(const container_type& m);

Constructs a multi-map by doing an element by element copy from the C++ Standard Library style hashed multi-map, `m`.

RWTPtrHashMultiMap<K,T,H,EQ>
(const RWTPtrHashMultiMap<K,T,H,EQ>& rwm);

Copy constructor.

RWTPtrHashMultiMap<K,T,H,EQ>
(value_type* first, value_type* last);

Constructs a map by copying elements from the array of `pair`s pointed to by `first`, up to, but not including, the pair pointed to by `last`.

```
RWTPtrHashMultiMap<K,T,H,EQ>
(const H& h, size_type sz = RWDEFAULT_CAPACITY);
```

This *Tools.h++* 6.x style constructor creates an empty hashed multi-map which uses the hash object `h` and has an initial capacity of `sz`.

Public Member Operators

```
RWTPtrHashMultiMap<K,T,H,EQ>&
operator=(const container_type&jjj m);
RWTPtrHashMultiMap<K,T,H,EQ>&
operator=(const RWTPtrHashMultiMap<K,T,H,EQ>& m);
```

Destroys all associations in self and replaces them by copying all associations from `m`.

```
bool
operator==(const RWTPtrHashMultiMap<K,T,H,EQ>& m);
```

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual keys that compare equal to each other. Keys are dereferenced before being compared.

Public Member Functions

```
void
apply(void (*fn)(const K*, T*&,void*),void* d);
void
apply(void (*fn)(const K*, const T*, void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every association in the collection. self function must have one of the prototypes:

```
void yourfun(const K* key, T*& a, void* d);
void yourfun(const K* key, const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K*, T*&,void*),void* d);
void
applyToKeyAndValue
(void (*fn)(const K*, const T*, void*), void* d) const;
```

This is a deprecated version of the `apply` member above. It behaves exactly the same as `apply`.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
size_type
capacity() const;
```

Returns the number of buckets (slots) available in the underlying hash representation. See `resize` below.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all associations from the collection *and* uses `operator delete` to destroy the objects pointed to by the keys and their associated items. Do not use self method if multiple pointers to the same keys or items are stored.

```
bool
contains(const K* key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `*key`, otherwise returns `false`.

```
bool
contains(bool (*fn)(value_type,void*),void* d) const;
```

Returns `true` if there exists an association a in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type* a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
float
fillRatio() const;
```

Returns the ratio `entries()`/`capacity()`.

```
const K*
find(const K* key) const;
```

If there exists a key `j` in self that compares equal to `*key`, then `j` is returned. Otherwise, returns `rwnil`.

```
value_type
find(bool (*fn)(value_type,void*), void* d) const;
```

If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, then returns `a`. Otherwise, returns `pair<rwnil,rwnil>`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
findValue(const K* key);
```

```
const T*
findValue(const K* key) const;
```

If there exists a key `j` in self that compares equal to `*key`, returns the item associated with `j`. Otherwise, returns `rwnil`.

```
const K*
findKeyAndValue(const K* key, T*& tr);
```

```
const K*
findKeyAndValue(const K* key, const T*& tr) const;
```

If there exists a key `j` in self that compares equal to `*key`, assigns the item associated with `j` to `tr`, and returns `j`. Otherwise, returns `rwnil` and leaves the value of `tr` unchanged.

```
bool
insert(K* key,T* a);
```

Adds `key` with associated item `a` to the collection. Returns `true`.

```
bool
insertKeyAndValue(K* key,T* a);
```

This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert**.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K* key) const;
```

Returns the number of keys `j` in self that compare equal to `*key`.

```
size_type
occurrencesOf
(bool(*fn)(value_type,void*),void* d)const;
```

Returns the number of associations `a` in self such that the expression`((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
K*
remove(const K* key);
```

Removes the first association with key `j` in self that compares equal to `*key`. Returns `rwnil` if there is no such association.

```
K*
remove(bool (*fn)(value_type,void*), void* d);
```

Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns its key. Returns `rwnil` if there is no such association. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const K* key);
```

Removes all associations with key `j` in self that compare equal to `*key`. Returns the number of associations removed.

```
size_type
removeAll(bool (*fn)(value_type,void*), void* d);
```

Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```

Changes the capacity of self by creating a new hashed multi-map with a capacity of `sz` . **resize** then copies every element of `self` into the new container and finally swaps the internal representation of the new container with `self`.

```
container_type&
std();
const container_type&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTPtrHashMultiMap<K,T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm,
       const RWTPtrHashMultiMap<K,T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm,
       RWTPtrHashMultiMap<K,T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm,
       RWTPtrHashMultiMap<K,T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm,
       RWTPtrHashMultiMap<K,T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm,
       RWTPtrHashMultiMap<K,T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tphmmap.h>
RWTPtrHashMultiMap<K,T,H,EQ> m;
RWTPtrHashMultiMap<K,T,H,EQ> itr(m);
```

Standard C++ Library Dependent!

RWTPtrHashMultiMapIterator **requires the Standard C++ Library.**

Description

RWTPtrHashMultiMapIterator is supplied with Tools 7 to provide an iterator interface to the new Standard Library based collections that has backward compatibility with the container iterators provided in Tools 6.

Iteration over an ***RWTPtrHashMultiMap*** is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used. The only useable relationship between consecutive elements is that elements which are defined to be equivalent by the equivalence object, `EQ`, will remain adjacent.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

struct silly_h{
   unsigned long operator()(RWCString x) const
     { return x.length() * (long)x(0); }
};

int main(){
   RWTPtrHashMultiMap
     <RWCString,int,silly_h,equal_to<RWCString> > age;

   RWTPtrHashMultiMapIterator
     <RWCString,int,silly_h,equal_to<RWCString> > itr(age);

   age.insert(new RWCString("John"),new int(30));
   age.insert(new RWCString("Steve"),new int(17));
```

```
   age.insert(new RWCString("Mark"),new int(24));
   age.insert(new RWCString("Steve"),new int(24));

   for(;++itr;)
     cout << *itr.key() << "\'s age is " << *itr.value() << endl;

   return 0;
}
```

Program Output (not necessarily in this order)

```
John's age is 30
Mark's age is 24
Steve's age is 24
Steve's age is 17
```

Public Constructors

RWTPtrHashMultiMapIterator<K,T,H,EQ>
`(RWTPtrHashMultiMap<K,T,H,EQ>&h);`

Creates an iterator for the hashed multi-map `h`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

`K*`
`operator()``();`

Advances self to the next element, dereferences the resulting iterator and returns its key. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

`RWBoolean`
`operator++``();`

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multi-map, self will now reference an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

`RWTPtrHashMultiMap<K,T,H,EQ>*`
`container()` `const;`

Returns a pointer to the collection being iterated over.

`K*`
`key``() const;`

Returns the key portion of the association currently referenced by self. Undefined if self is not referencing a value within the multimap.

```
void
reset();
void
reset(RWTPtrHashMultiMap<K,T,H,EQ>& h);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrHashMultiMap` to `reset()` will reset the iterator on that container.

```
T*
value();
```

Returns the value portion of the association referenced by self. Undefined if self is not valid.

Synopsis

```
#include <rw/tphasht.h>
RWTPtrHashMultiSet<T,H,EQ> hmset;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTPtrHashTable* described in Appendix A.

Description

This class maintains a pointer-based collection of values, which are stored according to a hash object of type `H`. Class ***T*** is the type pointed to by the items in the collection. `H` must provide a hash function on elements of type `T` via a public member

```
unsigned long operator()(const T& x)
```

Objects within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const T& x, const T& y)
```

which should return `true` if `x` and `y` are equivalent, `false` otherwise.

RWTPtrHashMultiSet<T,H,EQ> may contain multiple items that compare equal to each other. (***RWTPtrHashSet<T,H,EQ>*** will not accept an item that compares equal to an item already in the collection.)

Persistence

Isomorphic

Examples

```
//
// tphasht.cpp
//
#include <rw/tphasht.h>
#include <rw/cstring.h>
#include <iostream.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x(0); }
};

main(){
RWTPtrHashMultiSet<RWCString,silly_hash,equal_to<RWCString> > set1;
RWTPtrHashMultiSet<RWCString,silly_hash,equal_to<RWCString> > set2;

 set1.insert(new RWCString("one"));
 set1.insert(new RWCString("two"));
 set1.insert(new RWCString("three"));
```

```
 set1.insert(new RWCString("one"));  // OK: duplicates allowd

 cout << set1.entries() << endl;    // Prints "4"

 set2 = set1;
 cout << ((set1.isEquivalent(set2)) ? "TRUE" : "FALSE") << endl;
 // Prints "TRUE"

 set2.difference(set1);

 set1.clearAndDestroy();
 cout << set1.entries() << endl;    // Prints "0"
 cout << set2.entries() << endl;    // Prints "0"

 return 0;
}
```

Related Classes

Class *RWTPtrHashSet<T,H,EQ>* offers the same interface to a pointer-based collection that will not accept multiple items that compare equal to each other.

Class *rw_hashmultiset<T*,rw_deref_hash<H,T>,rw_deref_compare<EQ,T> >* is the C++-standard collection that serves as the underlying implementation for *RWTPtrHashMultiSet<T,H,EQ>*.

Public Typedefs

```
typedef rw_deref_compare<EQ,T>                          container_eq;
typedef rw_deref_hash<H,T>                              container_hash;

typedef rw_hashmultiset<T*,container_hash,container_eq>
                                                        container_type;
typedef container_type::size_type                       size_type;
typedef container_type::difference_type                 difference_type;
typedef container_type::iterator                        iterator;
typedef container_type::const_iterator                  const_iterator;
typedef T*                                              value_type;
typedef T* const&                                       reference;
typedef T* const&                                       const_reference;
```

Public Constructors

```
RWTPtrHashMultiSet<T,H,EQ>
(size_type sz=1024,const H& h = H(),const EQ& eq = EQ());
```

Constructs an empty multi set. The hash table representation used by self multi-set will have `sz` buckets, use `h` as a hashing function and `eq` to test for equality between stored elements.

```
RWTPtrHashMultiSet<T,H,EQ>
(const RWTPtrHashMultiSet<T,H,EQ>& rws);
```

Copy constructor.

```
RWTPtrHashMultiSet<T,H,EQ>
(const rw_hashmultiset<T*,container_hash, container_eq>& s);
```

Constructs a hashed multi-set, copying all element from `s`.

RWTPtrHashMultiSet<T,H,EQ>
`(const H& h,size_type sz = RWDEFAULT_CAPACITY);`

This *Tools.h++* 6.xstyle constructor creates an empty hashed multi-set which uses the hash object `h` and has an initial hash table capacity of `sz`.

RWTPtrHashMultiSet<T,H,EQ>`(T*const* first,T*const* last, size_type sz=1024,const H& h = H(),const EQ& eq = EQ());`

Constructs a set by copying elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`. The hash table representation used by self multi-set will have `sz` buckets, use `h` as a hashing function and `eq` to test for equality between stored elements.

Public Member Operators

`RWTPtrHashMultiSet<T,H,EQ>&`
operator=`(const RWTPtrHashMultiSet<T,H,EQ>& s);`

Clears all elements of self and replaces them by copying all elements of `s`.

`bool`
operator==`(const RWTPtrHashMultiSet<T,H,EQ>& s) const;`

Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

Public Member Functions

`void`
apply`(void (*fn)(const T*,void*), void* d) const;`

Applies the user-defined function pointed to by `fn` to every item in the collection. self function must have prototype:

```
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter `d`.

`iterator`
begin`();`

`const_iterator`
begin`() const;`

Returns an iterator positioned at the first element of self.

`size_type`
capacity`() const;`

Returns the number of buckets(slots) available in the underlying hash representation. See `resize` below.

`void`
clear`();`

Clears the collection by removing all items from self.

void
clearAndDestroy();

Removes all items from the collection *and* uses operator delete to destroy the objects pointed to by those items. Do not use self method if multiple pointers to the same object are stored.

bool
contains(const T* a) const;

Returns true if there exists an element t in self that compares equal to *a, otherwise returns false.

bool
contains(bool (*fn)(const T*,void*), void* d) const;

Returns true if there exists an element t in self such that the expression ((*fn)(t,d)) is true, otherwise returns false. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter d.

void
difference(const RWTPtrHashMultiSet<T,H,EQ>& s);

Sets self to the set-theoretic difference given by (self - s). Elements from each set are dereferenced before being compared.

iterator
end();

const_iterator
end() const;

Returns an iterator positioned "just past" the last element in self.

size_type
entries() const;

Returns the number of items in self.

float
fillRatio() const;

Returns the ratio entries()/capacity().

const T*
find(const T* a) const;

If there exists an element t in self that compares equal to *a, returns t. Otherwise, returns rwnil.

const T*
find(bool (*fn)(const T*,void*), void* d) const;

If there exists an element t in self such that the expression ((*fn)(t,d)) is true, returns t. Otherwise, returns rwnil. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter d.

bool
insert(T* a);
Adds the item a to the collection. Returns true.

void
intersection(const RWTPtrHashMultiSet<T,H,EQ>& s);
Destructively performs a set theoretic intersection of self and s, replacing the contents of self with the result.

bool
isEmpty() const;
Returns true if there are no items in the collection, false otherwise.

bool
isEquivalent(const RWTPtrHashMultiSet<T,H,EQ>& s) const;
Returns true if there is set equivalence between self and s; returns false otherwise.

bool
isProperSubsetOf(const RWTPtrHashMultiSet<T,H,EQ>& s) const;
Returns true if self is a proper subset of s; returns false otherwise.

bool
isSubsetOf(const RWTPtrHashMultiSet<T,H,EQ>& s) const;
Returns true if self is a subset of s or if self is set equivalent to s, false otherwise.

size_type
occurrencesOf(const T* a) const;
Returns the number of elements t in self that compare equal to *a.

size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
Returns the number of elements t in self such that the expression((*fn)(t,d)) is true. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter d.

T*
remove(const T* a);
Removes and returns the first element t in self that compares equal to *a. Returns rwnil if there is no such element.

```
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self that compare equal to `*a`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```

Changes the capacity of self by creating a new hashed multi-set with a capacity of `sz` . `resize` copies every element of self into the new container and finally swaps the internal representation of the new container with the internal representation of `self`.

```
rw_hashset<T*,container_hash,container_eq>&
std();
const rw_hashset<T*,container_hash,container_eq>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

```
void
symmetricDifference(const RWTPtrHashMultiSet<T,H,EQ>& rhs);
```

Destructively performs a set theoretic symmetric difference operation on self and `rhs`. Self is replaced by the result. A symmetric difference can be informally defined as $(A \cup B)-(A \cap B)$.

```
void
Union(const RWTPtrHashMultiSet<T,H,EQ>& rhs);
```

Destructively performs a set theoretic union operation on self and `rhs`. Self is replaced by the result. Note the uppercase “U” in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTPtrHashMultiSet<T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm,
       const RWTPtrHashMultiSet<T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm,
        RWTPtrHashMultiSet<T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm,
        RWTPtrHashMultiSet<T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm,
        RWTPtrHashMultiSet<T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm,
         RWTPtrHashMultiSet<T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tphasht.h>
RWTPtrHashMultiSet<T,H,EQ> m;
RWTPtrHashMultiSet<T,H,EQ> itr(m);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTPtrHashTableIterator* described in Appendix A.

Description

RWTPtrHashMultiSetIterator is supplied with *Tools.h++* 7.x to provide an iterator interface to the Standard Library based collections that has backward compatibility with the container iterators provided in *Tools.h++* 6.x.

Iteration over an ***RWTPtrHashMultiSet*** is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used. The only useable relationship between consecutive elements is that all elements which are defined to be equivalent by the equivalence object, `EQ`, will remain adjacent.

The current item referenced by this iterator is undefined after construction or after a call to `reset()` operation. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

struct silly_h{
   unsigned long operator()(RWCString x) const
     { return x.length() * (long)x(0); }
};

int main(){
   RWTPtrHashMultiSet<RWCString,silly_h,equal_to<RWCString> > age;

   RWTPtrHashMultiSetIterator
   <RWCString,silly_h,equal_to<RWCString> > itr(age);

   age.insert(new RWCString("John"));
```

```
   age.insert(new RWCString("Steve"));
   age.insert(new RWCString("Mark"));
   age.insert(new RWCString("Steve"));

   for(;++itr;)
     cout << *itr.key() << endl;

   return 0;
}
```

Program Output (not necessarily in this order)

```
John
Mark
Steve
Steve
```

Public Constructors

RWTPtrHashMultiSetIterator<T,H,EQ>
`(RWTPtrHashMultiSet<T,H,EQ>&h);`

Creates an iterator for the hashed multi-set `h`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

`T*`
operator()`();`

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

`RWBoolean`
operator++`();`

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multiset, self will now reference an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

`RWTPtrHashMultiSet<T,H,EQ>*`
container() `const;`

Returns a pointer to the collection being iterated over.

`T*`
key`() const;`

Returns the value currently referenced by `self`. Undefined if self is not referencing a value within the multiset.

```
void
reset();
void
reset(RWTPtrHashMultiSet<T,H,EQ>& h);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrHashMultiSet` to `reset()` will reset the iterator on that container.

Synopsis

```
#include <rw/tphset.h>
RWTPtrHashSet<T,H,EQ> s;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTPtrHashSet* described in Appendix A.

Description

This class maintains a pointer-based collection of values, which are stored according to a hash object of type `H`. Class ***T*** is the type pointed to by the items in the collection. `H` must provide a hash function on elements of type `T` via a public member

```
unsigned long operator()(const T& x)
```

Objects within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const T& x, const T& y)
```

which should return `true` if `x` and `y` are equivalent, `false` otherwise.

RWTPtrHashSet<T,H,EQ> will not accept an item that compares equal to an item already in the collection. (***RWTPtrHashMultiSet<T,H,EQ>*** may contain multiple items that compare equal to each other.) Equality is based on the equality object and *not* on the `==` operator.

Persistence

Isomorphic

Example

```
//
// tphset2.cpp
//
#include <rw/tphset.h>
#include <rw/cstring.h>
#include <iostream.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x(0); }
};

main(){
RWTPtrHashSet<RWCString,silly_hash,equal_to<RWCString> > set1;
RWTPtrHashSet<RWCString,silly_hash,equal_to<RWCString> > set2;

 set1.insert(new RWCString("one"));
```

```
 set1.insert(new RWCString("two"));
 set1.insert(new RWCString("three"));
 set1.insert(new RWCString("one")); // Duplicate insertion rejected

 cout << set1.entries() << endl;   // Prints "3"

 set2 = set1;
 cout << ((set1.isEquivalent(set2)) ? "TRUE" : "FALSE") << endl;
 // Prints "TRUE"

 set2.difference(set1);

 set1.clearAndDestroy();
 cout << set1.entries() << endl;    // Prints "0"
 cout << set2.entries() << endl;    // Prints "0"

 return 0;
}
```

Related Classes

Class *RWTPtrHashMultiSet<T,H,EQ>* offers the same interface to a pointer-based collection that accepts multiple items that compare equal to each other.

Class *rw_hashset<T*,rw_deref_hash<H,T>, rw_deref_compare<EQ,T> >* is the C++-standard collection that serves as the underlying implementation for *RWTPtrHashSet<T,H,EQ>*.

Public Typedefs

```
typedef rw_deref_compare<EQ,T>                      container_eq;
typedef rw_deref_hash<H,T>                          container_hash;
typedef rw_hashset<T*,container_hash, container_eq>
                                                    container_type;
typedef container_type::size_type                   size_type;
typedef container_type::difference_type             difference_type;
typedef container_type::iterator                    iterator;
typedef container_type::const_iterator              const_iterator;
typedef T*                                          value_type;
typedef T* const&                                   reference;
typedef T* const&                                   const_reference;
```

Public Constructors

`RWTPtrHashSet<T,H,EQ>`
`(size_type sz=1024,const H& h = H(),const EQ& eq = EQ());`

Constructs an empty hashed set. The underlying hash table representation will have `sz` buckets, will use `h` for its hashing function and will use `eq` to determine equality between elements.

`RWTPtrHashSet<T,H,EQ>``(const RWTPtrHashSet<T,H,EQ>& rws);`

Copy constructor.

`RWTPtrHashSet<T,H,EQ>`
`(const H& h,size_type sz = RWDEFAULT_CAPACITY);`

This *Tools.h++* 6.xstyle constructor creates an empty hashed set which uses the hash object `h` and has an initial hash table capacity of `sz`.

```
RWTPtrHashSet<T,H,EQ>
(const rw_hashset<T*,container_hash,container_eq>& s);
```

Constructs a pointer based hash set by copying all elements from `s`.

```
RWTPtrHashSet<T,H,EQ>(T*const* first,T*const* last,
size_type sz=1024,const H& h = H(),const EQ& eq = EQ());
```

Constructs a set by copying elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`. The underlying hash table representation will have `sz` buckets, will use `h` for its hashing function and will use `eq` to determine equality between elements.

Public Member Operators

```
RWTPtrHashSet<T,H,EQ>&
operator=(const RWTPtrHashSet<T,H,EQ>& s);
```

Clears all elements of self and replaces them by copying all elements of `s`.

```
bool
operator==(const RWTPtrHashSet<T,H,EQ>& s) const;
```

Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

Public Member Functions

```
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. self function must have prototype:

```
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
size_type
capacity() const;
```

Returns the number of buckets(slots) available in the underlying hash representation. See **resize** below.

```
void
clear();
```

Clears the collection by removing all items from self.

void
clearAndDestroy();

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use self method if multiple pointers to the same object are stored. (If the equality operator is reflexive, the container cannot hold such multiple pointers.)

bool
contains(const T* a) const;

Returns `true` if there exists an element `t` in self such that the expression`(*t == *a)` is `true`, otherwise returns `false`.

bool
contains(bool (*fn)(const T*,void*), void* d) const;

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

void
difference(const RWTPtrHashSet<T,H,EQ>& s);

Sets self to the set-theoretic difference given by `(self - s)`. Elements from each set are dereferenced before being compared.

iterator
end();

const_iterator
end() const;

Returns an iterator positioned “just past” the last element in self.

size_type
entries() const;

Returns the number of items in self.

float
fillRatio() const;

Returns the ratio `entries()`/`capacity()`.

const T*
find(const T* a) const;

If there exists an element `t` in self such that `*T` compares equal to `*a`, returns `t`. Otherwise, returns `rwnil`.

```
const T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an element with an equivalent key.

```
void
intersection(const RWTPtrHashSet<T,H,EQ>& s);
```

Destructively performs a set theoretic intersection of self and `s`, replacing the contents of self with the result.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isEquivalent(const RWTPtrHashSet<T,H,EQ>& s) const;
```

Returns `true` if there is set equivalence between self and `s`, and returns `false` otherwise.

```
bool
isProperSubsetOf(const RWTPtrHashSet<T,H,EQ>& s) const;
```

Returns `true` if self is a proper subset of `s`, and returns `false` otherwise.

```
bool
isSubsetOf(const RWTPtrHashSet<T,H,EQ>& s) const;
```

Returns `true` if self is a subset of `s` or if self is set equivalent to `s`, `false` otherwise.

```
size_type
occurrencesOf(const T* a) const;
```

Returns the number of elements `t` that compare equal to `*a`

```
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
remove(const T* a);
```

Removes and returns the first element `t` in self that compares equal to `*a`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self that compare equal to `*a`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```

Changes the capacity of self by creating a new hashed set with a capacity of `sz` . `resize` copies every element of self into the new container and finally swaps the internal representation of the new container with the internal representation of `self`.

```
rw_hashset<T*,container_hash, container_eq>&
std();
const rw_hashset<T*,container_hash, container_eq>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

```
void
symmetricDifference(const RWTPtrHashSet<T,H,EQ>& s);
```

Destructively performs a set theoretic symmetric difference operation on self and `s`. Self is replaced by the result. A symmetric difference can be defined as $(A \cup B)-(A \cap B)$.

```
void
Union(const RWTPtrHashSet<T,H,EQ>& s);
```

Destructively performs a set theoretic union operation on self and `s`. Self is replaced by the result. Note the uppercase "U" in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTPtrHashSet<T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm,
       const RWTPtrHashSet<T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrHashSet<T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrHashSet<T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrHashSet<T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrHashSet<T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tphset.h>
RWTPtrHashSet<T,H,EQ> m;
RWTPtrHashSet<T,H,EQ> itr(m);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTPtrHashSetIterator* described in Appendix A.

Description

RWTPtrHashSetIterator is supplied with *Tools.h++* 7.x to provide an iterator interface to the Standard Library based collections that has backward compatibility with the container iterators provided in *Tools.h++* 6.x.

Iteration over an *RWTPtrHashSet* is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a pre-increment or an `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

struct silly_h{
   unsigned long operator()(RWCString x) const
     { return x.length() * (long)x(0); }
};
int main(){
   RWTPtrHashSet <RWCString,silly_h,equal_to<RWCString> > age;
   RWTPtrHashSetIterator
     <RWCString,silly_h,equal_to<RWCString> > itr(age);
   age.insert(new RWCString("John"));
   age.insert(new RWCString("Steve"));
   age.insert(new RWCString("Mark"));
//Duplicate insertion is rejected
   age.insert(new RWCString("Steve"));
   for(;++itr;) cout << *itr.key() << endl;
```

```
   return 0;
}
```

Program Output (not necessarily in this order)

```
John
Mark
Steve
```

Public Constructors

`RWTPtrHashSetIterator<T,H,EQ>``(RWTPtrHashSet<T,H,EQ>&h);`

Creates an iterator for the hashed set `h`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

`T*`
`operator()``();`

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

`RWBoolean`
`operator++``();`

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multi-map, self will now point to an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

`RWTPtrHashSet<T,H,EQ>*`
`container()` `const;`

Returns a pointer to the collection being iterated over.

`T*`
`key``() const;`

Returns the element referenced by `self`. Undefined if self is not referencing a value within the set.

`void`
`reset``();`
`void`
`reset``(RWTPtrHashSet<T,H,EQ>& h);`

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrHashSet` to `reset()` will reset the iterator on that container.

Synopsis

```
#define RWTPtrHashTable RWTPtrHashMultiSet
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTPtrHashMultiSet*. Although the old name (*RWTPtrHashTable*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTPtrHashTable* in Appendix A.

Synopsis

```
#define RWTPtrHashTableIterator RWTPtrHashMultiSetIterator
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTPtrHashMultiSetIterator*. Although the old name (*RWTPtrHashTableIterator*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTPtrHashTableIterator* in Appendix A.

Synopsis

```
#include <rw/tpmap.h>
RWTPtrMap<K,T,C> m;
```

Standard C++ Library Dependent!

***RWTPtrMap* requires the Standard C++ Library.**

Description

This class maintains a pointer-based collection of associations of type `pair<K* const, T*>`. The first part of the association is a key of type `K*`, the second is its associated item of type `T*`. Order is determined by the key according to a comparison object of type `C`. `C` must induce a total ordering on elements of type `K` via a public member

```
bool operator()(const K& x, const K& y)
```

which returns `true` if `x` and its partner should precede `y` and its partner within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example. Note that keys will be dereferenced before being compared.

RWTPtrMap<K,T,C> will not accept a key that compares equal to any key already in the collection. (*RWTPtrMultiMap<K,T,C>* may contain multiple keys that compare equal to each other.) Equality is based on the comparison object and *not* on the `==` operator. Given a comparison object `comp`, keys `a` and `b` are equal if

```
!comp(a,b) && !comp(b,a).
```

Persistence

Isomorphic.

Examples

In this example, a map of *RWCString*s and *RWDate*s is exercised.

```
//
// tpmap.cpp
//
#include <rw/tpmap.h>
#include <rw/cstring.h>
#include <rw/rwdate.h>
#include <iostream.h>
#include <function.h>

main(){
  RWTPtrMap<RWCString, RWDate, less<RWCString> > birthdays;

  birthdays.insert
```

```
    (
      new RWCString("John"),
      new RWDate(12, "April", 1975)
    );
  birthdays.insert
    (
      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:

```
04/12/75
```

Related Classes

Class ***RWTPtrMultiMap<K,T,C>*** offers the same interface to a pointer-based collection that accepts multiple keys that compare equal to each other. ***RWTPtrSet<T,C>*** maintains a pointer-based collection of keys without the associated items.

Class ***map<K*,T*,deref_compare<C,K, allocator> >*** is the C++-standard collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef rw_deref_compare<C,K>                    container_comp;
typedef map<K*,T*,container_comp, allocator>     container_type;
typedef container_type::size_type                size_type;
typedef container_type::difference_type          difference_type;
typedef container_type::iterator                 iterator;
typedef container_type::const_iterator           const_iterator;
typedef pair <K* const, T*>                      value_type;
typedef pair <K* const, T*>&                     reference;
typedef const pair <K* const, T*>&               const_reference;
typedef K*                                       value_type_key;
typedef T*                                       value_type_data;
typedef K*&                                      reference_key;
typedef T*&                                      reference_data;
typedef const K*const&                     const_reference_key;
typedef const T*const&                     const_reference_data;
```

Public Constructors

```
RWTPtrMap<K,T,C>
(const container_comp& comp = container_comp());
```

Constructs an empty map with comparator `comp`.

```
RWTPtrMap<K,T,C>(const RWTPtrMap<K,T,C>& rwm);
```
Copy constructor.

```
RWTPtrMap<K,T,C>(const container_type& m);
```
Constructs a map by copying all elements from `m`.

```
RWTPtrMap<K,T,C>
(value_type* first,value_type* last,
 const container_comp& comp = container_comp());
```
Constructs a map by copying elements from the array of `pair`s pointed to by `first`, up to, but not including, the pair pointed to by `last`.

Public Member Operators

```
RWTPtrMap<K,T,C>&
operator=(const RWTPtrMap<K,T,C>& m);
RWTPtrMap<K,T,C>&
operator=(const container_type& m);
```
Destroys all associations in self and replaces them by copying all associations from `m`.

```
bool
operator<(const RWTPtrMap<K,T,C>& m) const;
```
Returns `true` if self compares lexicographically less than `m`, otherwise returns `false`. Keys in each collection are dereferenced before being compared. Assumes that type `K` has well-defined less-than semantics.

```
bool
operator==(const RWTPtrMap<K,T,C>& m) const;
```
Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual keys that compare equal to each other. Keys are dereferenced before being compared.

```
T*&
operator[](const K* key);
```
Looks up `key` and returns a reference to its associated item. If the key is not in the dictionary, then it will be added with an associated uninitialized pointer of type `T*`. Because of this, if there is a possibility that a key will not be in the dictionary, then this operator should only be used as an lvalue.

Public Member Functions

```
void
apply(void (*fn)(const K*,T*&,void*),void* d);
void
apply(void (*fn)(const K*,const T*,void*),void* d) const;
```
Applies the user-defined function pointed to by `fn` to every association in the collection. This function must have one of the prototypes:

```
void yourfun(const K* key, T*& a, void* d);
void yourfun(const K* key, const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K*,T*&,void*),void* d);
```

```
void
applyToKeyAndValue
(void (*fn)(const K*,const T*,void*),void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply**.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all associations from the collection *and* uses `operator delete` to destroy the objects pointed to by the keys and their associated items. Do not use this method if multiple pointers to the same object are stored. (This could happen even if keys all compare different, since items are not considered during comparison.)

```
bool
contains(const K* key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `*key`, otherwise returns `false`.

```
bool
contains(bool (*fn)(value_type,void*), void* d) const;
```

Returns `true` if there exists an association a in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
const K*
find(const K* key) const;
```

If there exists a key `j` in self that compares equal to `*key`, then `j` is returned. Otherwise, returns `rwnil`.

```
value_type
find(bool (*fn)(value_type,void*), void* d) const;
```

If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, then returns `a`. Otherwise, returns `pair<rwnil,rwnil>`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
findValue(const K* key);
const T*
findValue(const K* key) const;
```

If there exists a key `j` in self that compares equal to `*key`, returns the item associated with `j`. Otherwise, returns `rwnil`.

```
const K*
findKeyAndValue(const K* key, T*& tr);
const K*
findKeyAndValue(const K* key, const T*& tr) const;
```

If there exists a key `j` in self that compares equal to `*key`, assigns the item associated with `j` to `tr`, and returns `j`. Otherwise, returns `rwnil` and leaves the value of `tr` unchanged.

```
bool
insert(K* key, T* a);
```

Adds `key` with associated item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an association with the equivalent key.

```
bool
insertKeyAndValue(K* key, T* a);
```

This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert**.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

`size_type`
`occurrencesOf``(const K* key) const;`
Returns the number of keys `j` in self that compare equal to `*key`.

`size_type`
`occurrencesOf`
`(bool (*fn)(value_type,void*), void* d) const;`
Returns the number of associations `a` in self such that the expression`((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

`K*`
`remove``(const K* key);`
Removes the first association with key `j` in self that compare euqal to `*key` and returns `j`. Returns `rwnil` if there is no such association.

`K*`
`remove``(bool (*fn)(value_type,void*), void* d);`
Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns its key. Returns `rwnil` if there is no such association. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

`size_type`
`removeAll``(const K* key);`
Removes all associations with key `j` in self that compare equal to `*key`. Returns the number of associations removed.

`size_type`
`removeAll``(bool (*fn)(value_type,void*), void* d);`
Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
container_type
std();
const container_type
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Related Global Operations

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrMap<K,T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrMap<K,T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrMap<K,T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrMap<K,T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrMap<K,T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrMap<K,T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpmap.h>
RWTPMap<K,T,C> map;
RWTPMapIterator<K,T,C> itr(map);
```

Standard C++ Library Dependent!

***RWTPtrMapIterator* requires the Standard C++ Library.**

Description

RWTPrtMapIterator is supplied with Tools 7 to provide an iterator interface to the new Standard Library based collections that has backward compatibility with the container iterators provided in Tools 6.

The order of iteration over an ***RWTPtrMap*** is dependent on the comparator object supplied as applied to the key values of the stored associations.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTPtrMap<RWCString,int,less<RWCString> > age;
   RWTPtrMapIterator<RWCString,int,less<RWCString> > itr(age);
age.insert(new RWCString("John") ,new int(30));
   age.insert(new RWCString("Steve"),new int(17));
   age.insert(new RWCString("Mark") ,new int(24));
//Insertion is rejected, no duplicates allowed
   age.insert(new RWCString("Steve"),new int(24));
for(;itr();)
     cout << *itr.key() << "\'s age is " << *itr.value() << endl;
   return 0;
}
```

Program Output
```
John's age is 30
Mark's age is 24
Steve's age is 17
```

Public Constructors

RWTPtrMapIterator<K,T,C>(const RWTPtrMap<K,T,C>& rwm);

Creates an iterator for the map `rwm`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

K*
operator()();

Advances self to the next element, dereferences the resulting iterator and returns its key. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multimap, self will now point to an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

RWTPtrMap<K,T,C>*
container() const;

Returns a pointer to the collection being iterated over.

K*
key() const;

Returns the key portion of the association currently referenced by self. Undefined if self is not referencing a value within the map.

void
reset();
void
reset(RWTPtrMap<K,T,C>& h);

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrMap` to `reset()` will reset the iterator on that container.

T*
value();

Returns the value portion of the association pointed to by self. Undefined if self is not referencing a value within the map.

Synopsis

```
#include <rw/tpmmap.h>
RWTPtrMultiMap<K,T,C> m;
```

Standard C++ Library Dependent!

RWTPtrMultiMap **requires the Standard C++ Library.**

Description

This class maintains a pointer-based collection of associations of type `pair<K*, const T*>`. The first part of the association is a key of type `K*`, the second is its associated item of type `T*`. Order is determined by the key according to a comparison object of type `C`. `C` must induce a total ordering on elements of type `K` via a public member

```
bool operator()(const K& x, const K& y)
```

which returns `true` if `x` and its partner should precede `y` and its partner within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example. Note that keys will be dereferenced before being compared.

RWTPtrMultiMap<K,T,C> may contain multiple keys that compare equal to each other. (***RWTPtrMap<K,T,C>*** will not accept a key that compares equal to any key already in the collection.) Equality is based on the comparison object and *not* on the `==` operator. Given a comparison object `comp`, keys `a` and `b` are equal if

```
!comp(a,b) && !comp(b,a).
```

Persistence

Isomorphic.

Examples

In this example, a multimap of `RWCString`s and `RWDate`s is exercised.

```
//
// tpmmap.cpp
//
#include <rw/tpmmap.h>
#include <rw/cstring.h>
#include <rw/rwdate.h>
#include <iostream.h>

main(){
  typedef RWTPtrMultiMap<RWCString, RWDate, less<RWCString> >
   RWMMap;
  RWMMap birthdays;
```

```
  birthdays.insert(new RWCString("John"),
                            new RWDate(12, "April", 1975));
  birthdays.insert(new RWCString("Ivan"),
                            new RWDate(2, "Nov", 1980));
  birthdays.insert(new RWCString("Mary"),
                            new RWDate(22, "Oct", 1987));
  birthdays.insert(new RWCString("Ivan"),
                            new RWDate(19, "June", 1971));
  birthdays.insert(new RWCString("Sally"),
                            new RWDate(15, "March", 1976));
  birthdays.insert(new RWCString("Ivan"),
                            new RWDate(6, "July", 1950));

  // How many "Ivan"s?
  RWCString ivanstr("Ivan");
  RWMMap::size_type n = birthdays.occurrencesOf(&ivanstr);
  RWMMap::size_type idx = 0;
  cout << "There are " << n << " Ivans:" << endl;
  RWMMap::const_iterator iter =
                           birthdays.std().lower_bound(&ivanstr);

  while (++idx <= n)
    cout << idx << ".  " << *(*iter++).second << endl;
  return 0;
}
```

Program Output:

```
There are 3 Ivans:
1.  11/02/80
2.  06/19/71
3.  07/06/50
```

Related Classes

Class ***RWTPtrMap<K,T,C>*** offers the same interface to a pointer-based collection that will not accept multiple keys that compare equal to each other. ***RWTPtrMultiSet<T,C>*** maintains a pointer-based collection of keys without the associated values.

Class ***multimap<K*,T*,deref_compare<C,K,allocator> >*** is the C++-standard collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef rw_deref_compare<C,K>              container_comp;
typedef multimap<K*,T*,container_comp,allocator>
                 container_type;
typedef container_type::size_type          size_type;
typedef container_type::difference_type    difference_type;
typedef container_type::iterator           iterator;
typedef container_type::const_iterator     const_iterator;
typedef pair<K* const, T*>                 value_type;
typedef pair<K* const, T*>                 reference;
typedef const pair<K* const, T*>&          const_reference;
typedef K*                                 value_type_key;
typedef T*                                 value_type_data;
typedef K*&                                reference_key;
typedef T*&                                reference_data;
typedef const K*const&                     const_reference_key;
typedef const T*const&                     const_reference_data;
```

Public Constructors

RWTPtrMultiMap<K,T,C>
`(const container_comp& comp =container_comp());`

Constructs an empty map with comparator `comp`.

RWTPtrMultiMap<K,T,C>`(const container_type& m);`

Constructs a multimap by copying all element from `m`.

RWTPtrMultiMap<K,T,C>`(const RWTPtrMultiMap<K,T,C>& rwm);`

Copy constructor.

RWTPtrMultiMap<K,T,C>`(value_type* first,value_type* last,`
`const container_comp& comp = container_comp());`

Constructs a multimap by copying elements from the array of `pair`s pointed to by `first`, up to, but not including, the pair pointed to by `last`.

Public Member Operators

`RWTPtrMultiMap<K,T,C>&`
operator=`(const container_type& m);`
`RWTPtrMultiMap<K,T,C>&`
operator=`(const RWTPtrMultiMap<K,T,C>& m);`

Destroys all associations in self and replaces them by copying all associations from `m`.

`bool`
operator<`(const RWTPtrMultiMap<K,T,C>& m);`

Returns `true` if self compares lexicographically less than `m`, otherwise returns `false`. Keys in each collection are dereferenced before being compared. Assumes that type `K` has well-defined less-than semantics.

`bool`
operator==`(const RWTPtrMultiMap<K,T,C>& m);`

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual keys that compare equal to each other. Keys are dereferenced before being compared.

Public Member Functions

`void`
apply`(void (*fn)(const K*, T*&,void*),void* d);`
`void`
apply`(void (*fn)(const K*,const T*,void*),void* d) const;`

Applies the user-defined function pointed to by `fn` to every association in the collection. This function must have one of the prototypes:

```
void yourfun(const K* key, T*& a, void* d);
void yourfun(const K* key, const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K*, T*&,void*),void* d);
```

```
void
applyToKeyAndValue
(void (*fn)(const K*,const T*,void*),void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply**.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all associations from the collection *and* uses `operator delete` to destroy the objects pointed to by the keys and their associated items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains(const K* key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `*key`, otherwise returns `false`.

```
bool
contains(bool (*fn)(value_type,void*), void* d) const;
```

Returns `true` if there exists an association a in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
const K*
find(const K* key) const;
```
If there exists a key `j` in self that compares equal to `*key`, then `j` is returned. Otherwise, returns `rwnil`.

```
value_type
find(bool (*fn)(value_type,void*), void* d) const;
```
If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, then returns `a`. Otherwise, returns `pair<rwnil,rwnil>`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
findValue(const K* key);
const T*
findValue(const K* key) const;
```
If there exists a key `j` in self such that the expression `(*j == *key)` is `true`, returns the item associated with `j`. Otherwise, returns `rwnil`.

```
const K*
findKeyAndValue(const K* key, T*& tr);
const K*
findKeyAndValue(const K* key, const T*& tr) const;
```
If there exists a key `j` in self that compares equal to `*key`, assigns the item associated with `j` to `tr`, and returns `j`. Otherwise, returns `rwnil` and leaves the value of `tr` unchanged.

```
bool
insert(K* key, T* a);
```
Adds `key` with associated item `a` to the collection. Returns `true`.

```
bool
insertKeyAndValue(K* key, T* a);
```
This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert**.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K* key) const;
```
Returns the number of keys `j` in self that compare equal to `*key`.

size_type
occurrencesOf
(bool (*fn)(value_type,void*), void* d) const;

Returns the number of associations a in self such that the expression((*fn)(a,d)) is true. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter d.

K*
remove(const K* key);

Removes the first association with key j in self such that the expression (*j == *key) is true and returns j. Returns rwnil if there is no such association.

K*
remove(bool (*fn)(value_type,void*), void* d);

Removes the first association a in self such that the expression ((*fn)(a,d)) is true and returns its key. Returns rwnil if there is no such association. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter d.

size_type
removeAll(const K* key);

Removes all associations with key j in self that compare equal to *key. Returns the number of associations removed.

size_type
removeAll(bool (*fn)(value_type,void*), void* d);

Removes all associations a in self such that the expression ((*fn)(a,d))is true. Returns the number removed. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(value_type a, void* d);
```

Client data may be passed through parameter d.

container_type&
std();
const container_type&
std() const;

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTPtrMultiMap<K,T,C>& coll);
RWFile&
operator<<(RWFile& strm,
       const RWTPtrMultiMap<K,T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrMultiMap<K,T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrMultiMap<K,T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrMultiMap<K,T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrMultiMap<K,T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpmmap.h>
RWTPtrMultiMap<K,T,C> map;
RWTPtrMultiMapIterator<K,T,C> itr(map);
```

Standard C++ Library Dependent!

***RWTPtrMultiMapIterator* requires the Standard C++ Library.**

Description

RWTPtrMultiMapIterator is supplied with Tools 7 to provide an iterator interface to the new Standard Library based collections with backward compatibility to the Tools 6 container iterators.

The order of iteration over an *RWTPtrMultiMap* is dependent on the comparator object of the container as applied to the key values of the stored associations.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

```
#include<rw/tpmmap.h>
#include<iostream.h>
#include<rw/cstring.h>
#include<utility>

int main(){
   RWTPtrMultiMap<RWCString,int,less<RWCString> > age;
   RWTPtrMultiMapIterator<RWCString,int,less<RWCString> > itr(age);

   age.insert(new RWCString("John"), new int(30));
   age.insert(new RWCString("Steve"),new int(17));
   age.insert(new RWCString("Mark"), new int(24));
   age.insert(new RWCString("Steve"),new int(24));

   for(;itr();)
     cout << *itr.key() << "\'s age is " << *itr.value() << endl;

   return 0;
}
```

Program Output

```
John's age is 30
Mark's age is 24
Steve's age is 17
Steve's age is 24
```

Public Constructors

RWTPtrMultiMapIterator<K,T,C>`(const RWTPtrMultiMap<K,T,C>& m);`

Creates an iterator for the multimap `m`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

`K*`
`operator()``();`

Advances self to the next element, dereferences the resulting iterator and returns its key. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

`RWBoolean`
`operator++``();`

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multimap, self will now point to an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

`RWTPtrMultiMap<K,T,C>*`
`container()` `const;`

Returns a pointer to the collection being iterated over.

`K*`
`key``() const;`

Returns the key portion of the association currently referenced by `self`. Undefined if self is not referencing a value within the multimap.

`void`
`reset``();`
`void`
`reset``(RWTPtrMultiMap<K,T,C>& h);`

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrMultiMap` to `reset()` will reset the iterator on that container.

`T*`
`value``();`

Returns the value portion of the association referenced by `self`. Undefined if self is not referencing a value within the multimap.

Synopsis

```
#include <rw/tpmset.h>
RWTPtrMultiSet<T,C> s;
```

Standard C++ Library Dependent!

RWTPtrMultiSet **requires the Standard C++ Library.**

Description

This class maintains a pointer-based collection of values, which are ordered according to a comparison object of type `C`. Class `T` is the type pointed to by the items in the collection. `C` must induce a total ordering on elements of type `T` via a public member

```
bool operator()(const T& x, const T& y)
```

which returns `true` if `x` should precede `y` within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example. Note that items in the collection will be dereferenced before being compared.

RWTPtrMultiSet<T,C> may contain multiple items that compare equal to each other. (***RWTPtrSet<T,C>*** will not accept an item that compares equal to an item already in the collection.)

Persistence

Isomorphic.

Examples

In this example, a multi-set of `RWCString`s is exercised.

```
//
// tpmset.cpp
//
#include <rw/tpmset.h>
#include <rw/cstring.h>
#include <iostream.h>
#include <function.h>

main(){
  RWTPtrMultiSet<RWCString, less<RWCString> > set;

  set.insert(new RWCString("one"));
  set.insert(new RWCString("two"));
  set.insert(new RWCString("three"));
  set.insert(new RWCString("one"));  // OK: duplicates allowd

  cout << set.entries() << endl;   // Prints "4"

  set.clearAndDestroy();
  cout << set.entries() << endl;   // Prints "0"

  return 0;
}
```

Related Classes

Class ***RWTPtrSet<T,C>*** offers the same interface to a pointer-based collection that will not accept multiple items that compare equal to each other. ***RWTPtrMultiMap<K,T,C>*** maintains is a pointer-based collection of key-value pairs.

Class ***multiset<T*, rw_deref_compare<C,T>,allocator >*** is the C++-standard collection that serves as the underlying implementation for ***RWTPtrMultiSet<T,C>***.

Public Typedefs

```
typedef rw_deref_compare<C,T>                       container_comp;
typedef multiset<T*, container_comp,allocator> container_type;
typedef container_type::size_type                  size_type;
typedef container_type::difference_type            difference_type;
typedef container_type::iterator                   iterator;
typedef container_type::const_iterator             const_iterator;
typedef T*                                         value_type;
typedef T* const&                                  reference;
typedef T* const&                                  const_reference;
```

Public Constructors

RWTPtrMultiSet<T,C>(const container_comp& = container_comp());

Constructs an empty set.

RWTPtrMultiSet<T,C>(const RWTPtrMultiSet<T,C>& rws);

Copy constructor.

RWTPtrMultiSet<T,C>(const container_type>& ms);

Constructs a multimap by copying all elements from `ms`.

RWTPtrMultiSet<T,C>(T* const* first,T* const* last,const container_comp& = container_comp());

Constructs a set by copying elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

RWTPtrMultiSet<T,C>&
operator=(const container_type>& s);
RWTPtrMultiSet<T,C>&
operator=(const RWTPtrMultiSet<T,C>& s);

Clears all elements of self and replaces them by copying all elements of `s`.

bool
operator<(const RWTPtrMultiSet<T,C>& s) const;

Returns `true` if self compares lexicographically less than `s`, otherwise returns `false`. Items in each collection are dereferenced before being compared. Assumes that type `T` has well-defined less-than semantics.

bool
operator==(const RWTPtrMultiSet<T,C>& s) const;

Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that

compare equal to each other. Elements are dereferenced before being compared.

Public Member Functions

```
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have prototype:

```
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains(const T* a) const;
```

Returns `true` if there exists an element `t` in self that compares equal to `*a`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
difference(const RWTPtrMultiSet<T,C>& s);
```

Sets self to the set-theoretic difference given by `(self - s)`. Elements from each set are dereferenced before being compared.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries();
```

Returns the number of items in self.

```
const T*
find(const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
const T*
find(bool (*fn)(T*,void*), void* d);
const T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the collection. Returns `true`.

```
void
intersection(const RWTPtrMultiSet<T,C>& s);
```

Sets self to the intersection of self and `s`. Elements from each set are dereferenced before being compared.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isEquivalent(const RWTPtrMultiSet<T,C>& s) const;
```

Returns `true` if there is set equivalence between self and `s`, and returns `false` otherwise.

```
bool
isProperSubsetOf(const RWTPtrMultiSet<T,C>& s) const;
```

Returns `true` if self is a proper subset of `s`, and returns `false` otherwise.

```
bool
isSubsetOf(const RWTPtrMultiSet<T,C>& s) const;
```
Returns `true` if self is a subset of `s` or if self is set equivalent to `rhs`, `false` otherwise.

```
size_type
occurrencesOf(const T* a) const;
```
Returns the number of elements `t` in self that compare equal to `*a`.

```
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```
Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
remove(const T* a);
```
Removes and returns the first element `t` in self that compares equal to `*a`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)(const T*,void*), void* d);
```
Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```
Removes all elements `t` in self that compare equal to `*a`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```
Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
multiset<T*, container_comp,allocator>&
std();
const multiset<T*, container_comp,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

```
void
symmetricDifference(const RWTPtrMultiSet<T,C>& s);
```

Sets self to the symmetric difference of self and `s`. Elements from each set are dereferenced before being compared.

```
void
Union(const RWTPtrMultiSet<T,C>& s);
```

Sets self to the union of self and `s`. Elements from each set are dereferenced before being compared. Note the uppercase “U” in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrMultiSet<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrMultiSet<T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrMultiSet<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrMultiSet<T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrMultiSet<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrMultiSet<T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpmset.h>
RWTPtrMultiSet<T,C> set;
RWTPtrMultiSetIterator<T,C> itr(set);
```

Standard C++ Library Dependent!

RWTPtrMultiSetIterator **requires the Standard C++ Library.**

Description

RWTPtrMultiSetIterator is supplied with Tools 7 to provide an iterator interface to the new Standard Library based collections that has backward compatibility with the container iterators provided in Tools 6.

The order of iteration over an ***RWTPtrMultiSet*** is dependent upon the comparator object parameter `C` as applied to the values stored in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTPtrMultiSet<RWCString, less<RWCString> > a;
   RWTPtrMultiSetIterator<RWCString, less<RWCString> > itr(a);

   a.insert(new RWCString("John"));
   a.insert(new RWCString("Steve"));
   a.insert(new RWCString("Mark"));
   a.insert(new RWCString("Steve"));

   for(;itr();)
     cout << *itr.key() <<endl;

   return 0;
}
```

Program Output

```
John
Mark
Steve
Steve
```

Public Constructors

RWTPtrMultiSetIterator<T,C>(const RWTPtrMultiSet<T,C>& m);

Creates an iterator for the multi-set `m`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

T*
operator()();

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multi-set, self will now point to an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

RWTPtrMultiSet<T,C>*
container() const;

Returns a pointer to the collection being iterated over.

T*
key();

Returns the stored value referenced by `self`. Undefined if self is not referencing a value within the list.

void
reset();
void
reset(RWTPtrMultiSet<T,C>& h);

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrMultiSet` with `reset()` will reset the iterator on that container.

Synopsis

```
#include <rw/tpordvec.h>
RWTPtrOrderedVector<T> ordvec;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface for *RWTPtrOrderedVector* described in Appendix A.

Description

This class maintains a pointer-based collection of values, implemented as a vector. Class ***T*** is the type pointed to by the items in the collection

Persistence

Isomorphic

Example

In this example, a pointer-based vector of type `RWDate` is exercised.

```
//
// tporddat.cpp
//
#include <rw/tpordvec.h>
#include <rw/rwdate.h>
#include <iostream.h>

main(){
  RWTPtrOrderedVector<RWDate> week(7);

  RWDate begin;  // Today's date

  for (int i=0; i<7; i++)
    week.insert(new RWDate(begin++));

  for (i=0; i<7; i++)
    cout << *week[i] << endl;

  return 0;
}
```

Program Output:

```
05/31/95
06/01/95
06/02/95
06/03/95
06/04/95
06/05/95
06/06/95
```

Related Classes

Classes *RWTPtrDeque<T>*, *RWTPtrSlist<T>*, and *RWTPtrDlist<T>* also provide a Rogue Wave pointer-based interface to C++-standard sequence collections.

Class ***vector<T*,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef vector<T*,allocator>                  container_type;
typedef container_type::iterator              iterator;
typedef container_type::const_iterator        const_iterator;
typedef container_type::size_type             size_type;
typedef container_type::difference_type       difference_type;
typedef T*                                    value_type;
typedef T*&                                   reference;
typedef T* const&                             const_reference;
```

Public Constructors

`RWTPtrOrderedVector<T>``();`

Constructs an empty vector.

`RWTPtrOrderedVector<T>``(const RWTPtrOrderedVector<T>& rwvec);`

Copy constructor.

`RWTPtrOrderedVector<T>``(const vector<T*,allocator>& vec);`

Constructs an ordered vector by copying all elements of `vec`.

`RWTPtrOrderedVector<T>``(size_type n, T* a);`

Constructs a vector with `n` elements, each initialized to `a`.

`RWTPtrOrderedVector<T>``(T* const* first,T* const* last);`

Constructs a vector by copying elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTPtrOrderedVector<T>&`
`operator=``(const RWTPtrOrderedVector<T>& vec);`
`RWTPtrOrderedVector<T>&`
`operator=``(const vector<T*,allocator>& vec);`

Clears all elements of self and replaces them by copying all elements of `vec`.

`bool`
`operator<``(const RWTPtrOrderedVector<T>& vec) const;`

Returns `true` if self compares lexicographically less than `vec`, otherwise returns `false`. Items in each collection are dereferenced before being compared.

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

Returns `true` if self compares equal to `vec`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that

compare equal to each other. Elements are dereferenced before being compared.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the ith element of self. Index i should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the ith element of self. Index i must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(T* a);
```

Adds the item a to the end of the collection.

```
void
apply(void (*fn)(T*&,void*), void* d);
void
apply(void (*fn)(T*,void*), void* d);
void
apply(void (*fn)(const T*,void*), void*`d) const;
```

Applies the user-defined function pointed to by fn to every item in the collection. This function must have one of the prototypes:

```
void yourfun(reference a, void* d);
void yourfun(T* a, void* d);
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter d.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the ith element of self. Index i must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains(const T* a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(*t == *a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(T*,void*), void* d) const;
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*const*
data() const;
```

Returns a pointer to the first element of the vector.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned “just past” the last element in self.

```
size_type
entries();
```

Returns the number of items in self.

```
T*
find(const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
T*
find(bool (*fn)(T*,void*), void* d) const;
T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
```

```
const_reference
first() const;
```

Returns a reference to the first element of self.

```
size_type
index(const T* a) const;
```

Returns the position of the first item `t` in self such that `(*t == *a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(T*,void*), void* d) const;
size_type
index(bool (*fn)(const T*,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, T* a);
```

Inserts the item `a` in front of the item at position `i` in self. This position must be between zero and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
T*&
last();
```

```
T*const&
last() const;
```

Returns a reference to the last item in the collection.

```
size_type
length() const;
```

Returns the number of items in self.

```
reference
maxElement();
```

```
const_reference
maxElement() const;
```

```
reference
minElement();
```

```
const_reference
minElement() const;
```

Returns a reference to the maximum or minimum element in self.

```
size_type
occurrencesOf(const T* a) const;
```

Returns the number of elements `t` in self such that the expression `(*t == *a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(T*,void*),void* d) const;
size_type
occurrencesOf(bool (*fn)(const T*,void*),void* d) const;
```

Returns the number of elements `t` in self such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
prepend(T* a);
```

Adds the item `a` to the beginning of the collection.

```
T*
remove(const T* a);
```

Removes and returns the first element `t` in self such that the expression `(*t == *a)` is `true`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)( T*,void*), void* d);
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(const T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self such that the expression `(*t == *a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(T*,void*), void* d);
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T*
removeFirst();
```

Removes and returns the first item in the collection.

```
T*
removeLast();
```

Removes and returns the first item in the collection.

```
size_type
replaceAll(const T* oldVal, T* newVal);
```

Replaces with `newVal` all elements `t` in self such that the expression `(*t == *oldVal)` is `true`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(T*, void*),void* x,T* newVal);
size_type
replaceAll(bool (*fn)(const T*, void*),void* x,T* newVal);
```

Replaces with `newVal` all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type n);
```

Modify the capacity of the vector to be at least as large as `n`. The function has no effect if the capacity is already as large as `n`.

```
void
sort();
```

Sorts the collection using the less-than operator to compare elements. Elements are dereferenced before being compared.

```
vector<T*,allocator>&
std();
const vector<T*,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
      const RWTPtrOrderedVector<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrOrderedVector<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrOrderedVector<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrOrderedVector<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrOrderedVector<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrOrderedVector<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tpset.h>
RWTPtrSet<T,C> s;
```

Standard C++ Library Dependent!

***RWTPtrSet* requires the Standard C++ Library.**

Description

This class maintains a pointer-based collection of values, which are ordered according to a comparison object of type ***C***. Class ***T*** is the type pointed to by the items in the collection. ***C*** must induce a total ordering on elements of type ***T*** via a public member

```
bool operator()(const T& x, const T& y)
```

which returns `true` if `x` should precede `y` within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example. Note that items in the collection will be dereferenced before being compared.

RWTPtrSet<T,C> will not accept an item that compares equal to an item already in the collection. (***RWTPtrMultiSet<T,C>*** may contain multiple items that compare equal to each other.) Equality is based on the comparison object and *not* on the `==` operator. Given a comparison object `comp`, items `a` and `b` are equal if

```
!comp(a,b) && !comp(b,a).
```

Persistence

Isomorphic.

Examples

In this example, a pointer-based set of `RWCString`s is exercised.

```
//
//tpset.cpp
//
#include <rw/tpset.h>
#include <rw/cstring.h>
#include <iostream.h>
#include <function.h>

main(){
  RWTPtrSet<RWCString, less<RWCString> > set;

  set.insert(new RWCString("one"));
  set.insert(new RWCString("two"));
  set.insert(new RWCString("three"));
  set.insert(new RWCString("one"));  // Rejected: duplicate entry
```

```
  cout << set.entries() << endl;   // Prints "3"

  set.clearAndDestroy();
  cout << set.entries() << endl;   // Prints "0"

  return 0;
}
```

Related Classes

Class *RWTPtrMultiSet<T,C>* offers the same interface to a pointer-based collection that accepts multiple items that compare equal to each other. *RWTPtrMap<K,T,C>* is a pointer-based collection of key-value pairs.

Class ***set<T*,rw_deref_compare<C,T>,allocator>*** is the C++-standard collection that serves as the underlying implementation for *RWTPtrSet<T,C>*.

Public Typedefs

```
typedef rw_deref_compare<C,T>                container_comp;
typedef set<T*, container_comp,allocator>    container_type;
typedef container_type::size_type            size_type;
typedef container_type::difference_type      difference_type;
typedef container_type::iterator             iterator;
typedef container_type::const_iterator       const_iterator;
typedef T*                                   value_type;
typedef T*const&                             reference;
typedef T*const&                             const_reference;
```

Public Constructors

RWTPtrSet<T,C>(const container_comp& comp = container_comp());

Constructs an empty set.

RWTPtrSet<T,C>(const RWTPtrSet<T,C>& rws);

Copy constructor.

RWTPtrSet<T,C>(const container_type& s);

Creates a pointer based set by copying all elements from s.

RWTPtrSet<T,C>(T* const* first,T* const* last,const container_comp& comp = container_comp());

Constructs a set by copying elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

RWTPtrSet<T,C>&
operator=(const container_type& s);
RWTPtrSet<T,C>&
operator=(const RWTPtrSet<T,C>& s);

Clears all elements of self and replaces them by copying all elements of `s`.

bool
operator<(const RWTPtrSet<T,C>& s);

Returns `true` if self compares lexicographically less than `s`, otherwise returns `false`. Items in each collection are dereferenced before being compared. Assumes that type `T` has well-defined less-than semantics.

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

Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

Public Member Functions

```
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have prototype:

```
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items.

```
bool
contains(const T* a) const;
```

Returns `true` if there exists an element `t` in self that compares equal with `*a`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
difference(const RWTPtrSet<T,C>& s);
```

Sets self to the set-theoretic difference given by `(self - s)`. Elements from each set are dereferenced before being compared.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
const T*
find(const T* a) const;
```

If there exists an element `t` in self that compares equal with `*a`, returns `t`. Otherwise, returns `rwnil`.

```
const T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an element with an equivalent key.

```
void
intersection(const RWTPtrSet<T,C>& s);
```

Sets self to the intersection of self and `s`. Elements from each set are dereferenced before being compared.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isEquivalent(const RWTPtrSet<T,C>& s) const;
```

Returns `true` if there is set equivalence between self and `s`, and returns `false` otherwise.

bool
isProperSubsetOf(const RWTPtrSet<T,C>& s) const;

Returns `true` if self is a proper subset of `s`, and returns `false` otherwise.

bool
isSubsetOf(const RWTPtrSet<T,C>& s) const;

Returns `true` if self is a subset of `s` or if self is set equivalent to `s`, `false` otherwise.

size_type
occurrencesOf(const T* a) const;

Returns the number of elements `t` in self that compare equal with `*a`.

size_type
occurrencesOf(bool (*fn)(T*,void*), void* d);
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;

Returns the number of elements `t` in self such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

T*
remove(const T* a);

Removes and returns the first element `t` in self that compares equal with `*a`. Returns `rwnil` if there is no such element.

T*
remove(bool (*fn)(const T*,void*), void* d);

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parámeter `d`.

size_type
removeAll(const T* a);

Removes all elements `t` in self that compares equal with `*a`. Returns the number of items removed.

size_type
removeAll(bool (*fn)(const T*,void*), void* d);

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
set<T*, container_comp,allocator>&
std();
const set<T*, container_comp,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

```
void
symmetricDifference(const RWTPtrSet<T,C>& s);
```

Sets self to the symmetric difference of self and `s`. Elements from each set are dereferenced before being compared.

```
void
Union(const RWTPtrSet<T,C>& s);
```

Sets self to the union of self and `s`. Elements from each set are dereferenced before being compared. Note the uppercase “U” in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrSet<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrSet<T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSet<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrSet<T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSet<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrSet<T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpset.h>
RWTPtrSet<T,C> set;
RWTPtrSetIterator<T,C> itr(set);
```

Standard C++ Library Dependent!

RWTPtrSetIterator **requires the Standard C++ Library.**

Description

RWTPtrSetIterator is supplied with Tools 7 to provide an iterator interface to the new Standard Library based collections that has backward compatibility with the container iterators provided in Tools 6.

The order of iteration over an ***RWTPtrSet*** is dependent on the comparator object supplied as applied to the values stored in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTPtrSet<RWCString,less<RWCString> > a;
   RWTPtrSetIterator<RWCString,less<RWCString> > itr(a);

   a.insert(new RWCString("John"));
   a.insert(new RWCString("Steve"));
   a.insert(new RWCString("Mark"));

//Rejected, duplicate insertions not allowed
   a.insert(new RWCString("Steve"));

   for(;itr();)
     cout << *itr.key() <<endl;

   return 0;
}
```

Program Output
```
John
Mark
Steve
```

Public Constructors

RWTPtrSetIterator<T,C>`(const RWTPtrSet<T,C>& s);`

Creates an iterator for the set `s`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

`T*`
operator()`();`

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

`RWBoolean`
operator++`();`

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the set, self will now reference an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

Public Member Functions

`RWTPtrSet<T,C>*`
container() `const;`

Returns a pointer to the collection being iterated over.

`T*`
key`() const;`

Returns the stored value pointed to by `self`. Undefined if self is not referencing a value within the set.

`void`
reset`();`
`void`
reset`(RWTPtrSet<T,C>& h);`

Resets the iterator so that after being advanced it will point to the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrSet` to `reset()` will reset the iterator on the new container.

Synopsis

```
#include <rw/tpslist.h>
RWTPtrSlist<T> slist;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface for *RWTPtrSlist* described in Appendix A.

Description

This class maintains a pointer-based collection of values, implemented as a singly-linked list. Class ***T*** is the type pointed to by the items in the collection.

Persistence

Isomorphic

Example

```
//
// tpsldat.cpp
//
#include <rw/tpslist.h>
#include <rw/rwdate.h>
#include <iostream.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 * ret = dates.find(new RWDate(2,”June”,52));
  if (ret){
    cout << "Found date " << ret << endl;
  }

  // Remove in reverse order:
  while (!dates.isEmpty())
    cout << *dates.removeLast() << endl;

  return 0;
}
```

Program Output:

```
Found date
4/01/90
3/30/46
6/02/52
```

Related Classes

Classes *RWTPtrDlist<T>*, *RWTPtrDeque<T>*, and *RWTPtrOrderedVector<T>* also provide a Rogue Wave pointer-based interface to C++-standard sequence collections.

Class ***rw_slist<T*>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef rw_slist<T*>                          container_type;
typedef container_type::size_type             size_type;
typedef container_type::difference_type       difference_type;
typedef container_type::iterator              iterator;
typedef container_type::const_iterator        const_iterator;
typedef T*                                    value_type;
typedef T*&                                   reference;
typedef T*const&                              const_reference;
```

Public Constructors

`RWTPtrSlist<T>``();`
Constructs an empty, singly-linked list.

`RWTPtrSlist<T>``(const RWTPtrSlist<T>& rwlst);`
Copy constructor.

`RWTPtrSlist<T>``(const rw_slist<T*>& lst);`
Construct a singly linked list by copying all elements of `lst`.

`RWTPtrSlist<T>``(size_type n, const T* a=0);`
Constructs a singly-linked list with `n` elements, each initialized to `a`.

`RWTPtrSlist<T>``(T* const* first, T* const* last);`
Constructs a singly-linked list by copying elements from the array of `T*s` pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTPtrSlist<T>&`
`operator=``(const RWTPtrSlist<T>& lst);`
`RWTPtrSlist<T>&`
`operator=``(const rw_slist<T*>& lst);`
Empties self then inserts all elements of `lst`.

`bool`
`operator<``(const RWTPtrSlist<T>& lst) const;`
Returns `true` if self compares lexicographically less than `lst`, otherwise returns `false`. Items in each collection are dereferenced before being compared.

`bool`
`operator==``(const RWTPtrSlist<T>& lst) const;`
Returns `true` if self compares equal to `lst`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that

compare equal to each other. Elements are dereferenced before being compared.

```
reference
operator()(size_type i);
```

```
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
reference
operator[](size_type i);
```

```
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(T* a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(T*,void*), void* d);
void
apply(void (*fn)(T*&,void*), void* d);
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(T* a, void* d);
void yourfun(reference a, void* d);
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains(const T* a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(*t == *a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(T*,void*), void* d) const;
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
T*
find(const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
T*
find(bool (*fn)(T*,void*),void* d) const;
T*
find(bool (*fn)(const T*,void*),void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(const T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
```

```
const_reference
first() const;
```

Returns a reference to the first element of self.

```
T*
get();
```

Removes and returns the first element in the collection.

```
size_type
index(const T* a) const;
```

Returns the position of the first item `t` in self such that `(*t == *a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(T*,void*), void* d) const;
size_type
index(bool (*fn)(const T*,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true,` or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, T* a);
```

Inserts the item `a` in front of the item at position `i` in self. This position must be between zero and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
T*&
last();
```
```
T*const&
last() const;
```
Returns a reference to the last item in the collection.

```
reference
maxElement();
```
```
const_reference
maxElement() const;
```
```
reference
minElement();
```
```
const_reference
minElement() const;
```
Returns a reference to the maximum or minimum element in self.

```
size_type
occurrencesOf(const T* a) const;
```
Returns the number of elements `t` in self such that the expression `(*t == *a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(T*,void*), void* d) const;
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```
Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
prepend(T* a);
```
Adds the item `a` to the beginning of the collection.

```
T*
remove(const T* a);
```
Removes and returns the first element `t` in self such that the expression `(*t == *a)` is `true`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)(T*,void*), void* d);
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self such that the expression `(*t == *a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(T*,void*), void* d);
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T*
removeFirst();
```

Removes and returns the first item in the collection.

```
T*
removeLast();
```

Removes and returns the first item in the collection.

```
size_type
replaceAll(const T* oldVal,T* newVal);
```

Replaces with `newVal` all elements `t` in self such that the expression `(*t == *oldVal)` is `true`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(T*, void*),void* x,T* newVal);
size_type
replaceAll(bool (*fn)(const T*, void*),void* x,T* newVal);
```

Replaces with `newVal` all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have one of the prototypes:

```
bool yourTester(T* a, void* d);
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
sort();
```

Sorts the collection using the less-than operator to compare elements. Elements are dereferenced before being compared.

```
rw_slist<T*>&
std();
const rw_slist<T*>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrSlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrSlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrSlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrSlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpslist.h>
RWTPtrSlist<T> dl;
RWTPtrSlistIterator<T> itr(dl);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface for *RWTPtrSlistIterator* described in Appendix A.

Description

RWTPtrSlistIterator is supplied with Tools 7 to provide an iterator interface to the new Standard Library based collections that has backward compatibility with the container iterators provided in Tools 6.

The order of iteration over an *RWTPtrSlist* is dependent upon the order of insertion of items into the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTPtrSlist<RWCString> a;
   RWTPtrSlistIterator<RWCString> itr(a);
   a.insert(new RWCString("John"));
   a.insert(new RWCString("Steve"));
   a.insert(new RWCString("Mark"));
   a.insert(new RWCString("Steve"));

   for(;itr();)
     cout << *itr.key() <<endl;

   return 0;
}
```

Program Output

```
John
Steve
Mark
Steve
```

Public Constructors

RWTPtrSlistIterator<T>(RWTPtrSlist<T>& lst);

Creates an iterator for the list `lst`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

T*
operator()();

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the list, self will now reference an undefined value distinct from the reset value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

RWBoolean
operator+=(size_type n);

Behaves as if the `operator++` member function had been applied `n` times

Public Member Functions

RWTPtrSlist<T>*
container() const;

Returns a pointer to the collection being iterated over.

T*
findNext(const T* a);

Returns the first element `t` encountered by iterating self forward, such that the expression `(*t == *a)` is `true`. If no such element is found, returns `nil`. Leaves self referencing the found item or “off the end.”

T*
findNext(RWBoolean(*fn)(T*, void*), void* d);

Returns the first element `t` encountered by iterating self forward such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
void
insertAfterPoint(T* p);
```

Inserts the pointer `p` into the container directly after the element pointed to by `self`. Leaves self referencing the prior item, or in `reset` condition.

```
T*
key();
```

Returns the stored value pointed to by `self`. Undefined if self is not referencing a value within the list.

```
T*
remove();
```

Returns the stored value pointed to by `self`. and removes it from the collection. Undefined if self is not referencing a value within the list. Leaves self referencing the prior item, or in `reset` condition.

```
T*
removeNext(const T*);
```

Returns and removes the first element `t`, encountered by iterating self forward, such that the expression `(*t == *a)` is `true`. Leaves self referencing the prior item, or in `reset` condition.

```
T*
removeNext(RWBoolean(*fn)(T*, void*), void* d);
```

Returns and removes the first element `t,` encountered by iterating self forward, such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`. Leaves self referencing the prior item, or in `reset` condition.

```
void
reset();
void
reset(RWTPtrSlist<T>& l);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrSlist` to `reset()` will reset the iterator on the new container.

Synopsis

```
#include <rw/tpsrtdli.h>
RWTPtrSortedDlist<T,C> srtdlist;
```

Standard C++ Library Dependent!

RWTPtrSortedDlist **requires the Standard C++ Library.**

Description

This class maintains an always-sorted pointer-based collection of values, implemented as a doubly-linked list. Items are ordered according to a comparison object of type ***C***. Class ***T*** is the type pointed to by the items in the collection. ***C*** must induce a total ordering on elements of type ***T*** via a public member

```
bool operator()(const T& x, const T& y)
```

which returns `true` if `x` should precede `y` within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example. Note that items in the collection will be dereferenced before being compared.

Persistence

Isomorphic.

Example

In this example, a sorted doubly-linked list of *RW**Date***s is exercised.

```
//
// tpsrtdli.cpp
//
#include <rw/tpsrtdli.h>
#include <rw/rwdate.h>
#include <iostream.h>

main(){
  RWTPtrSortedDList<RWDate,greater<RWDate> > lst;

  lst.insert(new RWDate(10, "Aug", 1991));
  lst.insert(new RWDate(9, "Aug", 1991));
  lst.insert(new RWDate(1, "Sep", 1991));
  lst.insert(new RWDate(14, "May", 1990));
  lst.insert(new RWDate(1, "Sep", 1991));   // Add a duplicate
  lst.insert(new RWDate(2, "June", 1991));

  for (int i=0; i<lst.entries(); i++)
    cout << *lst[i] << endl;

  lst.clearAndDestroy();

  return 0;
}
```

Program Output:
```
09/01/91
09/01/91
08/10/91
08/09/91
06/02/91
05/14/90
```

Related Classes

Class ***RWTPtrSortedVector<T>*** is an alternative always-sorted pointer-based collection. ***RWTPtrDlist<T>*** is an unsorted pointer-based doubly-linked list.

Class ***list<T*,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef rw_deref_compare<C,T>                  container_comp;
typedef list<T*,allocator>                     container_type;
typedef container_type::size_type              size_type;
typedef container_type::difference_type        difference_type;
typedef container_type::const_iterator         const_iterator;
typedef container_type::iterator               iterator;
typedef T*                                     value_type;
typedef T*&                                    reference;
typedef T* const&                              const_reference;
```

Public Constructors

`RWTPtrSortedDlist<T,C>``();`
Constructs an empty doubly-linked list.

`RWTPtrSortedDlist<T,C>``(const RWTPtrSortedDlist<T,C>& lst);`
Copy constructor.

`RWTPtrSortedDlist<T,C>``(const list<T*,allocator>& lst);`
Constructs a doubly-linked list by iterating over all elements in `lst` and performing an order preserving insertion on self for each.

`RWTPtrSortedDlist<T,C>``(size_type n, T* p);`
Constructs a doubly-linked list with `n` elements, each initialized to `p`.

`RWTPtrSortedDlist<T,C>``(T** first,T** last);`
Constructs a doubly-linked list by copying and sorting elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`bool`
`operator<``(const RWTPtrSortedDlist<T,C>& lst) const;`
Returns `true` if self compares lexicographically less than `lst`, otherwise returns `false`. Items in each collection are dereferenced before being compared.

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

Returns `true` if self compares equal to `lst`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

```
reference
operator()(size_type i);
const_reference
operator()(size_type I) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
reference
operator[](size_type I);
const_reference
operator[](size_type I) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
apply(void (*fn)(T*&,void*), void* d);
void
apply(void (*fn)(T*,void*), void* d);
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(const T* a, void* d);
void yourfun(T* a, void* d);
void yourfun(T* &a,void* d)
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains(const T* a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(*t == *a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
const T*
find(const T* a) const;
```

If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
const T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
const_reference
first() const;
```

Returns a reference to the first element of self.

```
size_type
index(const T* a) const;
```

Returns the position of the first item `t` in self such that `(*t == *a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const T*,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
insert(const list<T*,allocator>& a);
```

Adds the items from `a` to self in an order preserving way. Returns the number of items inserted.

```
bool
insert(T* a);
```

Adds the item a to self. The collection remains sorted. Returns `true`.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isSorted() const;
```

Returns `true` if the collection is sorted relative to the supplied comparator object, `false` otherwise.

```
T*&
last();
T* const&
last() const;
```

Returns a reference to the last item in the collection.

```
size_type
merge(const RWTPtrSortedDlist<T,C>& dl);
```

Inserts all elements of `dl` into self, preserving sorted order. Returns the number of items inserted.

```
size_type
occurrencesOf(const T* a) const;
```

Returns the number of elements `t` in self such that the expression `(*t == *a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
remove(const T* a);
```

Removes and returns the first element `t` in self such that the expression `(*t == *a)` is `true`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self such that the expression `(*t == *a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T*
removeFirst();
```

Removes and returns the first item in the collection.

```
T*
removeLast();
```

Removes and returns the first item in the collection.

```
const list<T*,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTPtrSortedDlist<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrSortedDlist<T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSortedDlist<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrSortedDlist<T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSortedDlist<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrSortedDlist<T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tpsrtdli.h>
RWTPtrSortedDlist<T,C> dl;
RWTPtrSortedDlistIterator<T,C> itr(dl);
```

Standard C++ Library Dependent!

RWTPtrSortedDlistIterator **requires the Standard C++ Library.**

Description

RWTPtrSortedDlistIterator is supplied with *Tools.h++* 7.x to provide an iterator interface to the new Standard Library based collections that has backward compatibility with the container iterators provided in *Tools.h++* 6.x.

The order of iteration over an ***RWTPtrSortedDlist*** is dependent on the comparator object parameter C as applied to the values stored in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTPtrSortedDlist<RWCString,less<RWCString> > a;
   RWTPtrSortedDlistIterator<RWCString,less<RWCString> > itr(a);
   a.insert(new RWCString("John"));
   a.insert(new RWCString("Steve"));
   a.insert(new RWCString("Mark"));
   a.insert(new RWCString("Steve"));

   for(;itr();)
     cout << *itr.key() <<endl;

   return 0;
}
```

Program Output

```
John
Mark
Steve
Steve
```

Public Constructors

```
RWTPtrSortedDlistIterator<T,C>(RWTPtrSortedDlist<T,C>& l);
```

Creates an iterator for the list `l`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

```
T*
operator()();
```

Advances self to the next element, dereferences the resulting iterator and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a `nil` pointer equivalent to boolean `false`.

```
RWBoolean
operator++();
```

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the list, self will now point to an undefined value and a value equivalent to `false` will be returned. Otherwise, a value equivalent to `true` is returned. Note: no post-increment operator is provided.

```
RWBoolean
operator+=(size_type n);
```

Behaves as if `operator++()` had been applied `n` times.

```
RWBoolean
operator--();
```

Moves `self` back to the immediately previous element. If the iterator has been reset or just created, this operator will return `false`, otherwise it will return `true`. If `self` references the the first element, it will now be in the reset state. If `self` has been iterated past the last value in the list, it will now reference the last item in the list. Note: no post-decrement operator is provided.

```
RWBoolean
operator-=(size_type n);
```

Behaves as if `operator--()` had been applied `n` times

Public Member Functions

```
RWTPtrSortedDlist<T,C>*
container() const;
```

Returns a pointer to the collection being iterated over.

```
T*
findNext(const T* a);
```

Returns the first element `t` encountered by iterating self forward, such that the expression `(*t == *a)` is `true`. Otherwise returns `nil`. Leaves self referencing found item or “off the end.”

```
T*
findNext(RWBoolean(*fn)(T*, void*), void* d);
```

Returns the first element `t` encountered by iterating self forward such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`. Otherwise returns nil. Leaves self referencing found item or “off the end.”

```
T*
key();
```

Returns the stored value pointed to by `self`. Undefined if self is not referencing a value within the list.

```
T*
remove();
```

Returns the stored value pointed to by `self`. and removes it from the collection. Undefined if self is not referencing a value within the list. Leaves self referencing prior item or in reset state.

```
T*
removeNext(const T*);
```

Returns and removes the first element `t`, encountered by iterating self forward, such that the expression `(*t == *a)` is `true`. Otherwise returns `nil`. Leaves self referencing prior item or in reset state.

```
T*
removeNext(RWBoolean(*fn)(T*, void*), void* d);
```

Returns and removes the first element `t,` encountered by iterating self forward, such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`. Otherwise returns `nil`. Leaves self referencing prior item or in reset state.

```
void
reset();
void
reset(RWTPtrSortedDlist<T,C>& l);
```

Resets the iterator so that after being advanced it will point to the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTPtrSortedDlist` to `reset()` will reset the iterator on the new container.

Synopsis

```
#include <rw/tpsrtvec.h>
RWTPtrSortedVector<T,C> srtvec;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface for *RWTPtrSortedVector* described in Appendix A.

Description

This class maintains an always-sorted pointer-based collection of values, implemented as a vector. Items are ordered according to a comparison object of type ***C***. Class ***T*** is the type pointed to by the items in the collection. ***C*** must induce a total ordering on elements of type ***T*** via a public member

```
bool operator()(const T& x, const T& y)
```

which returns `true` if `x` should precede `y` within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example. Note that items in the collection will be dereferenced before being compared.

Persistence

Isomorphic.

Example

In this example, a sorted vector of *RWDate*s is exercised.

```
//
// tpsrtvec.cpp
//
#include <rw/rwdate.h>
#include <rw/tpsrtvec.h>
#include <iostream.h>

main(){
  RWTPtrSortedVector<RWDate, greater<RWDate> > vec;

  vec.insert(new RWDate(10, "Aug", 1991));
  vec.insert(new RWDate(9, "Aug", 1991));
  vec.insert(new RWDate(1, "Sep", 1991));
  vec.insert(new RWDate(14, "May", 1990));
  vec.insert(new RWDate(1, "Sep", 1991));   // Add a duplicate
  vec.insert(new RWDate(2, "June", 1991));

  for (int i=0; i<vec.entries(); i++)
    cout << *vec[i] << endl;

  vec.clearAndDestroy();

  return 0;
```

```
}
```
Program Output:
```
09/01/91
09/01/91
08/10/91
08/09/91
06/02/91
05/14/90
```

Related Classes

RWTPtrSortedDlist<T,C> is an alternative always-sorted pointer-based collection. ***RWTPtrOrderedVector<T>*** is an unsorted pointer-based vector.

Class ***vector<T*,allocator>*** is the Standard C++ Library collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef vector<T*,allocator>                  container_type;
typedef rw_deref_compare<C,T>                 container_comp;
typedef container_type::const_iterator        const_iterator;
typedef container_type::const_iterator        iterator;
typedef container_type::size_type             size_type;
typedef container_type::difference_type       difference_type;
typedef T*                                    value_type;
typedef T*&                                   reference;
typedef T* const&                             const_reference;
```

Public Constructors

RWTPtrSortedVector<T,C>`();`
Constructs an empty vector.

RWTPtrSortedVector<T,C>`(const vector<T*,allocator>& vec);`
Constructs a vector by copying and sorting all elements of `vec`.

RWTPtrSortedVector<T,C>`(const RWTPtrSortedVector<T,C>& rwvec);`
Copy constructor.

RWTPtrSortedVector<T,C>`(size_type n, T* p);`
Constructs a vector with `n` elements, each initialized to `p`.

RWTPtrSortedVector<T,C>`(size_type n);`
Constructs an empty vector with a capacity of `n` elements.

RWTPtrSortedVector<T,C>`(T** first,T** last);`
Constructs a vector by copying and sorted elements from the array of `T*`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`bool`
operator<`(const RWTPtrSortedVector<T,C>& vec) const;`
Returns `true` if self compares lexicographically less than `vec`, otherwise returns `false`. Items in each collection are dereferenced before being compared.

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

Returns `true` if self compares equal to `vec`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other. Elements are dereferenced before being compared.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
apply(void (*fn)(T*,void*), void* d);
void
apply(void (*fn)(T*&,void*), void* d);
void
apply(void (*fn)(const T*,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(T* a, void* d);
void yourfun(T*& a, void* d);
void yourfun(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```
Returns an iterator positioned at the first element of self.

```
void
clear();
```
Clears the collection by removing all items from self.

```
void
clearAndDestroy();
```
Removes all items from the collection *and* uses `operator delete` to destroy the objects pointed to by those items. Do not use this method if multiple pointers to the same object are stored.

```
bool
contains(const T* a) const;
```
Returns `true` if there exists an element `t` in self such that the expression`(*t == *a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const T*,void*), void* d) const;
```
Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T* const*
data() const;
```
Returns a pointer to the first element of the vector.

```
iterator
end();
const_iterator
end() const;
```
Returns an iterator positioned “just past” the last element in self.

```
size_type
entries() const;
```
Returns the number of items in self.

```
const T*
find(const T* a) const;
```
If there exists an element `t` in self such that the expression `(*t == *a)` is `true`, returns `t`. Otherwise, returns `rwnil`.

```
const T*
find(bool (*fn)(const T*,void*), void* d) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, returns `t`. Otherwise, returns `rwnil`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
const_reference
first() const;
```

Returns a reference to the first element of self. If the collection is empty, the function throws an exception of type *RW**BoundsErr***.

```
size_type
index(const T* a) const;
```

Returns the position of the first item `t` in self such that `(*t == *a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const T*,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(T* a);
```

Adds the item a to self. The collection remains sorted. Returns `true`.

```
size_type
insert(const vector<T*,allocator>& a);
```

Inserts all elements of `a` into self. The collection remains sorted. Returns the number of items inserted.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isSorted() const;
```

Returns `true` if the collection is sorted relative to the supplied comparator object, `false` otherwise.

```
T*&
last();
T* const&
last() const;
```

Returns a reference to the last item in the collection. If the collection is empty, the function throws an exception of type *RW**BoundsErr***.

```
size_type
length() const;
```

Returns the number of elements in self.

```
size_type
merge(const RWTPtrSortedVector<T,C>& vec);
```

Inserts all elements of `vec` into self, preserving sorted order. Returns the number of items inserted.

```
size_type
occurrencesOf(const T* a) const;
```

Returns the number of elements `t` in self such that the expression `(*t == *a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const T*,void*), void* d) const;
```

Returns the number of elements `t` in self such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
remove(const T* a);
```

Removes and returns the first element `t` in self such that the expression `(*t == *a)` is `true`. Returns `rwnil` if there is no such element.

```
T*
remove(bool (*fn)(const T*,void*), void* d);
```

Removes and returns the first element `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns `rwnil` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const T* a);
```

Removes all elements `t` in self such that the expression `(*t == *a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const T*,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T* a, void* d);
```

Client data may be passed through parameter `d`.

```
T*
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T*
removeFirst();
```

Removes and returns the first item in the collection. If the collection is empty, the function throws an exception of type *RW**BoundsErr***.

```
T*
removeLast();
```

Removes and returns the first item in the collection.

```
void
resize(size_type n);
```

Modify, if necessary, the capacity of the vector to be at least as large as `n`.

```
const vector<T*,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
           const RWTPtrSortedVector<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrSortedVector<T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSortedVector<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrSortedVector<T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSortedVector<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrSortedVector<T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

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 pointers must 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.

Persistence

Isomorphic

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;
    delete week[i];
  }

  return 0;
}
```

Program output:

```
March 16, 1996
March 17, 1996
March 18, 1996
March 19, 1996
March 20, 1996
March 21, 1996
March 22, 1996
```

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 are 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 pointing to the item `*ival`.

```
RWTPtrVector<T>(const RWTPtrVector& v);
```
Constructs self as a shallow copy of `v`. After construction, pointers held by the two vectors point to the same items.

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 pointersheld by the two vectors will point to the same items.

```
RWTPtrVector<T>&
operator=(T* p);
```
Sets all elements in self to point to the item `*p`.

```
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 length of the vector, less one. No bounds checking is performed.

```
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 lvalue, 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_t
length() const;
```
Returns the length of the vector.

```
void
reshape(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.

```
void
resize(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 `nil`.

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 *RWT**ValSlist<T>*** or *RWT**ValDlist<T>.*** Vectors, such as *RWT**ValOrderedVector<T>,*** can also be used, but tend to be less efficient at removing an object from the front of the list.

Persistence

None

Example

In this example a queue of *RW**CString***s, 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 Member Functions

```
void
clear();
```
Removes all items from the queue.

```
size_t
entries() const;
```
Returns the number of items in the queue.

```
T
first() const;
```
Returns, but does not remove, the first item in the queue (the item least recently inserted into the queue).

```
T
get();
```
Returns and removes the first item in the queue (the item least recently inserted into the queue).

```
RWBoolean
isEmpty() const;
```
Returns `TRUE` if there are no items in the queue, otherwise `FALSE`.

```
void
insert(T a);
```
Inserts the item `a` at the end of the queue.

```
T
last() const;
```
Returns, but does not remove, the last item in the queue (the item most recently inserted into the queue).

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 *RW**TValOrderedVector<T>*** or *RW**TValDlist<T>.*** Class *RW**TValSlist<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.

Persistence

None

Example

In this example a stack of `int`s, 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 Member Functions

```
void
clear();
```
Removes all items from the stack.

```
size_t
entries() const;
```
Returns the number of items currently on the stack.

```
RWBoolean
isEmpty() const;
```
Returns `TRUE` if there are currently no items on the stack, `FALSE` otherwise.

```
void
push(T a);
```
Push the item `a` onto the top of the stack.

```
T
pop();
```
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.

```
T
top() const;
```
Returns (but does not remove) the item at the top of the stack.

Synopsis

```
#include <rw/tvdeque.h>
RWTValDeque<T> deq;
```

Standard C++ Library Dependent!

***RWTValDeque* requires the Standard C++ Library.**

Description

This class maintains a collection of values implemented as a double-ended queue, or *deque*. Order is determined externally and elements are accessible by index. Use this class when insertions and deletions usually occur at either the beginning or the end of the collection.

Persistence

Isomorphic

Example

In this example, a double-ended queue of `ints` is exercised.

```
//
// tvdqint.cpp
//
#include <rw/tvdeque.h>
#include <iostream.h>

/*
 * This program partitions integers into even and odd numbers
 */

int main(){
  RWTValDeque<int> numbers;

  int n;

  cout << "Input an assortment of integers (EOF to end):"
       << endl;

  while (cin >> n) {
    if (n % 2 == 0)
      numbers.pushFront(n);
    else
      numbers.pushBack(n);
  }

  while (numbers.entries()) {
    cout << numbers.popFront() << endl;
  }

  return 0;
}
```

Program Input:

```
1 2 3 4 5
<eof>
```

Program Output:

```
4
2
1
3
5
```

Related Classes

Classes *RWTValSlist<T>*, *RWTValDlist<T>*, *RWTValSortedDlist<T>*, and *RWTValOrderedVector<T>* also provide a Rogue Wave interface to C++-standard sequence collections. The list classes should be considered for frequent insertions (or removals) in the interior of the collection. The vector may be more efficient if most insertions and removals occur at the end of the collection.

Class ***deque<T,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef deque<T,allocator>                    container_type;
typedef container_type::iterator              iterator;
typedef container_type::const_iterator        const_iterator;
typedef container_type::size_type             size_type;
typedef T                                     value_type;
typedef T&                                    reference;
typedef const T&                              const_reference;
```

Public Constructors

`RWTValDeque<T>``();`

Constructs an empty, double-ended queue.

`RWTValDeque<T>``(const deque<T,allocator>& deq);`

Constructs a double-ended queue by copying all elements of `deq`.

`RWTValDeque<T>``(const RWTValDeque<T>& rwdeq);`

Copy constructor.

`RWTValDeque<T>``(size_type n, const T& val = T());`

Constructs a double-ended queue with `n` elements, each initialized to `val`.

`RWTValDeque<T>``(const T* first, const T* last);`

Constructs a double-ended queue by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

```
RWTValDeque<T>&
operator=(const RWTValDeque<T,allocator>& deq);
RWTValDeque<T>&
operator=(const deque<T>& deq);
```

Calls the destructor on all elements of self and replaces them by copying all elements of `deq`.

```
bool
operator<(const RWTValDeque<T>& deq) const;
bool
operator<(const deque<T,allocator>& deq) const;
```

Returns `true` if self compares lexicographically less than `deq`, otherwise returns `false`. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

```
bool
operator==(const RWTValDeque<T>& deq) const;
bool
operator==(const deque<T,allocator>& deq) const;
```

Returns `true` if self compares equal to `deq`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between `0` and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed.*

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(const_reference a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(reference,void*), void* d);
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(const_reference a, void* d);
void yourfun(reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between `0` and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(t == a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns a *past-the-end* valued iterator of self.

```
size_type
entries() const;
```

Returns the number of elements in self.

```
bool
find(const_reference a,T& k) const;
```

If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d, T& k) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
```

```
const_reference
first() const;
```

Returns a reference to the first element of self.

```
size_type
index(const_reference a) const;
```

Returns the position of the first item `t` in self such that `(t == a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const_reference,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```
Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, const_reference a);
```
Inserts the item `a` in front of the item at position `i` in self. This position must be between `0` and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
reference
last();
```
```
const_reference
last() const;
```
Returns a reference to the last item in the collection.

```
reference
maxElement();
```
```
const_reference
maxElement() const;
```
```
reference
minElement();
```
```
const_reference
minElement() const;
```
Returns a reference to the minimum or maximum element in the collection. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

```
size_type
occurrencesOf(const_reference a) const;
```
Returns the number of elements `t` in self such that the expression `(t == a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),void* d) const;
```
Returns the number of elements `t` in self such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

`void`
`prepend``(const_reference a);`
Adds the item `a` to the beginning of the collection.

`T`
`popBack``();`
Removes and returns the last item in the collection.

`T`
`popFront``();`
Removes and returns the first item in the collection.

`void`
`pushBack``(const_reference a);`
Adds the item `a` to the end of the collection.

`void`
`pushFront``(const_reference a);`
Adds the item `a` to the beginning of the collection.

`bool`
`remove``(const_reference a);`
Removes the first element `t` in self such that the expression `(t == a)` is `true` and returns `true`. Returns `false` if there is no such element.

`bool`
`remove``(bool (*fn)(const_reference,void*), void* d);`
Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

`size_type`
`removeAll``(const_reference a);`
Removes all elements `t` in self such that the expression `(t == a)` is `true`. Returns the number of items removed.

`size_type`
`removeAll``(bool (*fn)(const_reference,void*), void* d);`
Removes all elements `t` in self such that the expression `((*fn)(t,d))` is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
T
removeAt(size_type i);
```
Removes and returns the item at position `i` in self. This position must be between `0` and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T
removeFirst();
```
Removes and returns the first item in the collection.

```
T
removeLast();
```
Removes and returns the first item in the collection.

```
size_type
replaceAll(const T& oldVal, const T& newVal);
```
Replaces all elements `t` in self such that the expression `(t == oldVal)` is `true` with `newVal`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(const T&,void*), void* d,
       const T& newVal);
```
Replaces all elements `t` in self such that the expression `((*fn)(t,d))`is `true`.with `newVal` Returns the number of items replaced. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T& a, void* d);
```

Client data may be passed through parameter `d`.

```
void
sort();
```
Sorts the collection using the less-than operator (`<`) to compare elements.

```
deque<T,allocator>&
std();
const deque<T,allocator>&
std() const;
```
Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of C++-standard collections.

Static Public Data Member

```
const size_type  npos;
```
This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValDeque<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValDeque<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValDeque<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValDeque<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValDeque<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValDeque<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tvdlist.h>
RWTValDlist<T> dlist;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValDlist* described in Appendix A.

Description

This class maintains a collection of values, implemented as a doubly-linked list.

Persistence

Isomorphic

Example

In this example, a doubly-linked list of user type `Dog` is exercised.

```
//
// tvdldog.cpp
//
#include <rw/tvdlist.h>
#include <iostream.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) {
   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; }
```

```
  // order alphabetically:
  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(){
  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
```

Related Classes

Classes *RWTValDeque<T>*, *RWTValSlist<T>*, and *RWTValOrderedVector<T>* also provide a Rogue Wave interface to C++-standard sequence collections.

Class ***list<T,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef list<T,allocator>                        container_type;
typedef container_type::iterator                 iterator;
typedef container_type::const_iterator           const_iterator;
typedef container_type::size_type                size_type;
typedef T                                        value_type;
typedef T&                                       reference;
typedef const T&                                 const_reference;
```

Public Constructors

`RWTValDlist<T>();`

Constructs an empty, doubly-linked list.

RWTValDlist<T>(const list<T,allocator>& lst);
Constructs a doubly-linked list by copying all elements of lst.

RWTValDlist<T>(const RWTValDlist<T>& rwlst);
Copy constructor.

RWTValDlist<T>(size_type n, const T& val = T());
Constructs a doubly-linked list with n elements, each initialized to val.

RWTValDlist<T>(const T* first, const T* last);
Constructs a doubly-linked list by copying elements from the array of Ts pointed to by first, up to, but not including, the element pointed to by last.

Public Member Operators

RWTValDlist<T>&
operator=(const RWTValDlist<T>& lst);

RWTValDlist<T>&
operator=(const list<T,allocator>& lst);
Calls the destructor on all elements of self and replaces them by copying all elements of lst.

bool
operator<(const RWTValDlist<T>& lst) const;

bool
operator<(const list<T,allocator>& lst) const;
Returns true if self compares lexicographically less than lst, otherwise returns false. Type T must have well-defined less-than semantics (T::operator<(const T&) or equivalent).

bool
operator==(const RWTValDlist<T>& lst) const;

bool
operator==(const list<T,allocator>& lst) const;
Returns true if self compares equal to lst, otherwise returns false. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

reference
operator()(size_type i);

const_reference
operator()(size_type i) const;
Returns a reference to the ith element of self. Index i should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed.*

```
reference
operator[](size_type i);
```

```
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RWBoundsErr*.

Public Member Functions

```
void
append(const_reference a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(reference,void*), void* d);
```

```
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(const_reference a, void* d);
void yourfun(reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
```

```
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RWBoundsErr*.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(t == a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression ((*fn)(t,d)) is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns a *past*-the-end valued iterator of self.

```
size_type
entries() const;
```

Returns the number of elements in self.

```
bool
find(const_reference a, T& k) const;
```

If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d, T& k) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
```

```
const_reference
first() const;
```

Returns a reference to the first element of self.

```
T
get();
```

Removes and returns the first element in the collection. If the collection is empty, the function throws an exception of type *RW**BoundsErr***. This method is identical to `removeFirst` and is included for compatibility with previous versions.

```
size_type
index(const_reference a) const;
```

Returns the position of the first item `t` in self such that `(t == a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const_reference,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```

Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i,const_reference a);
```

Inserts the item `a` in front of the item at position `i` in self. This position must be between 0 and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
reference
last();
const_reference
last() const;
```

Returns a reference to the last item in the collection.

```
reference
maxElement();
const_reference
maxElement() const;
reference
minElement();
const_reference
minElement() const;
```

Returns a reference to the minimum or maximum element in the collection. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

```
size_type
occurrencesOf(const_reference a) const;
```

Returns the number of elements `t` in self such that the expression `(t == a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
prepend(const_reference a);
```

Adds the item `a` to the beginning of the collection.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self such that the expression `(t == a)` is `true` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self such that the expression `(t == a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
T
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between 0 and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T
removeFirst();
```

Removes and returns the first item in the collection.

```
T
removeLast();
```

Removes and returns the first item in the collection.

```
size_type
replaceAll(const_reference oldVal, const_reference newVal);
```

Replaces all elements `t` in self such that the expression `(t == oldVal)` is `true` with `newVal`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(const_reference,void*), void* d,
           const value_type& newval);
```

Replaces all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
sort();
```

Sorts the collection using the less-than operator to compare elements.

```
list<T,allocator>&
std();
```

```
const list<T>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValDlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValDlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValDlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValDlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValDlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValDlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvdlist.h>
RWTValDlist<T> dl;
RWTValDlistIterator<T> itr(dl);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValDlistIterator* described in Appendix A.

Description

RWTValDlistIterator provides an iterator interface to the *Tools.h++* 7 Standard Library based collections which is compatible with the iterator interface provided for the *Tools.h++* 6.x containers.

The order of iteration over an ***RWTValDlist*** is dependent on the order of insertion of the values into the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equal to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTValDlist<RWCString> a;
   RWTValDlistIterator<RWCString> itr(a);

   a.insert("John");
   a.insert("Steve");
   a.insert("Mark");
   a.insert("Steve");

   for(;itr();)
     cout << itr.key() << endl;

   return 0;
}
```

Program Output

```
John
Steve
Mark
Steve
```

Public Constructors

RWTValDlistIterator<T>(RWTValDlist<T>& s);

Creates an iterator for the dlist `s`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

RWBoolean
operator()();

Advances self to the next element and returns its value. If the iterator has advanced past the last item in the container, the element returned will be a nil pointer equivalent to boolean `false`.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created, self will reference the first element. If, before iteration, self referenced the last value in the list, self will now reference an undefined value distinct from the reset value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

RWBoolean
operator+=(size_type n);

Behaves as if the `operator++` member function had been applied `n` times

RWBoolean
operator--();

Moves self back to the immediately previous element. If the iterator has been reset or just created, this operator will return `false`, otherwise it will return `true`. If self references the the first element, it will now be in the reset state. If self has been iterated past the last value in the list, it will now reference the last item in the list. Note: no postdecrement operator is provided.

RWBoolean
operator-=(size_type n);

Behaves as if the `operator--` member function had been applied `n` times

Public Member Functions

RWTValDlist<T>*
container() const;

Returns a pointer to the collection being iterated over.

```
RWBoolean
findNext(const T& a);
```

Advances self to the first element `t` encountered by iterating forward, such that the expression `(t == a)` is `true`. Returns `true` if an element was found, returns `false` otherwise.

```
RWBoolean
findNext(RWBoolean(*fn)(const T&, void*), void* d);
```

Advances self to the first element `t` encountered by iterating forward such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T a, void* d);
```

Client data may be passed through parameter `d`. Returns `true` if an element was found, returns `false` otherwise.

```
T
key();
```

Returns the stored value referenced by `self`.

```
RWBoolean
remove();
```

Removes the value referenced by `self` from the collection. `true` is returned if the removal is successful, `false` is returned otherwise.

```
RWBoolean
removeNext(const T);
```

Removes the first element `t`, encountered by iterating self forward, such that the expression `(t == a)` is `true`. Returns `true` if an element was found and removed, returns `false` otherwise.

```
RWBoolean
removeNext(RWBoolean(*fn)(T, void*), void* d);
```

Removes the first element `t,` encountered by iterating self forward, such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T a, void* d);
```

Client data may be passed through parameter `d`. Returns `true` if an element was found and removed, returns `false` otherwise.

```
void
reset();
void
reset(RWTValDlist<T>& l);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValDlist` to `reset()` will reset the iterator on the new container.

Synopsis

```
#define RWTValHashDictionary RWTValHashMap
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWT*_ValHashMap_. Although the old name (*RWT*_ValHashDictionary_) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWT*_ValHashDictionary_ in Appendix A.

Synopsis

```
#define RWTValHashDictionaryIterator RWTValHashMapIterator
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTValHashMapIterator*. Although the old name (*RWTValHashDictionaryIterator*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTValHashDictionaryIterator* in Appendix A.

Synopsis

```
#include <rw/tvhdict.h>
RWTValHashMap<K,T,H,EQ> m;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTValHashDictionary* described in Appendix A.

Description

This class maintains a collection of keys, each with an associated item of type `T`. These pairs are stored according to a hash object of type `H`. `H` must provide a hash function on elements of type `K` via a public member

```
unsigned long operator()(const K& x)
```

Equivalent keys within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const K& x, const K& y)
```

which should return `true` if `x` and `y` are equivalent.

RWTValHashMap<K,T,H,EQ> will not accept a key that compares equal to any key already in the collection. (*RWTValHashMultiMap<K,T,H,EQ>* may contain multiple keys that compare equal to each other.) Equality is based on the equality object and *not* on the `==` operator.

Persistence

Isomorphic

Related Classes

Class *RWTValHashMultiMap<K,T,H,EQ>* offers the same interface to a collection that accepts multiple keys that compare equal to each other.

Class *rw_hashmap<K,T,H,EQ>* is the C++-standard compliant collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef rw_hashmap<K,T,H,EQ>                      container_type;
typedef container_type::iterator                  iterator;
typedef container_type::const_iterator            const_iterator;
typedef container_type::size_type                 size_type;
typedef pair <const K,T>                          value_type;
typedef K                                         key_type;
typedef T                                         data_type;
typedef pair <const K,T>&                         reference;
typedef pair <const K,T>&                         const_reference;
```

Public Constructors

RWTValHashMap<K,T,H,EQ>();

Constructs an empty map.

RWTValHashMap<K,T,H,EQ>(const rw_hashmap<K,T,H,EQ>& m);

Constructs a map by copying all elements of `m`.

RWTValHashMap<K,T,H,EQ>
(const H& h, size_type sz = RWDEFAULT_CAPACITY);

Creates an empty hashed map which uses the hash object `h` and has an initial capacity of `sz`.

RWTValHashMap<K,T,H,EQ>(const RWTValHashMap<K,T,H,EQ>& rwm);

Copy constructor.

RWTValHashMap<K,T,H,EQ>(const value_type* first,
    const value_type* last);

Constructs a map by copying elements from the array of `value_type` pairs pointed to by `first`, up to, but not including, the pair pointed to by `last`.

Public Member Operators

RWTValHashMap<K,T,H,EQ>&
operator=(const RWTValHashMap<K,T,H,EQ>& m);

RWTValHashMap<K,T,H,EQ>&
operator=(const rw_hashmap<K,T,H,EQ>& m);

Destroys all elements of self and replaces them by copying all associations from `m`.

bool
operator==(const RWTValHashMap<K,T,H,EQ>& m) const;

bool
operator==(const rw_hashmap<K,T,H,EQ>& m) const;

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual pairs that compare equal to each other.

T&
operator[](const K& key);

Looks up `key` and returns a reference to its associated item. If the key is not in the dictionary, then it will be added with an associated item provided by the default constructor for type `T`.

Public Member Functions

void
apply(void (*fn)(const K&, T&, void*),void* d);

void
apply(void (*fn)(const K&,const T&,void*),void* d) const;

Applies the user-defined function pointed to by `fn` to every association in the collection. This function must have one of the prototypes:

```
void yourfun(const K& key, T& a, void* d);
```

```
void yourfun(const K& key, const T& a,void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K&, T&,void*),void* d);
void
applyToKeyAndValue
(void (*fn)(const K&, const T, void*),void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply.**

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
size_type
capacity() const;
```

Returns the number of buckets(slots) available in the underlying hash representation. See **resize** below.

```
void
clear();
```

Clears the collection by removing all items from self. Each key and its associated item will have its destructor called.

```
bool
contains(const K& key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `key`; otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
float
fillRatio() const;
```

Returns the ratio `entries()`/`capacity()`.

```
bool
find(const K& key, K& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
find(bool (*fn)(const_reference,void*),void* d,
                pair<K,T>& r) const;
```

If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, assigns `a` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const K& a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
findValue(const K& key, T& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns the item associated with `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
findKeyValue(const K& key, K& kr, T& tr) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `kr`, assigns the item associated with `j` to `tr`, and returns `true`. Otherwise, returns `false` and leaves the values of `kr` and `tr` unchanged.

```
bool
insert(const K& key, const T& a);
```

Adds `key` with associated item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an association with the equivalent key.

```
bool
insertKeyAndValue(const K& key,const T& a);
```

This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert.**

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K& key) const;
```
Returns the number of keys `j` in self that compare equal to `key`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),void* d) const;
```
Returns the number of associations `a` in self such that the expression `((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const K& key);
```
Removes the first association with key `j` in self such that the expression `(j == key)` is `true` and returns `true`. Returns `false` if there is no such association.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```
Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const K& key);
```
Removes all elements `j` in self that compare equal to `key`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```
Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```

Changes the capacity of self by creating a new hashed map with a capacity of `sz` . **resize** copies every element of self into the new container and finally swaps the internal representation of the new container with the internal representation of `self`.

```
rw_hashmap<K,T,H,EQ>&
std();
```

```
const rw_hashmap<K,T,H,EQ>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard compliant collections.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTValHashMap<K,T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValHashMap<K,T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashMap<K,T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm, RWTValHashMap<K,T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashMap<K,T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm, RWTValHashMap<K,T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvhdict.h>
RWTValHashMap<K,T,H,EQ> m;
RWTValHashMap<K,T,H,EQ> itr(m);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTValHashDictionaryIterator* described in Appendix A.

Description

RWTValHashMapIterator is supplied with Tools 7 to provide an iterator interface to *RWTValHashMapIterator* that has backward compatibility with the container iterators provided in Tools 6.

Iteration over an *RWTValHashMap* is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used. The only useable relationship between consecutive elements is that elements which are defined to be equivalent by the equivalence object, `EQ`, will remain adjacent.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or an `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Example

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

struct silly_h{
   unsigned long operator()(const RWCString& x) const
   { return x.length() * (long)x(0); }
};

int main(){
   RWTValHashMap
     <RWCString,int,silly_h,equal_to<RWCString> > age;
   RWTValHashMapIterator
     <RWCString, int, silly_h, equal_to<RWCString > > itr(age);

   age.insert(RWCString("John"), 30);
   age.insert(RWCString("Steve"),17);
```

```
   age.insert(RWCString("Mark"),24);
//Duplicate insertion rejected
   age.insert(RWCString("Steve"),24);
   for(;itr();)
     cout << itr.key() << "\'s age is " << itr.value() << endl;
   return 0;
}
```

Program Output (not necessarily in this order)

```
John’s age is 30
Steve’s age is 17
Mark’s age is 24
```

Public Constructors

RWTValHashMapIterator<K,T,H,EQ>
`(RWTValHashMap<K,T,H,EQ>&h);`

Creates an iterator for the hashmap `h`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

`RWBoolean`
operator()`();`

Advances `self` to the next element. Returns `false` if the iterator has advanced past the last item in the container and `true` otherwise.

`RWBoolean`
operator++`();`

Advances `self` to the next element. If the iterator has been reset or just created `self` will now reference the first element. If, before iteration, `self` referenced the last association in the multimap, `self` will now reference an undefined value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

Public Member Functions

`RWTValHashMap<K,T,H,EQ>*`
container() `const;`

Returns a pointer to the collection being iterated over.

`K`
key`() const;`

Returns the key portion of the association currently pointed to by `self`.

`void`
reset`();`
`void`
reset`(RWTValHashMap<K,T,H,EQ>& h);`

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValHashMap` to `reset()` will reset the iterator on that container.

`T`
value`();`

Returns the value portion of the association referenced by `self`.

Synopsis

```
#include <rw/tvhmmap.h>
RWTValHashMultiMap<K,T,H,EQ> m;
```

Standard C++ Library Dependent!

***RWTValHashMultiMap* requires the Standard C++ Library.**

Description

This class maintains a collection of keys, each with an associated item of type `T`. These pairs are stored according to a hash object of type `H`. `H` must provide a hash function on elements of type `K` via a public member

```
unsigned long operator()(const K& x) const
```

Equivalent keys within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const K& x, const K& y) const
```

which should return `true` if `x` and `y` are equivalent.

RWTValHashMultiMap<K,T,H,EQ> may contain multiple keys that compare equal to each other. (*RWTValHashMap<K,T,H,EQ>* will not accept a key that compares equal to any key already in the collection.) Equality is based on the comparison object and *not* on the `==` operator.

Persistence

Isomorphic.

Examples

```
//
// tvhmmrat.cpp
//
#include<rw/tvhmmap.h>
#include<iostream.h>
#include<rw/cstring.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x[0]; }
};
int main(){
  RWCString trd = "Third";
  RWTValHashMultiMap<RWCString,int,silly_hash,equal_to<RWCString> >
       contest;
  contest.insert("First", 7);
  contest.insert(trd,3);
  contest.insert(trd,6);      // self contains two distinct values

                              //equivalent to trd
  contest.insert("Second",2);
```

```
  contest.resize(8);
  cout << "The table is " << contest.fillRatio() * 100.0
       << "% full<< endl;
  return 0;
}
```

Program Output:
```
The table is 50% full
```

Related Classes

Class *RWTValHashMap<K,T,H,EQ>* offers the same interface to a collection that will not accept multiple keys that compare equal to each other.

Class *rw_hashmultimap<K,T,H,EQ>* is the C++-standard collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef rw_hashmultimap<K,T,H,EQ>               container_type;
typedef container_type::iterator                iterator;
typedef container_type::const_iterator          const_iterator;
typedef container_type::size_type               size_type;
typedef pair <const K,T>                        value_type;
typedef pair <const K,T>&                       reference;
typedef const pair<const K,T>&                  const_reference;
```

Public Constructors

RWTValHashMultiMap<K,T,H,EQ>`();`
Constructs an empty map.

RWTValHashMultiMap<K,T,H,EQ>
`(const rw_hashmultimap<K,T,H,EQ>& m);`
Constructs a map by copying all elements of `m`.

RWTValHashMultiMap<K,T,H,EQ>
`(const RWTValHashMultiMap<K,T,H,EQ>& rwm);`
Copy constructor.

RWTValHashMultiMap<K,T,H,EQ>
`(const value_type* first, const value_type* last);`
Constructs a map by copying elements from the array of association pairs pointed to by `first`, up to, but not including, the association pointed to by `last`.

Public Member Operators

`RWTValHashMultiMap<K,T,H,EQ>&`
operator=`(const RWTValHashMultiMap<K,T,H,EQ>& m);`

`RWTValHashMultiMap<K,T,H,EQ>&`
operator=`(const rw_hashmultimap<K,T,H,EQ>& m);`
Destroys all elements of self and replaces them by copying all associations from `m`.

```
bool
operator==(const RWTValHashMultiMap<K,T,H,EQ>& m) const;
bool
operator==(const rw_hashmultimap<K,T,H,EQ>& m) const;
```

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual keys that compare equal to each other.

Public Member Functions

```
void
apply(void (*fn)(const K&, T&, void*),void* d);
void
apply(void (*fn)(const K&,const T&, void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every association in the collection. This function must have one of the prototypes:

```
void yourfun(const K&, T& a, void* d);
void yourfun(const K&, const T& a,void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K&, T&, void*),void* d);
void
applyToKeyAndValue
(void (*fn)(const K&,const T&,void*), void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply.**

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
size_type
capacity() const;
```

Returns the number of buckets(slots) available in the underlying hash representation. See **resize** below.

```
void
clear();
```

Clears the collection by removing all items from self. Each key and its associated item will have its destructor called.

```
bool
contains(const K& key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `key`, otherwise returns `false`.

```
bool
contains
(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
float
fillRatio() const;
```

Returns the ratio `entries()`/`capacity()`.

```
bool
find(const K& key, Key& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
find (bool (*fn)(const_reference,void*),
     void* d,pair<K,T>& r) const;
```

If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, assigns `a` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
findValue(const K& key, T& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns the item associated with `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
findKeyValue(const K& key, K& kr, T& tr) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `kr`, assigns the item associated with `j` to `tr`, and returns `true`. Otherwise, returns `false` and leaves the values of `kr` and `tr` unchanged.

```
bool
insert(const K& key, const T& a);
```

Adds `key` with associated item `a` to the collection. Returns `true`.

```
bool
insertKeyAndValue(const K& key, const T& a);
```

This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert**.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K& key) const;
```

Returns the number of keys `j` in self that compares equal to `key`.

```
size_type
occurrencesOf
(bool (*fn)(const_reference,void*),void* d) const;
```

Returns the number of associations `a` in self such that the expression`((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const K& key);
```

Removes the first association with key `j` in self such that `j` compares equal to `key` and returns `true`. Returns `false` if there is no such association.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const K& key);
```
Removes all associations with key `j` in self where `j` compares equal to `key`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```
Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```
Changes the capacity of `self` by creating a new hashed multimap with a capacity of `sz` . **resize** then copies every element of `self` into the new container and finally swaps the internal representation of the new container with `self`.

```
rw_hashmultimap<K,T,H,EQ>&
std();
```

```
const rw_hashmultimap<K,T,H,EQ>&
std() const;
```
Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing accessibility to the C++-standard interface and interoperability with other software components that make use of the C++-standard collections.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
      const RWTValHashMultiMap<K,T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm,
      const RWTValHashMultiMap<K,T,H,EQ>& coll);
```
Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm,
      RWTValHashMultiMap<K,T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm, RWTValHashMultiMap<K,T,H,EQ>& coll);
```
Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashMultiMap<K,T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm, RWTValHashMultiMap<K,T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvhmmap.h>
RWTValHashMultiMap<K,T,H,EQ> m;
RWTValHashMultiMapIterator<K,T,H,EQ> itr(m);
```

Standard C++ Library Dependent!

***RWTValHashMultiMapIterator* requires the Standard C++ Library.**

Description

RWTValHashMultiMapIterator is supplied with *Tools.h++* 7 to provide an iterator interface to *RWTValHashMultiMapIterator* that is backward compatible with the container iterators provided in *Tools.h++* 6.x.

Iteration over an *RWTValHashMultiMap* is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used. The only useable relationship between consecutive elements is that elements which are defined to be equivalent by the equivalence object, `EQ`, will remain adjacent.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Example

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

struct silly_h{
   unsigned long operator()(const RWCString& x) const
   { return x.length() * (long)x(0); }
};

int main(){
   RWTValHashMultiMap
     <RWCString,int,silly_h,equal_to<RWCString> > age;
   RWTValHashMultiMapIterator
     <RWCString, int, silly_h, equal_to<RWCString > > itr(age);

   age.insert(RWCString("John"), 30);
   age.insert(RWCString("Steve"),17);
   age.insert(RWCString("Mark"),24);
   age.insert(RWCString("Steve"),24);
```

```
   for(;itr();)
     cout << itr.key() << "\'s age is " << itr.value() << endl;
   return 0;
}
```

Program Output (not necessarily in this order)

```
John's age is 30
Steve's age is 24
Steve's age is 17
Mark's age is 24
```

Public Constructors

RWTValHashMultiMapIterator<K,T,H,EQ>
`(RWTValHashMultiMap<K,T,H,EQ>&h);`

Creates an iterator for the hash multimap `h`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

`RWBoolean`
`operator()``();`

Advances `self` to the next element, dereferences the resulting iterator and returns `false` if the iterator has advanced past the last item in the container and `true` otherwise.

`RWBoolean`
`operator++``();`

Advances `self` to the next element. If the iterator has been reset or just created `self` will now reference the first element. If, before iteration, `self` referenced the last association in the multimap, `self` will now reference an undefined value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

Public Member Functions

`RWTValHashMultiMap<K,T,H,EQ>*`
`container()` `const;`

Returns a pointer to the collection being iterated over.

`K`
`key``() const;`

Returns the key portion of the association currently referenced by `self`.

`void`
`reset``();`
`void`
`reset``(RWTValHashMultiMap<K,T,H,EQ>& h);`

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValHashMultiMap` with `reset()` will reset the iterator on that container.

`T`
`value``();`

Returns the value portion of the association referenced by `self`.

Synopsis

```
#include <rw/tvhasht.h>
RWTValHashMultiSet<T,H,EQ>
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTValHashTable* described in Appendix A.

Description

This class maintains a collection of values, which are stored according to a hash object of type `H`. `H` must offer a hash function for elements of type `T` via a public member

```
unsigned long operator()(const T& x) const
```

Objects within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const T& x, const T& y) const
```

which should return `true` if `x` and `y` are equivalent, `false` otherwise.

RWTValHashMultiSet<T,H,EQ> may contain multiple items that compare equal to each other. (*RWTValHashSet<T,H,EQ>* will not accept an item that compares equal to an item already in the collection.)

Persistence

Isomorphic

Example

```
//
// tvhmsstr.cpp
//
#include <rw/tvhasht.h>
#include <rw/cstring.h>
#include <iostream.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x[0]; }
};

main(){
RWTValHashMultiSet<RWCString,silly_hash,equal_to<RWCString> > set1;
RWTValHashMultiSet<RWCString,silly_hash,equal_to<RWCString> > set2;

  set1.insert("one");
  set1.insert("two");
  set1.insert("three");
  set1.insert("one");     // OK: duplicates allowed
```

```
  set1.insert("one");

  cout << set1.entries() << endl;  // Prints "5"

  set2.insert("one");
  set2.insert("five");
  set2.insert("one");

  cout << ((set1.isEquivalent(set2)) ? "TRUE" : "FALSE") << endl;
  // Prints "FALSE"

  set2.intersection(set1);
  set1.clear();

  cout << set1.entries() << endl;    // Prints "0"
  cout << set2.entries() << endl;    // Prints "2"

  return 0;
}
```

Related Classes

Class *RWTValHashSet<T,H,EQ>* offers the same interface to a collection that will not accept multiple items that compare equal to each other.

Class *rw_hashmultiset<T,H,EQ>* is the C++-standard compliant collection that serves as the underlying implementation for *RWTValHashMultiSet<T,H,EQ>*.

Public Typedefs

```
typedef rw_hashmultiset<T,H,EQ>                   container_type;
typedef container_type::iterator                  iterator;
typedef container_type::const_iterator            const_iterator;
typedef container_type::size_type                 size_type;
typedef T                                         value_type;
typedef T&                                        reference;
typedef const T&                                  const_reference;
```

Public Constructors

RWTValHashMultiSet<T,H,EQ>
`(size_type sz = 1024,const H& h = H(),const EQ& eq = EQ());`

Constructs an empty set. The underlying hash table representation will have `sz` buckets, will use `h` as its hashing function and will use `eq` to determine equivalence between elements.

RWTValHashMultiSet<T,H,EQ>`(const rw_hashmultiset<T,H,EQ>& s);`

Constructs a set by copying all elements of `s`.

RWTValHashMultiSet<T,H,EQ>`(const RWTValHashMultiSet<T,H,EQ>&);`

Copy constructor.

RWTValHashMultiSet<T,H,EQ>
`(const H& h,size_type sz = RWDEFAULT_CAPACITY);`

Creates an empty hashed multi-set which uses the hash object `h` and has an initial hash table capacity of `sz`.

RWTValHashMultiSet<T,H,EQ>(const T* first,const T* last,size_type sz = 1024,const H& h = H(),const EQ& eq = EQ());

Constructs a set by copying elements from the array of Ts pointed to by first, up to, but not including, the element pointed to by last. The underlying hash table representation will have sz buckets, will use h as its hashing function and will use eq to determine equivalence between elements.

Public Member Operators

RWTValHashMultiSet<T,H,EQ>&
operator=(const RWTValHashMultiSet<T,H,EQ>& s);

RWTValHashMultiSet<T,H,EQ>&
operator=(const rw_hashmultiset<T,H,EQ>& s);

Destroys all elements of self and replaces them by copying all elements of s.

bool
operator==(const RWTValHashMultiSet<T,H,EQ>& s) const;

bool
operator==(const rw_hashmultiset<T,H,EQ>& s) const;

Returns true if self compares equal to s, otherwise returns false. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

Public Member Functions

void
apply(void (*fn)(const_reference,void*), void* d) const;

Applies the user-defined function pointed to by fn to every item in the collection. This function must have prototype:

```
void yourfun(const_reference a, void* d);
```

Client data may be passed through parameter d.

iterator
begin();

const_iterator
begin() const;

Returns an iterator positioned at the first element of self.

size_type
capacity() const;

Returns the number of buckets(slots) available in the underlying hash representation. See **resize** below.

void
clear();

Clears the collection by removing all items from self. Each item will have its destructor called.

`bool`
`contains``(const_reference a) const;`
Returns `true` if there exists an element `t` in self that compares equal to `a`, otherwise returns `false`.

`bool`
`contains``(bool (*fn)(const_reference,void*), void* d) const;`
Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

`void`
`difference``(const RWTValHashMultiSet<T,H,EQ>& s);`
Sets self to the set-theoretic difference given by `(self - s)`.

`iterator`
`end``();`

`const_iterator`
`end``() const;`
Returns an iterator positioned "just past" the last element in self.

`size_type`
`entries``() const;`
Returns the number of items in self.

`float`
`fillRatio``() const;`
Returns the ratio `entries()`/`capacity()`.

`bool`
`find``(const_reference a,T& k) const;`
If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

`bool`
`find``(bool (*fn)(const_reference,void*),void* d,T& k) const;`
If there exists an element `t` in self that compares equal to `a`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```
Adds the item `a` to the collection. Returns `true`.

```
void
intersection(const RWTValHashMultiSet<T,H,EQ>& s);
```
Destructively performs a set theoretic intersection of self and `s`, replacing the contents of self with the result.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isEquivalent(const RWTValHashMultiSet<T,H,EQ>& s) const;
```
Returns `true` if there is set equivalence between self and `s`, and returns `false` otherwise.

```
bool
isProperSubsetOf(const RWTValHashMultiSet<T,H,EQ>& s) const;
```
Returns `true` if self is a proper subset of `s`, and returns `false` otherwise.

```
bool
isSubsetOf(const RWTValHashMultiSet<T,H,EQ>& s) const;
```
Returns `true` if self is a subset of `s`, and returns `false` otherwise.

```
size_type
occurrencesOf(const_reference a) const;
```
Returns the number of elements `t` in self that compares equal to `a`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),void* d) const;
```
Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const_reference a);
```
Removes the first element `t` in self that compares equal to `a` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```
Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

size_type
removeAll(const_reference a);

Removes all elements `t` in self that compare equal to `a`. Returns the number of items removed.

size_type
removeAll(bool (*fn)(const_reference,void*), void* d);

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

void
resize(size_type sz);

Changes the capacity of self by creating a new hashed multi-set with a capacity of `sz` . **resize** copies every element of self into the new container and finally swaps the internal representation of the new container with the internal representation of `self`.

rw_hashmultiset<T,H,EQ>&
std();

const rw_hashmultiset<T,H,EQ>&
std() const;

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

void
symmetricDifference(const RWTValHashMultiSet<T,H,EQ>& s);

Destructively performs a set theoretic symmetric difference operation on self and `s`. Self is replaced by the result. A symmetric difference can be informally defined as $(A \cup B)-(A \cap B)$.

void
Union(const RWTValHashMultiSet<T,H,EQ>& rhs);

Destructively performs a set theoretic union operation on self and `rhs`. Self is replaced by the result. Note the uppercase "U" in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTValHashMultiSet<T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm,
       const RWTValHashMultiSet<T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashMultiSet<T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm, RWTValHashMultiSet<T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashMultiSet<T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm, RWTValHashMultiSet<T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvhasht.h>
RWTValHashMultiSet<T,H,EQ> m;
RWTValHashMultiSet<T,H,EQ> itr(m);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the interface for *RWTValHashTableIterator* described in Appendix A.

Description

RWTValHashMultiSetIterator is supplied with *Tools.h++* 7 to provide an iterator interface to ***RWTValHashMultiSetIterator*** that is backward compatible with the container iterators provided in *Tools.h++* 6.x.

Iteration over an ***RWTValHashMultiSet*** is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used. The only useable relationship between consecutive elements is that elements which are defined to be equivalent by the equivalence object, `EQ`, will remain adjacent.

The current item referenced by this iterator is undefined after construction or after a call to `reset().` The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Example

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

struct silly_h{
   unsigned long operator()(const RWCString& x) const
   { return x.length() * (long)x(0); }
};

int main(){
   RWTValHashMultiSet
     <RWCString, silly_h,equal_to<RWCString> > age;
   RWTValHashMultiSetIterator
     <RWCString, silly_h, equal_to<RWCString > > itr(age);
```

```
   age.insert("John");
   age.insert("Steve");
   age.insert("Mark");
   age.insert("Steve");

   for(;itr();)
     cout << itr.key() << endl;

   return 0;
}
```

Program Output (not necessarily in this order)

```
John
Steve
Mark
Steve
```

Public Constructors

RWTValHashMultiSetIterator<T,H,EQ> (RWTValHashMultiSet<T,H,EQ>&h);

Creates an iterator for the hashed multi-set `h`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

RWBoolean
operator()();

Advances `self` to the next element. Returns `false` if the iterator has advanced past the last item in the container and `true` otherwise.

RWBoolean
operator++();

Advances `self` to the next element. If the iterator has been reset or just created `self` will now reference the first element. If, before iteration, `self` referenced the last value in the multi-set, `self` will now reference an undefined value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

Public Member Functions

RWTValHashMultiSet<T,H,EQ>*
container() const;

Returns a pointer to the collection being iterated over.

T
key() const;

Returns the value currently referenced by self.

void
reset();
void
reset(RWTValHashMultiSet<T,H,EQ>& h);

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValHashMultiSet` to `reset()` will reset the iterator on that container.

Synopsis

```
#include <rw/tvhset.h>
RWTValHashSet<T,H,EQ> s;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValHashSet* described in Appendix A.

Description

This class maintains a collection of values, which are stored according to a hash object of type `H`. `H` must offer a hash function for elements of type `T` via a public member

```
unsigned long operator()(const T& x) const
```

Objects within the collection will be grouped together based on an equality object of type `EQ`. `EQ` must ensure this grouping via public member

```
bool operator()(const T& x, const T& y) const
```

which should return `true` if `x` and `y` are equivalent, `false` otherwise.

RWTValHashSet<T,H,EQ> will not accept an item that compares equal to an item already in the collection. (*RWTValHashMultiSet<T,H,EQ>* may contain multiple items that compare equal to each other.) Equality is based on the equality object and *not* on the `==` operator.

Persistence

Isomorphic

Example

```
//
// tvhsstr.cpp
//
#include <rw/tvhset.h>
#include <rw/cstring.h>
#include <iostream.h>

struct silly_hash{
   unsigned long operator()(RWCString x) const
   { return x.length() * (long)x(0); }
};

main(){
RWTValHashSet<RWCString,silly_hash,equal_to<RWCString> > set1;
RWTValHashSet<RWCString,silly_hash,equal_to<RWCString> > set2;

  set1.insert("one");
  set1.insert("two");
```

```
  set1.insert("three");

//Rejected, no duplicates allowed
  set1.insert("one");

  cout << set1.entries() << endl;  // Prints "3"

  set2.insert("one");
  set2.insert("five");

//Rejected, no duplicates allowed
  set2.insert("one");

  cout << ((set1.isEquivalent(set2)) ? "TRUE" : "FALSE") << endl;
  // Prints "FALSE"

  set2.intersection(set1);

  set1.clear();
  cout << set1.entries() << endl;    // Prints "0"
  cout << set2.entries() << endl;    // Prints "1"

  return 0;
}
```

Related Classes

Class *RWTValHashMultiSet<T,H,EQ>* offers the same interface to a collection that accepts multiple items that compare equal to each other.

Class *rw_hashset<T,H,EQ>* is the C++-standard compliant collection that serves as the underlying implementation for *RWTValHashSet<T,H,EQ>*.

Public Typedefs

```
typedef rw_hashset<T,H,EQ>                    container_type;
typedef container_type::iterator              iterator;
typedef container_type::const_iterator        const_iterator;
typedef container_type::size_type             size_type;
typedef T                                     value_type;
typedef T&                                    reference;
typedef const T&                              const_reference;
```

Public Constructors

```
RWTValHashSet<T,H,EQ>
(size_type sz = 1024,const H& h = H(),const EQ& eq= EQ());
```

Constructs an empty set. The underlying hash table representation will have `sz` buckets, will use `h` for its hashing function and will use `eq` to determine equality between elements

```
RWTValHashSet<T,H,EQ>(const rw_hashset<T,H,EQ>& s);
```

Constructs a set by copying all elements of `s`.

```
RWTValHashSet<T,H,EQ>(const RWTValHashSet<T,H,EQ>& rws);
```

Copy constructor.

RWTPtrHashSet<T,H,EQ>
`(const H& h,size_type sz = RWDEFAULT_CAPACITY);`

Creates an empty hashed set which uses the hash object `h` and has an initial hash table capacity of `sz`.

RWTValHashSet<T,H,EQ>`(const T* first,const T* last,`
`size_type sz = 1024,const H& h = H(),const EQ& eq = EQ());`

Constructs a set by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`. The underlying hash table representation will have `sz` buckets, will use `h` for its hashing function and will use `eq` to determine equality between elements

Public Member Operators

`RWTValHashSet<T,H,EQ>&`
operator=`(const RWTValHashSet<T,H,EQ>& s);`

`RWTValHashSet<T,H,EQ>&`
operator=`(const rw_hashset<T,H,EQ>& s);`

Destroys all elements of self and replaces them by copying all elements of `s`.

`bool`
operator==`(const RWTValHashSet<T,H,EQ>& s) const;`

`bool`
operator==`(const rw_hashset<T,H,EQ>& s) const;`

Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

Public Member Functions

`void`
apply`(void (*fn)(const_reference,void*), void* d) const;`

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have prototype:

```
void yourfun(const T& a, void* d);
```

Client data may be passed through parameter `d`.

`iterator`
begin`();`

`const_iterator`
begin`() const;`

Returns an iterator positioned at the first element of self.

`size_type`
capacity`() const;`

Returns the number of buckets(slots) available in the underlying hash representation. See **resize** below.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns `true` if there exists an element `t` in self that compares equal to `a`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
difference(const RWTValHashSet<T,H,EQ>& s);
void
difference(const rw_hashset<T,H,EQ>& s);
```

Sets self to the set-theoretic difference given by `(self - s)`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
float
fillRatio() const;
```

Returns the ratio `entries()`/`capacity()`.

```
bool
find(const_reference a, value_type& k) const;
```

If there exists an element `t` in self that compares equal to `a`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d,
     value_type& k) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```

Adds the item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an element with the equivalent key.

```
void
intersection(const RWTValHashSet<T,H,EQ>& rhs);
void
intersection(const rw_hashset<T,H,EQ>& rhs);
```

Destructively performs a set theoretic intersection of self and `rhs`, replacing the contents of self with the result.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isEquivalent(const RWTValHashSet<T,H,EQ>& s) const;
```

Returns `true` if there is set equivalence between self and `s`, and returns `false` otherwise.

```
bool
isProperSubsetOf(const RWTValHashSet<T,H,EQ>& s) const;
```

Returns `true` if self is a proper subset of `s`, and returns `false` otherwise.

```
bool
isSubsetOf(const RWTValHashSet<T,H,EQ>& s) const;
```

Returns `true` if self is a subset of `s` or if self is set equivalent to `s`, `false` otherwise.

```
size_type
occurrencesOf(const_reference a) const;
```

Returns the number of elements `t` in self that compare equal to `a`.

```
size_type
occurrencesOf
(bool (*fn)(const_reference,void*),void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self that compares equal to `a`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self that compare equal to `a`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type sz);
```

Changes the capacity of self by creating a new hashed set with a capacity of `sz`. **resize** copies every element of self into the new container and finally swaps the internal representation of the new container with the internal representation of `self`.

```
rw_hashset<T,H,EQ>&
std();
```

```
const rw_hashset<T,H,EQ>&
std() const;
```

Returns a reference to the underlying collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

```
void
symmetricDifference(const RWTValHashSet<T,H,EQ>& s);
void
symmetricDifference(const rw_hashset<T,H,EQ>& s);
```

Destructively performs a set theoretic symmetric difference operation on self and `s`. Self is replaced by the result. A symmetric difference can be defined as $(A \cup B)-(A \cap B)$.

```
void
Union(const RWTValHashSet<T,H,EQ>& s);
void
Union(const rw_hashsett<T,H,EQ>& s);
```

Destructively performs a set theoretic union operation on self and `s`. Self is replaced by the result. Note the use of the uppercase "U"in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTValHashSet<T,H,EQ>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValHashSet<T,H,EQ>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashSet<T,H,EQ>& coll);
RWFile&
operator>>(RWFile& strm, RWTValHashSet<T,H,EQ>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValHashSet<T,H,EQ>*& p);
RWFile&
operator>>(RWFile& strm, RWTValHashSet<T,H,EQ>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvhset.h>
RWTValHashSet<T,H,EQ> m;
RWTValHashSetIterator<T,H,EQ> itr(m);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValHashSetIterator* described in Appendix A.

Description

RWTValHashSetIterator is supplied with *Tools.h++* 7 to provide an iterator interface to ***RWTValHashSetIterator*** that is backward compatible with the container iterators provided in *Tools.h++* 6.x.

Iteration over an ***RWTValHashSet*** is pseudorandom and dependent on the capacity of the underlying hash table and the hash function being used. The only useable relationship between consecutive elements is that elements which are defined to be equivalent by the equivalence object, `EQ`, will remain adjacent.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a pre-increment or an `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Example

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

struct silly_h{
   unsigned long operator()(const RWCString& x) const
   { return x.length() * (long)x(0); }
};

int main(){
   RWTValHashSet <RWCString, silly_h,equal_to<RWCString> > age;
   RWTValHashSetIterator
     <RWCString, silly_h, equal_to<RWCString > > itr(age);

   age.insert("John");
```

```
   age.insert("Steve");
   age.insert("Mark");

//Duplicate insertion rejected
   age.insert("Steve");

   for(;itr();) cout << itr.key() << endl;

   return 0;
}
```

Program Output (not necessarily in this order)

```
John
Steve
Mark
```

Public Constructors

RWTValHashSetIterator<T,H,EQ> (RWTValHashSet<T,H,EQ>&h);

Creates an iterator for the hashset h. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

RWBoolean
operator()();

Advances self to the next element. Returns false if the iterator has advanced past the last item in the container and true otherwise.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last value in the multi-set, self will now reference an undefined value and false will be returned. Otherwise, true is returned. Note: no postincrement operator is provided.

Public Member Functions

RWTValHashSet<T,H,EQ>*
container() const;

Returns a pointer to the collection being iterated over.

T
key() const;

Returns the value currently pointed to by self.

void
reset();
void
reset(RWTValHashSet<T,H,EQ>& h);

Resets the iterator so that after being advanced it will reference the first element of the collection. Using reset() with no argument will reset the iterator on the current container. Supplying a RWTValHashSet to reset() will reset the iterator on that container.

Synopsis

```
#define RWTValHashTable RWTValHashMultiSet
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTValHashMultiSet*. Although the old name (*RWTValHashTable*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTValHashTable* in Appendix A.

Synopsis

```
#define RWTValHashTableIterator RWTValHashMultiSetIterator
```

Please Note!

If you have the Standard C++ Library, refer to the reference for this class under its new name: *RWTValHashMultiSetIterator*. Although the old name (*RWTValHashTableIterator*) is still supported, we recommend that you use the new name when coding your applications.

If you do *not* have the Standard C++ Library, refer to the description of *RWTValHashTableIterator* in Appendix A.

Synopsis

```
#include <rw/tvmap.h>
RWTValMap<K,T,C> m;
```

Standard C++ Library Dependent!

***RWTValMap* requires the Standard C++ Library.**

Description

This class maintains a collection of keys, each with an associated item of type `T`. Order is determined by the key according to a comparison object of type `C`. `C` must induce a total ordering on elements of type `K` via a public member

```
bool operator()(const K& x, const K& y) const
```

which returns `true` if `x` and its partner should precede `y` and its partner within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example.

RWTValMap<K,T,C> will not accept a key that compares equal to any key already in the collection. (***RWTValMultiMap<K,T,C>*** may contain multiple keys that compare equal to each other.) Equality is based on the comparison object and *not* on the `==` operator. Given a comparison object `comp`, keys `a` and `b` are equal if

`!comp(a,b) && !comp(b,a)`.

Persistence

Isomorphic.

Examples

In this example, a map of `RWCString`s and `RWDate`s is exercised.

```
//
// tvmbday.cpp
//
#include <rw/tvmap.h>
#include <rw/cstring.h>
#include <rw/rwdate.h>
#include <iostream.h>

main(){
  RWTValMap<RWCString, RWDate, less<RWCString> > birthdays;

  birthdays.insert("John", RWDate(12, "April",1975));
  birthdays.insert("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:
`04/12/75`

Related Classes

Class *RWTValMultiMap<K,T,C>* offers the same interface to a collection that accepts multiple keys that compare equal to each other. *RWTValSet<T,C>* maintains a collection of keys without the associated values.

Class ***map<K,T,C,allocator>*** is the C++-standard collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef map<K,T,C,allocator>                    container_type;
typedef container_type::iterator                iterator;
typedef container_type::const_iterator          const_iterator;
typedef container_type::size_type               size_type;
typedef pair <const K,T>                        value_type;
typedef pair <const K,T>&                       reference;
typedef const pair <const K,T>&                 const_reference;
```

Public Constructors

`RWTValMap<K,T,C>``(const C& comp = C());`
Constructs an empty map with comparator `comp`.

`RWTValMap<K,T,C>``(const container_type& m);`
Constructs a map by copying all elements of `m`.

`RWTValMap<K,T,C>``(const RWTValMap<K,T,C>& rwm);`
Copy constructor.

`RWTValMap<K,T,C>``(const value_type* first,`
`    const value_type* last,const C& comp = C());`
Constructs a map by copying elements from the array of `value_type` pairs pointed to by `first`, up to, but not including, the pair pointed to by `last`.

Public Member Operators

`RWTValMap<K,T,C>&`
`operator=``(const RWTValMap<K,T,C>& m);`

`RWTValMap<K,T,C>&`
`operator=``(const container_type& m);`
Destroys all elements of self and replaces them by copying all associations from `m`.

```
bool
operator<(const RWTValMap<K,T,C>& m) const;
bool
operator<(const container_type & m) const;
```

Returns `true` if self compares lexicographically less than `m`, otherwise returns `false`. Assumes that type `K` has well-defined less-than semantics (`T::operator<(const K&)` or equivalent).

```
bool
operator==(const RWTValMap<K,T,C>& m) const;
bool
operator==(const container_type & m) const;
```

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual pairs that compare equal to each other.

```
T&
operator[](const K& key);
```

Looks up `key` and returns a reference to its associated item. If the key is not in the dictionary, then it will be added with an associated item provided by the default constructor for type `T`.

Public Member Functions

```
void
apply(void (*fn)(const K&, T&, void*),void* d);
void
apply(void (*fn)(const K&, const T&, void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every association in the collection. This function must have one of the prototypes:

```
void yourfun(const K& key, T& a, void* d);
 void yourfun(const K& key, const T& a,void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K&, T&, void*),void* d);
void
applyToKeyAndValue
(void (*fn)(const K&, const T&, void*), void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply.**

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
void
clear();
```

Clears the collection by removing all items from self. Each key and its associated item will have its destructor called.

```
bool
contains(const K& key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `key`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an association a in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
bool
find(const K& key, Key& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d,
     pair<K,T>& r) const;
```

If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, assigns `a` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
findValue(const K& key, T& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns the item associated with `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
findKeyValue(const K& key, K& kr, T& tr) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `kr`, assigns the item associated with `j` to `tr`, and returns `true`. Otherwise, returns `false` and leaves the values of `kr` and `tr` unchanged.

```
bool
insert(const K& key, const T& a);
```

Adds `key` with associated item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an association with the equivalent key.

```
bool
insertKeyAndValue(const K& key, const T& a);
```

This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert**.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K& key) const;
```

Returns the number of keys `j` in self that compare equal to `key`.

```
size_type
occurrencesOf
(bool (*fn)(const_reference&,void*),void* d) const;
```

Returns the number of associations `a` in self such that the expression`((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference& a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const K& key);
```

Removes the first association with key `j` in self such that `j` compares equal to `key` and returns `true`. Returns `false` if there is no such association.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const K& key);
```

Removes all associations with key j in self such that `j` compares equal to `key`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
map<K,T,C,allocator>&
std();
const map<K,T,C,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValMap<K,T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValMap<K,T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValMap<K,T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTValMap<K,T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValMap<K,T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTValMap<K,T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvmap.h>
RWTValMap<K,T,C> vm;
RWTValMapIterator<K,T,C> itr(vm);
```

Standard C++ Library Dependent!

***RWTValMapIterator* requires the Standard C++ Library.**

Description

RWTValMapIterator is supplied with *Tools.h++* 7 to provide an iterator interface to ***RWTValMapIterator*** that is backward compatable with the container iterators provided in *Tools.h++* 6.x.

The order of iteration over an ***RWTValMap*** is dependent on the comparator object supplied as applied to the key values of the stored associations.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

xamples

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

int main(){
   RWTValMap<RWCString,int,greater<RWCString> > age;
   RWTValMapIterator<RWCString,int,greater<RWCString> > itr(age);

   age.insert("John", 30);
   age.insert("Steve",17);
   age.insert("Mark",24);

//Insertion is rejected, no duplicates allowed
   age.insert("Steve",24);

   for(;itr();)
     cout << itr.key() << "\'s age is " << itr.value() << endl;

   return 0;
}
```

Program Output
```
Steve's age is 17
Mark's age is 24
John's age is 30
```

Public Constructors

`RWTValMapIterator<K,T,C>`
`(RWTValMap<K,T,C>&h);`

Creates an iterator for the map `h`. The iterator begins in an undefined state and must be advanced before the first association will be accessible.

Public Member Operators

`RWBoolean`
`operator()``();`

Advances `self` to the next element. If the iterator has advanced past the last element in the collection, `false` will be returned. Otherwise, `true` will be returned.

`RWBoolean`
`operator++``();`

Advances `self` to the next element. If the iterator has been reset or just created `self` will now reference the first element. If, before iteration, `self` pointed to the last association in the map, `self` will now reference an undefined value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

Public Member Functions

`RWTValMap<K,T,C>*`
`container()` `const;`

Returns a pointer to the collection being iterated over.

`K`
`key``() const;`

Returns the key portion of the association currently referenced by `self`.

`void`
`reset``();`
`void`
`reset``(RWTValMap<K,T,C>& h);`

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValMap` with `reset()` will reset the iterator on that container.

`T`
`value``();`

Returns the value portion of the association referenced by `self`.

Synopsis

```
#include <rw/tvmmap.h>
RWTValMultiMap<K,T,C> m;
```

Standard C++ Library Dependent!

RWTValMultiMap **requires the Standard C++ Library.**

Description

This class maintains a collection of keys, each with an associated item of type `T`. Order is determined by the key according to a comparison object of type `C`. `C` must induce a total ordering on elements of type `K` via a public member

`bool operator()(const K& x, const K& y) const`

which returns `true` if `x` and its partner should precede `y` and its partner within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example.

RWTValMultiMap<K,T,C> may contain multiple keys that compare equal to each other. (***RWTValMap<K,T,C>*** will not accept a key that compares equal to any key already in the collection.) Equality is based on the comparison object and *not* on the `==` operator. Given a comparison object `comp`, keys `a` and `b` are equal if

`!comp(a,b) && !comp(b,a)`.

Persistence

Isomorphic.

Examples

In this example, a map of ***RWCString***s and ***RWDate***s is exercised.

```
//
// tvmmbday.cpp
//
#include <rw/tvmmap.h>
#include <rw/cstring.h>
#include <rw/rwdate.h>
#include <iostream.h>
#include <function.h>

main(){
  typedef RWTValMultiMap<RWCString, RWDate, less<RWCString> >
     RWMMap;
  RWMMap birthdays;

  birthdays.insert("John", RWDate(12, "April",1975));
  birthdays.insert("Ivan", RWDate(2, "Nov", 1980));
  birthdays.insert("Mary", RWDate(22, "Oct", 1987));
  birthdays.insert("Ivan", RWDate(19, "June", 1971));
```

```
  birthdays.insert("Sally",RWDate(15, "March",1976));
  birthdays.insert("Ivan",  RWDate(6, "July", 1950));

  // How many "Ivan"s?
  RWMMap::size_type n = birthdays.occurrencesOf("Ivan");
  RWMMap::size_type idx = 0;
  cout << "There are " << n << " Ivans:" << endl;
  RWMMap::iterator iter = birthdays.std().lower_bound("Ivan");
  while (++idx <= n)
    cout << idx << ".  " << (*iter++).second << endl;
  return 0;
}
```

Program Output:

```
There are 3 Ivans:
1.  11/02/80
2.  06/19/71
3.  07/06/50
```

Related Classes

Class *RWTValMap<K,T,C>* offers the same interface to a collection that will not accept multiple keys that compare equal to each other. *RWTValMultiSet<T,C>* maintains a collection of keys without the associated values.

Class ***multimap<K,T,C,allocator>*** is the C++-standard collection that serves as the underlying implementation for this collection.

Public Typedefs

```
typedef multimap<K,T,C,allocator>             container_type;
typedef container_type::iterator              iterator;
typedef container_type::const_iterator        const_iterator;
typedef container_type::size_type             size_type;
typedef pair <const K,T>                       value_type;
typedef pair <const K,T>&                      reference;
typedef const pair <const K,T>&                const_reference;
```

Public Constructors

`RWTValMultiMap<K,T,C>``(const C& comp = C());`

Constructs an empty map with comparator `comp`.

`RWTValMultiMap<K,T,C>``(const container_type& m);`

Constructs a map by copying all elements of `m`.

`RWTValMultiMap<K,T,C>``(const RWTValMultiMap<K,T,C>& rwm);`

Copy constructor.

`RWTValMultiMap<K,T,C>`
`(const value_type* first, const value_type* last,`
` const C& comp = C());`

Constructs a map by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

```
RWTValMultiMap<K,T,C>&
operator=(const RWTValMultiMap<K,T,C>& m);
RWTValMultiMap<K,T,C>&
operator=(const container_type& m) const;
```

Destroys all elements of self and replaces them by copying all associations from `m`.

```
bool
operator<(const RWTValMultiMap<K,T,C>& m);
bool
operator<(const container_type& m) const;
```

Returns `true` if self compares lexicographically less than `m`, otherwise returns `false`. Assumes that type `K` has well-defined less-than semantics (`T::operator<(const K&)` or equivalent).

```
bool
operator==(const RWTValMultiMap<K,T,C>& m) const;
bool
operator==(const container_type& m) const;
```

Returns `true` if self compares equal to `m`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual pairs that compare equal to each other.

Public Member Functions

```
void
apply(void (*fn)(const K&, T&, void*),void* d);
void
apply(void (*fn)(const K&, const T&, void*),void* d) const;
```

Applies the user-defined function pointed to by `fn` to every association in the collection. This function must have one of the prototypes:

```
void yourfun(const K& key, T& a, void* d);
void yourfun(const K& key, const T& a,void* d);
```

Client data may be passed through parameter `d`.

```
void
applyToKeyAndValue(void (*fn)(const K&, T&, void*),void* d);
void
applyToKeyAndValue
(void (*fn)(const K&, const T&, void*),void* d) const;
```

This is a deprecated version of the **apply** member above. It behaves exactly the same as **apply.**

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first pair in self.

```
void
clear();
```

Clears the collection by removing all items from self. Each key and its associated item will have its destructor called.

```
bool
contains(const K& key) const;
```

Returns `true` if there exists a key `j` in self that compares equal to `key`, otherwise returns `false`.

```
bool
contains
(bool (*fn)(const_reference,void*),void* d) const;
```

Returns `true` if there exists an association a in self such that the expression `((*fn)(a,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last association in self.

```
size_type
entries() const;
```

Returns the number of associations in self.

```
bool
find(const K& key, Key& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
find(bool (*fn)(const_reference,void*),void* d,
     pair<K,T>& r) const;
```

If there exists an association `a` in self such that the expression `((*fn)(a,d))` is `true`, assigns `a` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
findValue(const K& key, T& r) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns the item associated with `j` to `r` and returns `true`. Otherwise, returns `false` and leaves the value of `r` unchanged.

```
bool
findKeyValue(const K& key, K& kr, T& tr) const;
```

If there exists a key `j` in self that compares equal to `key`, assigns `j` to `kr`, assigns the item associated with `j` to `tr`, and returns `true`. Otherwise, returns `false` and leaves the values of `kr` and `tr` unchanged.

```
bool
insert(const K& key, const T& a);
```

Adds `key` with associated item `a` to the collection. Returns `true`.

```
bool
insertKeyAndValue(const K& key, const T& a);
```

This is a deprecated version of the **insert** member above. It behaves exactly the same as **insert.**

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
size_type
occurrencesOf(const K& key) const;
```

Returns the number of keys `j` in self that compare equal to `key`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),
              void* d) const;
```

Returns the number of associations `a` in self such that the expression`((*fn)(a,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const K& key);
```

Removes the first association with key `j` in self where j compares equal to `key` and returns `true`. Returns `false` if there is no such association.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first association `a` in self such that the expression `((*fn)(a,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const K& key);
```

Removes all associations in self that have a key `j` that compares equal to `key`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all associations `a` in self such that the expression `((*fn)(a,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
multimap<K,T,C,allocator>&
std();
const multimap<K,T,C,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
      const RWTValMultiMap<K,T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValMultiMap<K,T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValMultiMap<K,T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTValMultiMap<K,T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValMultiMap<K,T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTValMultiMap<K,T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvmmap.h>
RWTValMultiMap<K,T,C> vm;
RWTValMultiMapIterator<K,T,C> itr(vm);
```

Standard C++ Library Dependent!

RWTValMultiMapIterator **requires the Standard C++ Library.**

Description

RWTValMultiMapIterator is supplied with *Tools.h++* 7 to provide an iterator interface for class ***RWTValMultiMap*** that has backward compatibility with the container iterators provided in *Tools.h++* 6.x.

The order of iteration for an ***RWTValMultiMap*** is dependent upon the comparator object as applied to the keys of the stored associations.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTValMultiMap<RWCString,int,greater<RWCString> > a;
   RWTValMultiMapIterator
      <RWCString,int,greater<RWCString> > itr(a);

   a.insert("John", 30);
   a.insert("Steve",17);
   a.insert("Mark",24);
   a.insert("Steve",24);

   for(;itr();)
     cout << itr.key() << "\'s age is " << itr.value() << endl;

   return 0;
}
```

Program Output
```
Steve's age is 17
Steve's age is 24
Mark's age is 24
John's age is 30
```

Public Constructors

`RWTValMultiMapIterator<K,T,C>`
`(RWTValMultiMap<K,T,C>&m);`

Creates an iterator for the multi-map `m`. The iterator begins in an undefined state and must be advanced before the first association will be accessible.

Public Member Operators

`RWBoolean`
`operator()``();`

Advances self to the next element. If the iterator has advanced past the last item in the collection, returns `false`. Otherwise, returns `true`.

`RWBoolean`
`operator++``();`

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multi-map, self will now reference an undefined value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operation is provided.

Public Member Functions

`RWTValMultiMap<K,T,C>*`
`container()` `const;`

Returns a pointer to the collection being iterated over.

`K`
`key``() const;`

Returns the key portion of the association currently referenced by `self`.

`void`
`reset``();`
`void`
`reset``(RWTValMultiMap<K,T,C>& h);`

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValMultiMap` to `reset()` will reset the iterator on the new container.

`T`
`value``();`

Returns the value portion of the association referenced by `self`.

Synopsis

```
#include <rw/tvmset.h>
RWTValMultiSet<T,C>
```

Standard C++ Library Dependent!

***RWTPtrMultiSet* requires the Standard C++ Library.**

Description

This class maintains a collection of values, which are ordered according to a comparison object of type `C`. `C` must induce a total ordering on elements of type T via a public member

```
bool operator()(const T& x, const T& y) const
```

which returns `true` if `x` should precede `y` within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example.

RWTValMultiSet<T,C> may contain multiple items that compare equal to each other. (***RWTValSet<T,C>*** will not accept an item that compares equal to an item already in the collection.)

Persistence

Isomorphic.

Examples

In this example, a multi-set of `RWCString`s is exercised.

```
//
// tvmsstr.cpp
//
#include <rw/tvmset.h>
#include <rw/cstring.h>
#include <iostream.h>

main(){
 RWTValMultiSet<RWCString,less<RWCString> > set;

  set.insert("one");
  set.insert("two");
  set.insert("three");
  set.insert("one");              // OK, duplicates allowed

  cout << set.entries() << endl;   // Prints "4"
  return 0;
}
```

Related Classes

Class ***RWTValSet<T,C>*** offers the same interface to a collection that will not accept multiple items that compare equal to each other.

RWTValMultiMap<K,T,C> maintains a collection of key-value pairs.

Class ***multiset<T,C,allocator>*** is the C++-standard collection that serves as the underlying implementation for *RW**TValMultiSet<T,C>***.

Public Typedefs

```
typedef multiset<T,C,allocator>                     container_type;
typedef container_type::iterator                    iterator;
typedef container_type::const_iterator              const_iterator;
typedef container_type::size_type                   size_type;
typedef T                                           value_type;
typedef const T&                                    const_reference;
```

Public Constructors

`RWTValMultiSet<T,C>``(const C& cmp = C());`

Constructs an empty set.

`RWTValMultiSet<T,C>``(const container_type& s);`

Constructs a set by copying all elements of `s`.

`RWTValMultiSet<T,C>``(const RWTValMultiSet<T,C>& rws);`

Copy constructor.

`RWTValMultiSet<T,C>`
`(const T* first,const T* last,const C& cmp = C());`

Constructs a set by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTValMultiSet<T,C>&`
`operator=``(const RWTValMultiSet<T,C>& s);`

`RWTValMultiSet<T,C>&`
`operator=``(const container_type& s);`

Destroys all elements of self and replaces them by copying all elements of `s`.

`bool`
`operator<``(const RWTValMultiSet<T,C>& s) const;`

`bool`
`operator<``(const container_type& s) const;`

Returns `true` if self compares lexicographically less than `s`, otherwise returns `false`. Assumes that type `T` has well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

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

`bool`
`operator==``(const container_type& s) const;`

Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

Public Member Functions

```
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by fn to every item in the collection. This function must have prototype:

```
void yourfun(const_reference a, void* d);
```

Client data may be passed through parameter d.

```
iterator
begin();
```

```
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns true if there exists an element t in self that compares equal to a, otherwise returns false.

```
bool
contains(bool (*fn)(const_reference, void*), void* d) const;
```

Returns true if there exists an element t in self such that the expression ((*fn)(t,d)) is true, otherwise returns false. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter d.

```
void
difference(const RWTValMultiSet<T,C>& s);
```

```
void
difference(const container_type& s);
```

Sets self to the set-theoretic difference given by (self - s).

```
iterator
end();
```

```
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
bool
find(const_reference a, T& k) const;
```

If there exists an element t in self that compares equal to a, assigns t to k and returns true. Otherwise, returns false and leaves the value of k unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d, T& k) const;
```

If there exists an element t in self such that the expression ((*fn)(t,d)) is true, assigns t to k and returns true. Otherwise, returns false and leaves the value of k unchanged. fn points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter d.

```
void
intersection(const RWTValMultiSet<T,C>& s);
void
intersection(const container_type& s);
```

Sets self to the intersection of self and s.

```
bool
insert(const_reference a);
```

Adds the item a to the collection. Returns true.

```
bool
isEmpty() const;
```

Returns true if there are no items in the collection, false otherwise.

```
bool
isEquivalent(const RWTValMultiSet<T,C>& s) const;
```

Returns true if there is set equivalence between self and s, and returns false otherwise.

```
bool
isProperSubsetOf(const RWTValMultiSet<T,C>& s) const;
```

Returns true if self is a proper subset of s, and returns false otherwise.

```
bool
isSubsetOf(const RWTValMultiSet<T,C>& s) const;
```

Returns true if self is a subset of s or if self is set equivalent to rhs, false otherwise.

```
size_type
occurrencesOf(const_reference a) const;
```

Returns the number of elements t in self that compare equal to a.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self that compares equal to `a` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self that compare equal to `a`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
multiset<T,C,allocator>&
std();
```

```
const multiset<T,C,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

void
symmetricDifference(const RWTValMultiSet<T,C>& s);

void
symmetricDifference(const container_type& s);

Sets self to the symmetric difference of self and `s`.

void
Union(const RWTValMultiSet<T,C>& s);

void
Union(const container_type& s);

Sets self to the union of self and `s`. Note the use of the uppercase "U"in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

RWvostream&
operator<<(RWvostream& strm, const RWTValMultiSet<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValMultiSet<T,C>& coll);

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

RWvistream&
operator>>(RWvistream& strm, RWTValMultiSet<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTValMultiSet<T,C>& coll);

Restores the contents of the collection `coll` from the input stream `strm`.

RWvistream&
operator>>(RWvistream& strm, RWTValMultiSet<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTValMultiSet<T,C>*& p);

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvmset.h>
RWTValMultiSet< T,C> vs;
RWTValMultiSetIterator< T,C> itr(vs);
```

Standard C++ Library Dependent!

RWTValMultiSetIterator **requires the Standard C++ Library.**

Description

RWTValMultiSetIterator is supplied with *Tools.h++* 7 to provide an iterator interface for class ***RWTValMultiSetIterator*** that has backward compatibility with the container iterators provided in *Tools.h++* 6.x.

The order of iteration over an ***RWTValMultiSet*** is dependent on the supplied comparator object parameter C as applied to the values stored in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTValMultiSet<RWCString,greater<RWCString> > a;
   RWTValMultiSetIterator<RWCString,greater<RWCString> > itr(a);
   a.insert("John");
   a.insert("Steve");
   a.insert("Mark");
   a.insert("Steve");

   for(;itr();)
     cout << itr.key() << endl;

   return 0;
}
```

Program Output

```
Steve
Steve
Mark
John
```

Public Constructors

RWTValMultiSetIterator<T,C>(RWTValMultiSet< T,C> &h);

Creates an iterator for the multi-set h. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

RWBoolean
operator()();

Advances self to the next element. If the iterator has advanced past the last element in the collection, false will be returned. Otherwise, true will be returned.

RWBoolean
operator++();

Advances self to the next element. If the iterator has been reset or just created self will now reference the first element. If, before iteration, self referenced the last association in the multi-set, self will now reference an undefined value and false will be returned. Otherwise, true is returned. Note: no postincrement operator is provided.

Public Member Functions

RWTValMultiSet<T,C>*
container() const;

Returns a pointer to the collection being iterated over.

T
key();

Returns the value pointed to by self.

void
reset();
void
reset(RWTValMultiSet<T,C>& h);

Resets the iterator so that after being advanced it will point to the first element of the collection. Using reset() with no argument will reset the iterator on the current container. Supplying a RWTValMultiSet to reset() will reset the iterator on that container.

Synopsis

```
#include <rw/tvordvec.h>
RWTValOrderedVector<T> ordvec;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValOrderedVector* described in Appendix A.

Description

This class maintains a collection of values, implemented as a vector.

Persistence

Isomorphic

Example

In this example, a vector of type `double` is exercised.

```
//
// tvordvec.cpp
//
#include <rw/tvordvec.h>
#include <iostream.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
```

Related Classes

Classes *RWTValDeque<T>*, *RWTValSlist<T>*, and *RWTValDlist<T>* also provide a Rogue Wave interface to C++-standard sequence collections.

Class ***vector<T,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef vector<T,allocator>                      container_type;
typedef container_type::iterator                 iterator;
typedef container_type::const_iterator           const_iterator;
typedef container_type::size_type                size_type;
typedef T                                        value_type;
typedef T&                                       reference;
typedef const T&                                 const_reference;
```

Public Constructors

`RWTValOrderedVector<T>``();`
Constructs an empty vector.

`RWTValOrderedVector<T>``(const vector<T,allocator>& vec);`
Constructs a vector by copying all elements of `vec`.

`RWTValOrderedVector<T>``(const RWTValOrderedVector<T>& rwvec);`
Copy constructor.

`RWTValOrderedVector<T>``(size_type n, const T& val);`
Constructs a vector with `n` elements, each initialized to `val`.

`RWTValOrderedVector<T>``(size_type n);`
Constructs an empty vector with a capacity of `n` elements.

`RWTValOrderedVector<T>``(const T* first, const T* last);`
Constructs a vector by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTValOrderedVector<T>&`
`operator=``(const RWTValOrderedVector<T>& vec);`
`RWTValOrderedVector<T>&`
`operator=``(const vector<T,allocator>& vec);`
Calls the destructor on all elements of self and replaces them by copying all elements of `vec`.

`bool`
`operator<``(const RWTValOrderedVector<T>& vec);`
`bool`
`operator<``(const vector<T>& vec);`
Returns `true` if self compares lexicographically less than `vec`, otherwise returns false. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

`bool`
`operator==``(const RWTValOrderedVector<T>& vec) const;`
`bool`
`operator==``(const vector<T>& vec) const;`
Returns `true` if self compares equal to `vec`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating

through both collections produces, in turn, individual elements that compare equal to each other.

```
T&
operator()(size_type i);
const T&
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
T&
operator[](size_type i);
const T&
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(const_reference a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(reference,void*), void* d);
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(const_reference a, void* d);
void yourfun(reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```
Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```
Returns `true` if there exists an element `t` in self such that the expression`(t == a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```
Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is true, otherwise returns false. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
const T*
data() const;
```
Returns a pointer to the first element of the vector.

```
iterator
end();
const_iterator
end() const;
```
Returns a *past-the-end* valued iterator of self.

```
size_type
entries() const;
```
Returns the number of elements in self.

```
bool
find(const_reference a, value_type& k) const;
```
If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d,
     value_type& k) const;
```
If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T& a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
const_reference
first() const;
```

Returns a reference to the first element of self.

```
size_type
index(const_reference a) const;
```

Returns the position of the first item `t` in self such that `(t == a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const_reference,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```

Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, const_reference a);
```

Inserts the item `a` in front of the item at position `i` in self. This position must be between 0 and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
reference
last();
const_reference
last() const;
```

Returns a reference to the last item in the collection.

```
size_type
length() const;
```

Returns the number of elements in self.

```
reference
maxElement();
const_reference
maxElement() const;
reference
minElement();
const_reference
minElement() const;
```

Returns a reference to the minimum or maximum element in the collection. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

```
size_type
occurrencesOf(const_reference a) const;
```

Returns the number of elements `t` in self such that the expression `(t == a)` is `true`.

```
size_type
occurrencesOf
(bool (*fn)(const_reference,void*), void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
prepend(const_reference a);
```

Adds the item `a` to the beginning of the collection.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self such that the expression `(t == a)` is `true` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self such that the expression `(t == a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
value_type
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between 0 and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
value_type
removeFirst();
```

Removes and returns the first item in the collection.

```
value_type
removeLast();
```

Removes and returns the first item in the collection.

```
size_type
replaceAll(const_reference oldVal, const_reference newVal);
```

Replaces all elements `t` in self such that the expression `(t == oldVal)` is `true` with `newVal`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(const_reference,void*),
           void* d, const T& newval);
```

Replaces all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
resize(size_type n);
```

Modify the capacity of the vector to be at least as large as `n`. The function has no effect if the capacity is already as large as `n`.

```
void
sort();
```

Sorts the collection using the less-than operator to compare elements.

```
vector<T,allocator>&
std();
const vector<T,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTValOrderedVector<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValOrderedVector<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValOrderedVector<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValOrderedVector<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValOrderedVector<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValOrderedVector<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tvset.h>
RWTValSet<T,C> s;
```

Standard C++ Library Dependent!

***RWTValSet* requires the Standard C++ Library.**

Description

This class maintains a collection of values, which are ordered according to a comparison object of type `C`. `C` must induce a total ordering on elements of type `T` via a public member

`bool operator()(const T& x, const T& y) const`

which returns `true` if `x` should precede `y` within the collection. The structure `less<T>` from the C++-standard header file `<functional>` is an example.

RWTValSet<T,C> will not accept an item that compares equal to an item already in the collection. (*RWTValMultiSet<T,C>* may contain multiple items that compare equal to each other.) Equality is based on the comparison object and *not* on the `==` operator. Given a comparison object `comp`, items `a` and `b` are equal if

`!comp(a,b) && !comp(b,a)`.

Persistence

Isomorphic.

Examples

In this example, a set of `RWCString`s is exercised.

```
//
// tvsstr.cpp
//
#include <rw/tvset.h>
#include <rw/cstring.h>
#include <iostream.h>
#include <function.h>

main(){
  RWTValSet<RWCString,less<RWCString> > set;

  set.insert("one");
  set.insert("two");
  set.insert("three");
  set.insert("one");     // Rejected: already in collection

  cout << set.entries() << endl;     // Prints "3"
  return 0;
}
```

Related Classes

Class *RWTValMultiSet<T,C>* offers the same interface to a collection that accepts multiple items that compare equal to each other.
RWTValMap<K,T,C> maintains a collection of key-value pairs.

Class ***set<T,C,allocator>*** is the C++-standard collection that serves as the underlying implementation for *RWTValSet<T,C>*.

Public Typedefs

```
typedef set<T,C,allocator>                  container_type;
typedef container_type::iterator            iterator;
typedef container_type::const_iterator      const_iterator;
typedef container_type::size_type           size_type;
typedef T                                   value_type;
typedef const T&                            const_reference;
```

Public Constructors

`RWTValSet<T,C>``(const C& comp = C());`
Constructs an empty set.

`RWTValSet<T,C>``(const container_type& s);`
Constructs a set by copying all elements of `s`.

`RWTValSet<T,C>``(const RWTValSet<T,C>& rws);`
Copy constructor.

`RWTValSet<T,C>`
`(const T* first,const T* last,const C& comp = C());`
Constructs a set by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTValSet<T,C>&`
`operator=``(const RWTValSet<T,C>& s);`
`RWTValSet<T,C>&`
`operator=``(const container_type& s);`
Destroys all elements of self and replaces them by copying all elements of `s.`

`bool`
`operator<``(const RWTValSet<T,C>& s) const;`
`bool`
`operator<``(const container_type& s) const;`
Returns `true` if self compares lexicographically less than `s`, otherwise returns `false`. Assumes that type `T` has well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

`bool`
`operator==``(const RWTValSet<T,C>& s) const;`
`bool`
`operator==``(const set<T,C>& s) const;`
Returns `true` if self compares equal to `s`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

Public Member Functions

```
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have prototype:

```
void yourfun(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns `true` if there exists an element `t` in self that compares equal to `a`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
difference(const RWTValSet<T,C>& s);
void
difference(const container_type& s);
```

Sets self to the set-theoretic difference given by `(self - s)`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
bool
find(const_reference a, T& k) const;
```
If there exists an element `t` in self that compares equal to `a`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d, T& k) const;
```
If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```
Adds the item `a` to the collection. Returns `true` if the insertion is successful, otherwise returns `false`. The function will return `true` unless the collection already holds an element with the equivalent key.

```
void
intersection(const RWTValSet<T,C>& s);
void
intersection(const container_type& s);
```
Sets self to the intersection of self and `s`.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isEquivalent(const RWTValSet<T,C>& s) const;
```
Returns `true` if there is set equivalence between self and `s`, and returns `false` otherwise.

```
bool
isProperSubsetOf(const RWTValSet<T,C>& s) const;
```
Returns `true` if self is a proper subset of `s`, and returns `false` otherwise.

```
bool
isSubsetOf(const RWTValSet<T,C>& s) const;
```
Returns `true` if self is a subset of `s`; `false` otherwise.

```
size_type
occurrencesOf(const_reference a) const;
```
Returns the number of elements `t` in self that compare equal to `a`.

```
size_type
occurrencesOf(bool (*fn)(const T&,void*),void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self that compares equal to `a` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self that compare equal to `a`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
set<T,C,allocator>&
std();
const set<T,C,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

```
void
symmetricDifference(const RWTValSet<T,C>& s);
void
symmetricDifference(const container_type& s);
```

Sets self to the symmetric difference of self and `s`.

```
void
Union(const RWTValSet<T,C>& s);
void
Union(const container_type& s);
```

Sets self to the union of self and `s`. Note the use of the uppercase "U"in `Union` to avoid conflict with the C++ reserved word.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValSet<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValSet<T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSet<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTValSet<T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSet<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTValSet<T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvset.h>
RWTValSet<T,C> vs;
RWTValSetIterator<T,C> itr(vs);
```

Standard C++ Library Dependent!

RWTValSetIterator **requires the Standard C++ Library.**

Description

RWTValSetIterator is supplied with *Tools.h++* 7 to provide an iterator interface for class ***RWTValSetIterator*** that is backward compatable with the container iterators provided in *Tools.h++* 6.x.

The order of iteration over an ***RWTValSet*** is dependent on the supplied comparator object parameter C as applied to the values stored in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTValSet<RWCString,greater<RWCString> > a;
   RWTValSetIterator<RWCString,greater<RWCString> > itr(a);

   a.insert("John");
   a.insert("Steve");
   a.insert("Mark");

//Rejected, duplicates are not allowed
   a.insert("Steve");

   for(;itr();)
     cout << itr.key() << endl;

   return 0;
}
```

Program Output

```
Steve
Mark
John
```

Public Constructors

```
RWTValSetIterator<T,C>(RWTValSet<T,C>&s);
```

Creates an iterator for the set `s`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

```
RWBoolean
operator()();
```

Advances `self` to the next element. If the iterator has advanced past the last element in the collection, `false` will be returned. Otherwise, `true` will be returned.

```
RWBoolean
operator++();
```

Advances `self` to the next element. If the iterator has been reset or just created `self` will now reference the first element. If, before iteration, `self` referenced the last association in the set, `self` will now reference an undefined value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

Public Member Functions

```
RWTValSet<T,C>*
container() const;
```

Returns a pointer to the collection being iterated over.

```
T
key() const;
```

Returns the value referenced by `self`.

```
void
reset();
void
reset(RWTValSet<T,C>& s);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValSet` to `reset()` will reset the iterator on that container.

Synopsis

```
#include <rw/tvslist.h>
RWTValSlist<T> lst;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValSlist* described in Appendix A.

Description

This class maintains a collection of values, implemented as a singly-linked list.

Persistence

Isomorphic

Example

In this example, a singly-linked list of `RWDate`s is exercised.

```
//
// tvslint.cpp
//
#include<rw/tvslist.h>
#include<iostream.h>

void div5(int& x, void *y){x = x/5;}

int main()
{
  const int vec[10] = {45,10,5,15,25,30,35,20,40,50};

  RWTValSlist<int> lst(vec, vec+10);
  RWTValSlistIterator<int> itr(lst);

  lst.apply(div5, 0);
  lst.sort();

  for(;itr();)
     cout << itr.key() << "  ";
  cout << endl;

  return 0;
}
```

Program Output:

```
1 2 3 4 5 6 7 8 9 10
```

Related Classes

Classes *RWTValDeque<T>*, *RWTValDlist<T>*, and *RWTValOrderedVector<T>* also provide a Rogue Wave interface to C++-standard sequence collections.

The Rogue Wave supplied, standard-compliant class ***rw_slist<T>*** is the collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef rw_slist<T>                          container_type;
typedef container_type::iterator             iterator;
typedef container_type::const_iterator       const_iterator;
typedef container_type::size_type            size_type;
typedef T                                    value_type;
typedef T&                                   reference;
typedef const T&                             const_reference;
```

Public Constructors

`RWTValSlist<T>``();`
Constructs an empty, singly-linked list.

`RWTValSlist<T>``(const rw_slist<T>& lst);`
Constructs a singly-linked list by copying all elements of `lst`.

`RWTValSlist<T>``(const RWTValSlist<T>& rwlst);`
Copy constructor.

`RWTValSlist<T>``(size_type n, const T& val = T());`
Constructs a singly-linked list with `n` elements, each initialized to `val`.

`RWTValSlist<T>``(const T* first, const T* last);`
Constructs a singly-linked list by copying elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`RWTValSlist<T>&`
`operator=``(const RWTValSlist<T>& lst);`
`RWTValSlist<T>&`
`operator=``(const rw_slist<T>& lst);`
Calls the destructor on all elements of self and replaces them by copying all elements of `lst`.

`bool`
`operator<``(const RWTValSlist<T>& lst) const;`
`bool`
`operator<``(const rw_slist<T>& lst) const;`
Returns `true` if self compares lexicographically less than `lst`, otherwise returns false. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

`bool`
`operator==``(const RWTValSlist<T>& lst) const;`
`bool`
`operator==``(const rw_slist<T>& lst) const;`
Returns `true` if self compares equal to `lst`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed.*

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
append(const_reference a);
```

Adds the item `a` to the end of the collection.

```
void
apply(void (*fn)(reference,void*), void* d);
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have one of the prototypes:

```
void yourfun(const_reference a, void* d);
void yourfun(reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

`bool`
`contains``(const T& a) const;`

Returns `true` if there exists an element `t` in self such that the expression`(t == a)` is `true`, otherwise returns `false`.

`bool`
`contains``(bool (*fn)(const T&,void*), void* d) const;`

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T& a, void* d);
```

Client data may be passed through parameter `d`.

`iterator`
`end``();`
`const_iterator`
`end``() const;`

Returns a *past-the-end* valued iterator of self.

`size_type`
`entries``() const;`

Returns the number of elements in self.

`bool`
`find``(const_reference a,reference k) const;`

If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

`bool`
`find`
`(bool (*fn)(const_reference,void*),void* d,reference k) const;`

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

`reference`
`first``();`
`const_reference`
`first``() const;`

Returns a reference to the first element of self.

```
T*
get();
```
Removes and returns the first element in the collection. This method is identical to `removeFirst` and is included to provide compatibility with previous versions.

```
size_type
index(const_reference a) const;
```
Returns the position of the first item `t` in self such that `(t == a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const_reference,void*), void* d) const;
```
Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
insert(const_reference a);
```
Adds the item `a` to the end of the collection. Returns `true`.

```
void
insertAt(size_type i, const T& a);
```
Inserts the item `a` in front of the item at position `i` in self. This position must be between 0 and the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
bool
isEmpty() const;
```
Returns `true` if there are no items in the collection, `false` otherwise.

```
T
last() const;
```
Returns a reference to the last item in the collection.

```
reference
maxElement();
const_reference
maxElement() const;
reference
minElement();
const_reference
minElement() const;
```
Returns a reference to the minimum or maximum element in the collection. Type `T` must have well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

```
size_type
occurrencesOf(const_reference a) const;
```

Returns the number of elements `t` in self such that the expression `(t == a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
prepend(const_reference a);
```

Adds the item `a` to the beginning of the collection.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self such that the expression `(t == a)` is `true` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self such that the expression `(t == a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
T
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between 0 and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T
removeFirst();
```

Removes and returns the first item in the collection.

```
T
removeLast();
```

Removes and returns the first item in the collection.

```
size_type
replaceAll(const_reference oldVal,const_reference newVal);
```

Replaces all elements `t` in self such that the expression `(t == oldVal)` is `true` with `newVal`. Returns the number of items replaced.

```
size_type
replaceAll(bool (*fn)(const_reference,void*),
           void* d,const_reference nv);
```

Replaces all elements `t` in self such that the expression `((*fn)(t,d))`is `true` with the value `nv`. Returns the number of items replaced. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
void
sort();
```

Sorts the collection using the less-than operator to compare elements.

```
rw_slist<T>&
std();
```

```
const rw_slist<T>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. This reference may be used freely, providing access to the C++-standard interface as well as interoperability with other software components that make use of the C++-standard collections.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValSlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValSlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValSlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValSlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvslist.h>
RWTValSlist<T> dl;
RWTValSlistIterator<T> itr(dl);
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValSlistIterator* described in Appendix A.

Description

RWTValSlistIterator is supplied with *Tools.h++* 7 to provide an iterator interface for class *RWTValSlistIterator* that is backward compatible with the container iterators provided in *Tools.h++* 6.x.

The order of iteration over an *RWTValSlist* is dependent on the order of insertion of the values into the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equal to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTValSlist<RWCString> a;
   RWTValSlistIterator<RWCString> itr(a);

   a.insert("John");
   a.insert("Steve");
   a.insert("Mark");
   a.insert("Steve");

   for(;itr();)
     cout << itr.key() << endl;

   return 0;
}
```

Program Output
```
John
Steve
Mark
Steve
```

Public Constructors

RWTValSlistIterator<T>(RWTValSlist<T>& s);

Creates an iterator for the singly linked list `s`. The iterator begins in an undefined state and must be advanced before the first element will be accessible

Public Member Operators

RWBoolean
operator()();

Advances `self` to the next element. If the iterator has advanced past the last element in the collection, `false` will be returned. Otherwise, `true` will be returned.

RWBoolean
operator++();

Advances `self` to the next element. If the iterator has been reset or just created, `self` will reference the first element. If, before iteration, `self` referenced the last value in the list, `self` will now reference an undefined value distinct from the reset value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

RWBoolean
operator+=(size_type n);

Behaves as if the `operator++` member function had been applied `n` times

RWBoolean
operator--();

Moves `self` back to the immediately previous element. If the iterator has been reset or just created, this operator will return `false`, otherwise it will return `true`. If `self` references the the first element, it will now be in the reset state. If `self` has been iterated past the last value in the list, it will now reference the last item in the list. Note: no postdecrement operator is provided.

RWBoolean
operator-=(size_type n);

Behaves as if the `operator--` member function had been applied `n` times

Public Member Functions

RWTValSlist<T>*
container() const;

Returns a pointer to the collection being iterated over.

`RWBoolean`
`findNext``(const_reference a);`

Advances self to the first element `t` encountered by iterating forward, such that the expression `(t == a)` is `true`. Returns `true` if an element was found, returns `false` otherwise.

`RWBoolean`
`findNext``(RWBoolean(*fn)(const_reference, void*), void* d);`

Advances self to the first element `t` encountered by iterating forward such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`. Returns `true` if an element was found, returns `false` otherwise.

`void`
`insertAfterPoint``(T* p);`

Inserts the pointer `p` into the container directly after the element referenced by `self`.

`T`
`key``();`

Returns the stored value referenced by `self`.

`RWBoolean`
`remove``();`

Removes the value referenced by `self` from the collection. `true` is returned if the removal is successful, `false` is returned otherwise.

`RWBoolean`
`removeNext``(const T);`

Removes the first element `t`, encountered by iterating self forward, such that the expression `(t == a)` is `true`. Returns `true` if an element was found and removed, returns `false` otherwise.

`RWBoolean`
`removeNext``(RWBoolean(*fn)(T, void*), void* d);`

Removes the first element `t,` encountered by iterating self forward, such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T a, void* d);
```

Client data may be passed through parameter `d`. Returns `true` if an element was found and removed, returns `false` otherwise.

```
void
reset();
void
reset(RWTValSlist<T>& l);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValSlist` to `reset()` will reset the iterator on the new container.

Synopsis

```
#include <rw/tvsrtdli.h>
RWTValSortedDlist<T,C> srtdlist;
```

Standard C++ Library Dependent!

RWTValSortedDlist **requires the Standard C++ Library.**

Description

This class maintains an always-sorted collection of values, implemented as a doubly-linked list.

Persistence

Isomorphic.

Example

In this example, a sorted doubly-linked list of `RWDate`s is exercised.

```
//
// tvsdldat.cpp
//
#include <rw/tvsrtdli.h>
#include <rw/rwdate.h>
#include <iostream.h>
#include <function.h>

main(){
  RWTValSortedDList<RWDate, less<RWDate> > lst;

  lst.insert(RWDate(10, "Aug", 1991));
  lst.insert(RWDate(9, "Aug", 1991));
  lst.insert(RWDate(1, "Sep", 1991));
  lst.insert(RWDate(14, "May", 1990));
  lst.insert(RWDate(1, "Sep", 1991));   // Add a duplicate
  lst.insert(RWDate(2, "June", 1991));

  for (int i=0; i<lst.entries(); i++)
    cout << lst[i] << endl;
  return 0;
}
```

Program Output:

```
05/14/90
06/02/91
08/09/91
08/10/91
09/01/91
09/01/91
```

Related Classes

RWTValSortedVector<T> is an alternative always-sorted collections.
RWTValDlist<T> is an unsorted doubly-linked list of values.

Class ***list<T,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef list<T,allocator>                   container_type;
typedef container_type::const_iterator      iterator;
typedef container_type::const_iterator      const_iterator;
typedef container_type::size_type           size_type;
typedef T                                   value_type;
typedef T&                                  reference;
typedef const T&                            const_reference;
```

Public Constructors

```
RWTValSortedDlist<T,C>();
```

Constructs an empty doubly-linked list.

```
RWTValSortedDlist<T,C>(const list<T,allocator>& lst);
```

Constructs a doubly-linked list by copying and sorting all elements of `lst`.

```
RWTValSortedDlist<T,C>(const RWTValSortedDlist<T,C>& rwlst);
```

Copy constructor.

```
RWTValSortedDlist<T,C>(size_type n, const T& val = T());
```

Constructs a doubly-linked list with `n` elements, each initialized to `val`.

```
RWTValSortedDlist<T,C>(const T* first, const T* last);
```

Constructs a doubly-linked list by copying and sorting elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

```
RWTValSortedDlist<T,C>&
operator=(const RWTValSortedDlist<T,C>& lst);
RWTValSortedDlist<T,C>&
operator=(const list<T,allocator>& lst);
```

Destroys all elements of self and replaces them by copying (and sorting, if necessary) all elements of `lst`.

```
bool
operator<(const RWTValSortedDlist<T,C>& lst) const;
bool
operator<(const list<T,allocator>& lst) const;
```

Returns `true` if self compares lexicographically less than `lst`, otherwise returns `false`. Assumes that type `T` has well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

```
bool
operator==(const RWTValSortedDlist<T,C>& lst) const;
bool
operator==(const list<T>& lst) const;
```

Returns `true` if self compares equal to `lst`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

```
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have prototype:

```
void yourfun(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(t==a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
bool
find(const_reference a, value_type& k) const;
```

If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d,
value_type& k) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
const_reference
first() const;
```

Returns a reference to the first element of self.

```
size_type
index(const_reference a) const;
```

Returns the position of the first item `t` in self such that `(t == a)`, or returns the static member `npos` if no such item exists.

```
size_type
index(bool (*fn)(const_reference,void*), void* d) const;
```

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
insert(const list<T,allocator>& a);
```

Adds the items from `a` to self in an order preserving manner. Returns the number of items inserted into self.

```
bool
insert(const_reference a);
```

Adds the item `a` to self. The collection remains sorted. Returns `true`.

```
bool
isEmpty() const;
```

Returns `true` if there are no items in the collection, `false` otherwise.

```
bool
isSorted() const;
```

Returns `true` if the collection is sorted relative to the supplied comparator object, `false` otherwise.

```
const_reference
last() const;
```

Returns a reference to the last item in the collection.

```
size_type
merge(const RWTValSortedDlist&<T,C> dl);
```

Inserts all elements of `dl` into self, preserving sorted order.

```
size_type
occurrencesOf(const_reference) const;
```

Returns the number of elements `t` in self such that the expression `(t == a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),
              void* d) const;
```

Returns the number of elements `t` in self such that the expression`((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self such that the expression `(t == a)` is `true` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self such that the expression `(t == a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
T
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
T
removeFirst();
```

Removes and returns the first item in the collection.

```
T
removeLast();
```

Removes and returns the first item in the collection.

```
list<T,allocator>&
std();
const list<T,allocator>&
std() const;
```

Returns a reference to the underlying C++-standard collection that serves as the implementation for self. It is your responsibility not to violate the ordering of the elements within the collection.

Static Public Data Member

```
const size_type  npos;
```

This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
      const RWTValSortedDlist<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValSortedDlist<T,C>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSortedDlist<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTValSortedDlist<T,C>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSortedDlist<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTValSortedDlist<T,C>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include<rw/tvsrtdli.h>
RWTValSortedDlist<T,C> dl;
RWTValSortedDlistIterator<T,C> itr(dl);
```

Standard C++ Library Dependent!

***RWTValSortedDlistIterator* requires the Standard C++ Library.**

Description

RWTValSortedDlistIterator is supplied with *Tools.h++* 7 to provide an iterator interface to ***RWTValSortedDlistIterator*** that is backward compatable with the container iterators provided in *Tools.h++* 6.x.

The order of iteration over an ***RWTValSortedDlist*** is dependent on the supplied comparator object supplied as applied to the values stored in the container.

The current item referenced by this iterator is undefined after construction or after a call to `reset()`. The iterator becomes valid after being advanced with either a preincrement or `operator()`.

For both `operator++` and `operator()`, iterating past the last element will return a value equivalent to boolean `false`. Continued increments will return a value equivalent to `false` until `reset()` is called.

Persistence

None

Examples

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

int main(){
   RWTValSortedDlist<RWCString, less<RWCString> > a;
   RWTValSortedDlistIterator<RWCString, less<RWCString> > itr(a);

   a.insert("John");
   a.insert("Steve");
   a.insert("Mark");
   a.insert("Steve");

   for(;itr();)
     cout << itr.key() << endl;

   return 0;
}
```

Program Output

```
John
Mark
Steve
Steve
```

Public Constructors

RWTValSortedDlistIterator<T,C>(RWTValSortedDlist<T,C>&s);

Creates an iterator for the sorted dlist `s`. The iterator begins in an undefined state and must be advanced before the first element will be accessible.

Public Member Operators

RWBoolean
operator()();

Advances `self` to the next element. If the iterator has advanced past the last item in the container, the element returned will be a nil pointer equivalent to boolean `false`.

RWBoolean
operator++();

Advances `self` to the next element. If the iterator has been reset or just created, self will reference the first element. If, before iteration, self referenced the last value in the list, self will now point to an undefined value distinct from the reset value and `false` will be returned. Otherwise, `true` is returned. Note: no postincrement operator is provided.

RWBoolean
operator+=(size_type n);

Behaves as if the `operator++` member function had been applied `n` times

RWBoolean
operator--();

Moves self back to the immediately previous element. If the iterator has been reset or just created, this operator will return `false`, otherwise it will return `true`. If self references the the first element, it will now be in the reset state. If self has been iterated past the last value in the list, it will now point to the last item in the list. Note: no postdecrement operator is provided.

RWBoolean
operator-=(size_type n);

Behaves as if the `operator--` member function had been applied `n` times

Public Member Functions

RWTValSortedDlist<T,C>*
container() const;

Returns a pointer to the collection being iterated over.

```
RWBoolean
findNext(const T a);
```

Advances self to the first element `t` encountered by iterating forward, such that the expression `(t == a)` is `true`. Returns `true` if such an element if found, `false` otherwise.

```
RWBoolean
findNext(RWBoolean(*fn)(T, void*), void* d);
```

Advances self to the first element `t` encountered by iterating forward, such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T a, void* d);
```

Client data may be passed through parameter `d`. Returns `true` if such an element if found, `false` otherwise.

```
T
key();
```

Returns the stored value referenced by self.

```
RWBoolean
remove();
```

Removes the stored value referenced by self from the collection. Returns `true` if the value was successfully removed, `false` otherwise.

```
RWBoolean
removeNext(const T);
```

Removes the first element `t`, encountered by iterating self forward, such that the expression `(t == a)` is `true`. Returns `true` if such an element is successfully removed, `false` otherwise.

```
RWBoolean
removeNext(RWBoolean(*fn)(T, void*), void* d);
```

Removes the first element `t,` encountered by iterating self forward, such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const T a, void* d);
```

Client data may be passed through parameter `d`. Returns `true` if such an element is successfully removed, `false` otherwise.

```
void
reset();
void
reset(RWTValSortedDlist<T,C>& l);
```

Resets the iterator so that after being advanced it will reference the first element of the collection. Using `reset()` with no argument will reset the iterator on the current container. Supplying a `RWTValSortedDlist` to `reset()` will reset the iterator on the new container.

Synopsis

```
#include <rw/tvsrtvec.h>
RWTValSortedVector<T,C> srtvec;
```

Please Note!

If you have the Standard C++ Library, use the interface described here. Otherwise, use the restricted interface to *RWTValSortedVector* described in Appendix A.

Description

This class maintains an always-sorted collection of values, implemented as a vector.

Persistence

Isomorphic

Example

In this example, a sorted vector of *RWDate*s is exercised.

```
//
// tvsvcdat.cpp
//
#include <rw/tvsrtvec.h>
#include <rw/rwdate.h>
#include <iostream.h>

main(){
  RWTValSortedVector<RWDate, less<RWDate> > vec;

  vec.insert(RWDate(10, "Aug", 1991));
  vec.insert(RWDate(9, "Aug", 1991));
  vec.insert(RWDate(1, "Sep", 1991));
  vec.insert(RWDate(14, "May", 1990));
  vec.insert(RWDate(1, "Sep", 1991));   // Add a duplicate
  vec.insert(RWDate(2, "June", 1991));

  for (int i=0; i<vec.entries(); i++)
    cout << vec[i] << endl;
  return 0;
}
```

Program Output:

```
05/14/90
06/02/91
08/09/91
08/10/91
09/01/91
09/01/91
```

Related Classes

RWTValSortedDlist<T,C> is an alternative always-sorted collection.

RWTValOrderedVector<T> is an unsorted vector of values.

Class ***vector<T,allocator>*** is the C++-standard collection that serves as the underlying implementation for this class.

Public Typedefs

```
typedef vector<T,allocator>                  container_type;
typedef container_type::const_iterator       iterator;
typedef container_type::const_iterator       const_iterator;
typedef container_type::size_type            size_type;
typedef T                                    value_type;
typedef const T&                             reference;
typedef const T&                             const_reference;
```

Public Constructors

`RWTValSortedVector<T,C>``();`
Constructs an empty vector.

`RWTValSortedVector<T,C>``(const vector<T,allocator>& vec);`
Constructs a vector by copying and sorting all elements of `vec`.

`RWTValSortedVector<T,C>``(const RWTValSortedVector<T,C>& rwvec);`
Copy constructor.

`RWTValSortedVector<T,C>``(size_type n, const T& val);`
Constructs a vector with `n` elements, each initialized to `val`.

`RWTValSortedVector<T,C>``(size_type n);`
Constructs an empty vector with a capacity of `n` elements.

`RWTValSortedVector<T,C>``(const T* first, const T* last);`
Constructs a vector by copying and sorting elements from the array of `T`s pointed to by `first`, up to, but not including, the element pointed to by `last`.

Public Member Operators

`bool`
`operator<``(const RWTValSortedVector<T,C>& vec) const;`
`bool`
`operator<``(const vector<T,allocator>& vec) const;`
Returns `true` if self compares lexicographically less than `vec`, otherwise returns `false`. Assumes that type `T` has well-defined less-than semantics (`T::operator<(const T&)` or equivalent).

`bool`
`operator==``(const RWTValSortedVector<T,C>& vec) const;`
`bool`
`operator==``(const vector<T,allocator>& vec) const;`
Returns `true` if self compares equal to `vec`, otherwise returns `false`. Two collections are equal if both have the same number of entries, and iterating through both collections produces, in turn, individual elements that compare equal to each other.

```
reference
operator()(size_type i);
const_reference
operator()(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` should be between 0 and one less then the number of entries, otherwise the results are undefined—*no bounds checking is performed*.

```
reference
operator[](size_type i);
const_reference
operator[](size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

Public Member Functions

```
void
apply(void (*fn)(const_reference,void*), void* d) const;
```

Applies the user-defined function pointed to by `fn` to every item in the collection. This function must have the prototype:

```
void yourfun(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
at(size_type i);
const_reference
at(size_type i) const;
```

Returns a reference to the `i`th element of self. Index `i` must be between 0 and one less then the number of entries in self, otherwise the function throws an exception of type *RW**BoundsErr***.

```
iterator
begin();
const_iterator
begin() const;
```

Returns an iterator positioned at the first element of self.

```
void
clear();
```

Clears the collection by removing all items from self. Each item will have its destructor called.

```
bool
contains(const_reference a) const;
```

Returns `true` if there exists an element `t` in self such that the expression`(t==a)` is `true`, otherwise returns `false`.

```
bool
contains(bool (*fn)(const_reference,void*), void* d) const;
```

Returns `true` if there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, otherwise returns `false`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
const T*
data();
```

Returns a pointer to the first element of the vector.

```
iterator
end();
const_iterator
end() const;
```

Returns an iterator positioned "just past" the last element in self.

```
size_type
entries() const;
```

Returns the number of items in self.

```
bool
find(const_reference a, value_type& k) const;
```

If there exists an element `t` in self such that the expression `(t == a)` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged.

```
bool
find(bool (*fn)(const_reference,void*), void* d,
     value_type& k) const;
```

If there exists an element `t` in self such that the expression `((*fn)(t,d))` is `true`, assigns `t` to `k` and returns `true`. Otherwise, returns `false` and leaves the value of `k` unchanged. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
reference
first();
const_reference
first() const;
```

Returns a reference to the first element of self.

`size_type`
`index``(const_reference a) const;`

Returns the position of the first item `t` in self such that `(t == a)`, or returns the static member `npos` if no such item exists.

`size_type`
`index``(bool (*fn)(const_reference,void*), void* d) const;`

Returns the position of the first item `t` in self such that`((*fn)(t,d))` is `true`, or returns the static member `npos` if no such item exists. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

`bool`
`insert``(const_reference a);`

Adds the item `a` to self. The collection remains sorted. Returns `true`.

`size_type`
`insert``(const vector<T,allocator>& a);`

Inserts all elements of `a` into self. The collection remains sorted. Returns the number of items inserted.

`bool`
`isEmpty``() const;`

Returns `true` if there are no items in the collection, `false` otherwise.

`bool`
`isSorted``() const;`

Returns `true` if the collection is sorted relative to the supplied comparator object, `false` otherwise.

`const_reference`
`last``() const;`

Returns a reference to the last item in the collection.

`size_type`
`length``() const;`

Returns the maximum number of elements which can be stored in self without first resizing.

`size_type`
`merge``(const RWTValSortedVector<T,C>& dl);`

Inserts all elements of `dl` into self, preserving sorted order.

`size_type`
`occurrencesOf``(const_reference a) const;`

Returns the number of elements `t` in self such that the expression `(t == a)` is `true`.

```
size_type
occurrencesOf(bool (*fn)(const_reference,void*),
              void* d) const;
```

Returns the number of elements `t` in self such that the expression `((*fn)(t,d))` is `true`. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
bool
remove(const_reference a);
```

Removes the first element `t` in self such that the expression `(t == a)` is `true` and returns `true`. Returns `false` if there is no such element.

```
bool
remove(bool (*fn)(const_reference,void*), void* d);
```

Removes the first element `t` in self such that the expression `((*fn)(t,d))` is `true` and returns `true`. Returns `false` if there is no such element. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
size_type
removeAll(const_reference a);
```

Removes all elements `t` in self such that the expression `(t == a)` is `true`. Returns the number of items removed.

```
size_type
removeAll(bool (*fn)(const_reference,void*), void* d);
```

Removes all elements `t` in self such that the expression `((*fn)(t,d))`is `true`. Returns the number of items removed. `fn` points to a user-defined tester function which must have prototype:

```
bool yourTester(const_reference a, void* d);
```

Client data may be passed through parameter `d`.

```
value_type
removeAt(size_type i);
```

Removes and returns the item at position `i` in self. This position must be between zero and one less then the number of entries in the collection, otherwise the function throws an exception of type *RW**BoundsErr***.

```
value_type
removeFirst();
```

Removes and returns the first item in the collection.

```
value_type
removeLast();
```
Removes and returns the first item in the collection.

```
void
resize(size_type n);
```
Modify, if necessary, the capacity of the vector to be at least as large as `n`.

```
vector<T,allocator>&
std();
const vector<T,allocator>&
std() const;
```
Returns a reference to the underlying C++-standard collection that serves as the implementation for self. It is your responsibility not to violate the ordering of the elements within the collection.

Static Public Data Member

```
const size_type  npos;
```
This is the value returned by member functions such as `index` to indicate a non-position. The value is equal to `~(size_type)0`.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
     const RWTValSortedVector<T,C>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValSortedVector<T,C>& coll);
```
Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSortedVector<T,C>& coll);
RWFile&
operator>>(RWFile& strm, RWTValSortedVector<T,C>& coll);
```
Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSortedVector<T,C>*& p);
RWFile&
operator>>(RWFile& strm, RWTValSortedVector<T,C>*& p);
```
Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

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.

Persistence

Isomorphic

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, 1996
March 17, 1996
March 18, 1996
March 19, 1996
March 20, 1996
March 21, 1996
March 22, 1996
```

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`.

```
const T&
operator()(size_t i) const;
T&
operator()(size_t i);
```

Returns a reference to 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.

```
const T&
operator[](size_t i) const;
T&
operator[](size_t i);
```

Returns a reference to 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_t
length() const;
```

Returns the length of the vector.

```
void
reshape(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`.

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 as needed basis. If an element is updated 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 first line. 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.

Persistence None

Example In this example, a virtual vector of objects of type `ErsatzInt` is exercised. A disk-based page heap is used for swapping space.

```
#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<10000L; i++)
    vec1[i] = i;      // Some compilers may need a cast here

  cout << vec1[100] << endl; // Prints "100"
  cout << vec1[300] << endl; // Prints "300"

  RWTValVirtualArray<ErsatzInt> vec2 = vec1.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.

```
void
operator=(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.

```
T
operator=(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. A copy of `val` is returned.

```
T
operator[](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.

```
T
val(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.

```
void
set(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.

```
void
reshape(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.

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 *RW**Handle***. 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 *RW**VirtualPageHeap*** 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 that can work with an abstract heap to bring things in and out of memory automatically. Indeed, this is what has been done with class *RW**TValVirtualArray**<T>*, which represents a virtual array of elements of type `T`. Elements are automatically swapped in as necessary as they are addressed.

Persistence

None

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;
  RWHandle  next;
};
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;

Deallocate 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.

Synopsis

```
#include <vstream.h>
```

(abstract base class)

Description

*RW**vios*** is an abstract base class. It defines an interface similar to the C++ streams class ***ios***. However, unlike ***ios***, it offers the advantage of not necessarily being associated with a ***streambuf***.

This is useful for classes that cannot use a ***streambuf*** in their implementation. An example of such a class is *RW**XDRistream***, where the XDR model does not permit ***streambuf*** functionality.

Specializing classes that do use ***streambuf***s in their implementation (*e.g.*, *RW**pistream***) can usually just return the corresponding ***ios*** function.

Persistence

None

Public Member Functions

```
virtual int
eof() = 0;
```

Returns a nonzero integer if an `EOF` has been encountered.

```
virtual int
fail() = 0;
```

Returns a nonzero integer if the fail or bad bit has been set. Normally, this indicates that some storage or retrieval has failed but that the stream is still in a usable state.

```
virtual int
bad() = 0;
```

Returns a nonzero integer if the bad bit has been set. Normally this indicates that a severe error has occurred from which recovery is probably impossible.

```
virtual int
good() = 0;
```

Returns a nonzero integer if no error bits have been set.

```
virtual int
rdstate() = 0;
```

Returns the current error state.

```
virtual void
clear(int v=0) = 0;
```

Sets the current error state to `v`. If `v` is zero, then this clears the error state.

operator void*();

If fail() then return 0 else return self.

*RW***vistream** → *RW***vios**

Synopsis

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

Description

Class *RW***vistream** is an abstract base class. It provides an interface for format-independent retrieval of fundamental types and arrays of fundamental types. Its counterpart, *RW***vostream**, provides a complementary interface for the storage of the fundamental types.

Because the interface of *RW***vistream** and *RW***vostream** 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 *RW***pistream** and *RW***postream**), a binary format (classes *RW***bistream** and *RW***bostream**), 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 *RW***vistream** and *RW***vostream** should act.

See class *RW***vostream** for additional explanations and examples of format-independent stream storage.

Persistence

None

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 Destructor

`virtual ~`**`RWvistream`**`();`

This virtual destructor allows specializing classes to deallocate any resources that they may have allocated.

Public Operators

`virtual RWvistream&`
`operator>>``(char& c) = 0;`

Get the next `char` from the input stream and store it in `c`.

```
virtual RWvistream&
operator>>(wchar_t& wc) = 0;
```
Get the next `wchar_t` 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`.

```
operator void*();
```
Inherited from *RW**vios***.

Public Member Functions

```
virtual int
get() = 0;
```
Get and return the next byte 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 wchar_t from the input stream, returning its value in wc.

```
virtual RWvistream&
get(unsigned char& c) = 0;
```

Get the next unsigned char from the input stream, returning its value in c.

```
virtual RWvistream&
get(char* v, size_t N) = 0;
```

Get a vector of chars and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit. Note that get retrieves raw characters and does not perform any conversions on speical characters such as "\n".

```
virtual RWvistream&
get(wchar_t* v, size_t N) = 0;
```

Get a vector of wide characterss and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit. Note that get retrieves raw characters and does not perform any conversions on speical characters such as "\n".

```
virtual RWvistream&
get(double* v, size_t N) = 0;
```

Get a vector of N doubles and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit.

```
virtual RWvistream&
get(float* v, size_t N) = 0;
```

Get a vector of N floats and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit.

```
virtual RWvistream&
get(int* v, size_t N) = 0;
```

Get a vector of N ints and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit.

```
virtual RWvistream&
get(long* v, size_t N) = 0;
```

Get a vector of N longs and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason,get stores what has already been retrieved from the stream into v, and sets the failbit.

```
virtual RWvistream&
get(short* v, size_t N) = 0;
```

Get a vector of N shorts and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason,get stores what has already been retrieved from the stream into v, and sets the failbit.

```
virtual RWvistream&
get(unsigned char* v, size_t N) = 0;
```

Get a vector of N unsigned chars and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit. Note that this member preserves ASCII numerical codes, not their corresponding character values. If you wish to restore a character string, use the function getString(char*, size_t).

```
virtual RWvistream&
get(unsigned short* v, size_t N) = 0;
```

Get a vector of N unsigned shorts and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit.

```
virtual RWvistream&
get(unsigned int* v, size_t N) = 0;
```

Get a vector of N unsigned ints and store them in the array beginning at v. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, get stores what has already been retrieved from the stream into v, and sets the failbit.

`virtual RWvistream&`
`get``(unsigned long* v, size_t N) = 0;`

Get a vector of `N unsigned long`s and store them in the array beginning at `v`. If the restore operation stops prematurely because there are no more data available on the stream, because an exception is thrown, or for some other reason, `get` stores what has already been retrieved from the stream into `v`, and sets the failbit.

`virtual RWvistream&`
`getString``(char* s, size_t N) = 0;`

Restores a character string from the input stream that was stored to the output stream with `RWvostream::putstring` 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 `N-1` characters have been read and the `Nth` character is not the string terminator, 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 that was stored to the output stream with `RWvostream::putstring` 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 `N-1` characters have been read and the `Nth` character is not the string terminator, then the failbit of the stream will be set. In either case, the string will be terminated with a null byte.

*RW***vostream** → *RW***vios**

Synopsis

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

Description

Class *RW***vostream** is an abstract base class. It provides an interface for format-independent storage of fundamental types and arrays of fundamental types. Its counterpart, *RW***vistream**, provides a complementary interface for the retrieval of variables of the fundamental types.

Because the interface of *RW***vistream** and *RW***vostream** 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 *RW***pistream** and *RW***postream**), a binary format (classes *RW***bistream** and *RW***bostream**), 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 *RW***vistream** and *RW***vostream** should act.

Note that 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 *RW***vistream**.

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 insertion (<<) and extraction (>>) operators seek to store and restore characters preserving their symbolic meaning. That is, 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 *RW***postream**.

Persistence

None

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 Destructor

`virtual ~RWvostream();`

This virtual destructor allows specializing classes to deallocate any resources that they may have allocated.

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 that `c` is treated as a character, not a number.

`virtual RWvostream&`
`operator<<(wchar_t wc) = 0;`

Store the `wchar_t wc` to the output stream. Note that `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 that `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.

```
operator void*();
```
Inherited from *RW**vios***.

Public Member Functions

```
virtual RWvostream&
flush();
```
Send the contents of the stream buffer to output immediately.

```
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 `wchar_t 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 `N 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 `N wchar_ts` 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 `N` `unsigned char`s 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 `N` `short`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const unsigned short* p, size_t N) = 0;
```

Store the vector of `N` `unsigned short`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const int* p, size_t N) = 0;
```

Store the vector of `N` `int`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const unsigned int* p, size_t N) = 0;
```

Store the vector of `N` `unsigned int`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const long* p, size_t N) = 0;
```

Store the vector of `N` `long`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const unsigned long* p, size_t N) = 0;
```

Store the vector of `N` `unsigned long`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const float* p, size_t N) = 0;
```

Store the vector of `N` `float`s starting at `p` to the output stream.

```
virtual RWvostream&
put(const double* p, size_t N) = 0;
```

Store the vector of `N` `double`s starting at `p` to the output stream.

```
virtual RWvostream&
putString(const char*s, size_t N);
```

Store the character string, *including embedded nulls*, starting at s to the output string.

Synopsis

```
#include <rw/wstring.h>
RWWString a;
```

Description

Class *RW**WString*** offers very powerful and convenient facilities for manipulating wide character strings.

This string class manipulates *wide characters* of the fundamental 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 *RW**WString*** constructors, and by the *RW**WString*** 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.

*RW**WString***s tolerate embedded nulls.

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 *RW**WSubString*** class supports substring extraction and modification operations.

Persistence

Simple

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 };`

Allow conversion from multibyte character strings to wide character strings. See constructor below.

`enum RWWString::ascii_ {ascii };`

Allow 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);`

Creates a string from the wide 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* sizeof(wchar_t)` bytes or `N` wide characters long.

`RWWString``(RWSize_T ic);`

Creates a string of length zero (the null string). The string's *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``(char c);`

Constructs a string containing the single character `c`.

```
RWWString(char 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 by simply stripping the high-order bit and, hence, is much faster than the more general constructor given immediately 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 character 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.

Type Conversion

```
operator
const wchar_t*() const;
```

Access to the *RWWString*'s data as a null terminated wide 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 *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 char* 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 RWWSubString& sub);
```
Assignment operator. The string will *copy* `sub`'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 wide 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 wide character string `cs` to self. Exactly `N` wide characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N*sizeof(wchar_t)` 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 = RWWString::exact) const;
int
compareTo(const wchar_t*   str,
          caseCompare = RWWString::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 `RWWString::ignoreCase`.

```
RWBoolean
contains(const RWWString& cs,
         caseCompare = RWWString::exact) const;
RWBoolean
contains(const wchar_t* str,
         caseCompare = RWWString::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 *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 occurrence of the wide character `c` in self. Returns `RW_NPOS` if there is no such character or if there is an embedded null prior to finding `c`.

```
size_t
first(wchar_t c, size_t) const;
```

Returns the index of the first occurrence of the wide character `c` in self. Continues to search past embedded nulls. Returns `RW_NPOS` if there is no such character.

```
size_t
first(const wchar_t* str) const;
```

Returns the index of the first occurrence in self of any character in `str`. Returns `RW_NPOS` if there is no match or if there is an embedded null prior to finding any character from `str`.

```
size_t
first(const wchar_t* str, size_t N) const;
```

Returns the index of the first occurrence in self of any character in `str`. Exactly `N` characters in `str` are checked *including any embedded nulls* so `str` must point to a buffer containing at least `N` wide characters. Returns `RW_NPOS` if there is no match.

```
unsigned
hash(caseCompare = RWWString::exact) const;
```

Returns a suitable hash value.

```
size_t
index(const wchar_t* pat,size_t i=0,
      caseCompare = RWWString::exact) const;
size_t
index(const RWWString& pat,size_t i=0,
      caseCompare = RWWString::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` wide characters of `cs` into self at position `pos`. Exactly `N` wide characters are copied, *including any embedded nulls*. Hence, the buffer pointed to by `cs` must be at least `N*sizeof(wchar_t)` 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` wide 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 string has zero length (*i.e.*, the null string).

```
size_t
last(wchar_t c) const;
```
Returns the index of the last occurrence in the string of the wide 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 wide 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*sizeof(wchart_t)` 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` wide 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 storing characters into wide string. Characters are then read from the input stream `s`, replacing previous contents of self, until trailing whitespace or an `EOF` is encountered. The trailing 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`, which must be no greater than `length()`, to the end of string. Returns a reference to self.

```
RWWString&
remove(size_t pos, size_t N);
```

Removes `N` wide characters or to the end of string (whichever comes first) starting at the position `pos`, which must be no greater than `length()`. Returns a reference to self.

```
RWWString&
replace(size_t pos, size_t N, const wchar_t* cs);
```

Replaces `N` wide characters or to the end of string (whichever comes first) starting at position `pos`, which must be no greater than `length()`, 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`, which must be no greater than `length()`, 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*sizeof(wchart_t)` 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`, which must be no greater than `length()`, 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`, which must be no greater than `length()`, 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.*, `L' '`) or truncating as necessary.

```
RWWSubString
strip(stripType s = RWWString::trailing, wchar_t c = L' ');
const RWWSubString
strip(stripType s = RWWString::trailing, wchar_t c = L' ')
      const;
```

Returns a substring of self where the character `c` has been stripped off the beginning, end, or both ends of the string. The first variant can be used as an lvalue. 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 = RWWString::exact);
const RWWSubString
subString(const wchar_t* cs, size_t start=0,
          caseCompare = RWWString::exact) const;
```

Returns a substring representing the first occurrence of the null-terminated string pointed to by "`cs`". Case sensitivity is according to the `caseCompare` argument; it defaults to `RWWString::exact`. The first variant can be used as an lvalue.

```
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 unsigned
hash(const RWWString& wstr);
```

Returns the hash value of `wstr` as returned by `wstr.hash(RWWString::exact)`.

```
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 wide 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 wide string should it shrink, and returns the old value. The initial setting is 15 wide 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 wide string. Returns the old value. The initial setting is 16 wide characters.

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 *RW**WString*** 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 *RW**File***, respectively.

```
RWvistream&
operator>>(RWvistream&, RWWString& str);
RWFile&
operator>>(RWFile&,     RWWString& str);
```

Restores a wide character string into `str` from a virtual stream or *RW**File***, 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()`.

Synopsis

```
#include <rw/wstring.h>
RWWString s(L"test string");
s(6,3);   // "tri"
```

Description

The class ***RWWSubString*** allows some subsection of an ***RWWString*** to be addressed by defining a *starting position* and an *extent*. For example the 7th through the 11th 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 length*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.

Persistence

None

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(15,0) = RWWString(L" three times");
 cout << s << endl;
 return 0;
}
```

Program output:

```
What I tell you three times is true.
```

Assignment Operators

```
void
operator=(const RWWString&);
```

Assignment from an ***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 wstr(L"Mary had a little lamb");
wchar_t dat[] = L"Perrier";
wstr(11,4) = dat; // "Mary had a Perrier"
```

Note that the number of characters selected need not match: if they differ, wstr will be resized appropriately. If self is the null substring, then the statement has no effect.

Indexing Operators

```
wchar_t
operator[](size_t i);
wchar_t&
operator[](size_t i) const;
```

Returns the ith 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 ith 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 that 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 (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 wide 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==()`

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 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.

Persistence

None

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();
```

Advance to the next token and return it as a substring. The tokens are delimited by any of the four wide characters in `L" \t\n\0"`. (space, tab, newline and null).

`RWWSubString`
`operator()``(const wchar_t* s);`

Advance to the next token and return it as a widesubstring. The tokens are delimited by any wide character in `s`, or any embedded wide null.

`RWWSubString`
`operator()``(const wchar_t* s,size_t num);`

Advance to the next token and return it as a substring. The tokens are delimited by any of the first `num` wide characters in `s.` Buffer `s` may contain embedded nulls, and must contain at least `num` wide characters. Tokens will not be delimited by nulls unless `s` contains nulls.

*RW***XDRistream** → *RW***vistream** → *RW***vios**
→ *RW***ios**

Synopsis

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

XDR xdr;
xdrstdio_create(&xdr, stdin, XDR_DECODE);
RWXDRistream rw_xdr(&xdr);
```

Description

Class *RW***XDRistream** is a portable input stream based on XDR routines. Class *RW***XDRistream** 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 *RW***XDRistream** enables one to decode an XDR structure to a machine representation. Class *RW***XDRistream** 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 iostreams and 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 *RW***XDRistream** object.

*RW***XDRistream** can be interrogated as to the status of the stream using member functions `bad()`, `clear()`, `eof()`, `fail()`, `good()`, and `rdstate()`.

Persistence

None

Example

The example that follows is a “reader” program that decodes an XDR structure from a file stream. The example for class *RW***XDRostream** is the “writer” program that encodes the XDR structures onto the file stream.

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","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 Constructor

RWXDRistream(XDR* xp);

Initialize an *RWXDRistream* from the XDR structure xp.

RWXDristream(streambuf*);

Initialize RWXDRistream with a pointer to streambuf. Streambuf must be already allocated.

RWXDRistream(istream&);

Initialize RWXDRistream with an input stream.

Public Destructor

~virtual **RWXDRistream**();

Deallocate previously allocated resources.

Public Member Functions

virtual int
get();

Redefined from class *RWvistream*. Gets and returns 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. This member only preserves ASCII numerical codes, not the coresponding character symbol.

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 *RW**vistream***. 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 *RW**vistream***. Gets a vector of `N` 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 *RW**vistream***. Gets a vector of `N` 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 *RW**vistream***. Gets a vector of `N` `double`s 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 *RW**vistream***. Gets a vector of `N` `float`s 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 *RW**vistream***. Gets a vector of `N` `int`s 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 *RW**vistream***. Gets a vector of `N` unsigned `int`s 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 *RW**vistream***. Gets a vector of `N` `long`s 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 *RW***vistream**. Gets a vector of `N` unsigned `long`s 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 *RW***vistream**. Gets a vector of `N` `short`s 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 *RW***vistream**. Gets a vector of `N` unsigned `short`s 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 *RW***vistream**. Gets a vector of `N` 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 *RW***vistream**. Restores a character string from the XDR input stream that was stored to the XDR output stream with `RWXDRistream::putstring` and stores the characters 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 `maxlen-1` characters have been read and the `maxlen`th character is not the string terminator, 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 *RW***vistream**. Gets the next character from the XDR input stream and stores it in `c`. If the operation fails, it sets the failbit. This member attempts to preserve the symbolic characters' values transmitted over the stream.

```
virtual RWvistream&
operator>>(double& d);
```

Redefined from class *RW***vistream**. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. 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 *RW**vistream***. Gets the next unsigned `short` from the XDR input stream and stores it in `s`. If the operation fails, it sets the failbit.

*RW***XDRostream** → *RW***vostream** → *RW***vios**

→ *RW***ios**

Synopsis

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

XDR xdr;
xdrstdio_create(&xdr, stdout, XDR_ENCODE) ;
RWXDRostream rw_xdr(&xdr);
```

Description

Class *RW***XDRostream** is a portable output stream based on XDR routines. Class *RW***XDRostream** 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 *RW***XDRostream** enables one to output from a stream and encode an XDR structure from a machine representation. Class *RW***XDRostream** 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 iostreams and 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 *RW***XDRostream** object.

*RW***XDRostream** can be interrogated as to the status of the stream using member functions `bad()`, `clear()`, `eof()`, `fail()`, `good()`, and `rdstate()`.

Persistence

None

Example

The example that follows is a “writer” program that encodes an XDR structure onto a file stream. The example for class *RW***XDRistream** 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 Constructor

RWXDRostream(XDR* xp);

Initialize a *RW***XDRostream** from the XDR structure xp.

RWXDRostream(streambuf*);

Initialize *RW***XDRostream** with a pointer to ***streambuf***. ***streambuf*** must already be allocated.

RWXDRostream(ostream&);

Initialize *RW***XDRostream** with an output stream.

Public Destructor

virtual ~**RWXDRostream**();

Deallocate previously allocated resources.

Public Member Functions

virtual RWvostream&
operator<<(const char* s);

Redefined from class *RW***vostream**. 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 *RW***vostream**. Store the character c to the output stream using the XDR format. Note that c is treated as a character, not a number. This member attempts to preserve the symbolic characters values transmitted over the stream.

virtual RWvostream&
operator<<(wchar_t wc);

Redefined from class *RW***vostream**. 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 *RW***vostream**. 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 *RW**vostream***. Store the `double d` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(float f);
```

Redefined from class *RW**vostream***. Store the `float f` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(int i);
```

Redefined from class *RW**vostream***. Store the integer `i` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(unsigned int i);
```

Redefined from class *RW**vostream***. Store the unsigned integer `i` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(long l);
```

Redefined from class *RW**vostream***. Store the `long l` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(unsigned long l);
```

Redefined from class *RW**vostream***. Store the unsigned `long l` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(short s);
```

Redefined from class *RW**vostream***. Store the `short s` to the output stream using the XDR format.

```
virtual RWvostream&
operator<<(unsigned short );
```

Redefined from class *RW**vostream***. Store the unsigned `short s` to the output stream using the XDR format.

```
virtual RWvostream&
put(char c);
```

Redefined from class *RW**vostream***. Store the character `c` to the output stream using the XDR format. If the operation fails, it sets the failbit. This member only preserves ASCII numerical codes, not the coresponding character symbol.

```
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 `N` 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 `N` 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 `N` `short`s 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 `N` unsigned `short`s 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 `N` 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 `N` 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 *RW***vostream**. Store the vector of `N 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 *RW***vostream**. Store the vector of `N` 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 *RW***vostream**. Store the vector of `N 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 *RW***vostream**. Store the vector of `N doubles` starting at `p` to the output stream using the XDR format. If the operation fails, it sets the failbit.

```
Virtual RWXDRostream&
```
flush`();`

Send the contents of the stream buffer to output immediately.

```
Virtual RWXDRostream&
```
putString`(const char*s, size_t N);`

Store the character string for retrieval by `RWXDRistream::getString`.

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-saving time is in use, the names and offsets from UTC (also known as GMT) for both standard and daylight-saving times, and the start and stop dates for daylight-saving 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. daylight-saving 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.

Persistence

None

Example

```
#include <rw/zone.h>
#include <rw/rwtime.h>
#include <rw/rstream.h>

main(){
 RWTime now;
 cout << now.asString('\0', RWZone::local()) << endl;
 cout << now.asString("%x %X", RWZone::utc()) << endl;
 return 0;
}
```

Enumerations

```
enum DstRule { NoDST, NoAm, WeEu };
```

Used by the static member function `dstRule()`, described below, and by constructors for classes derived from ***RWZone***.

```
enum 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 *RW**Zone***.

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-saving time in this zone.

```
virtual RWBoolean
daylightObserved() const = 0;
```

Returns `TRUE` if daylight-saving 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-saving time for this zone. The elements of the `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-saving time begins, or ends, for the year indicated by the `tm_year` member passed in. If daylight-saving time is not observed, the `struct tm` members are all set to a negative value. Note that in the southern hemisphere, daylight-saving 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-saving time zones represented, such as `"PST"` and `"PDT"`. Note that the current date and time have no effect on the return values of these functions.

Static Public Member Functions

```
static const RWZone&
local();
```

Returns a reference to an *RWZone* representing local time. By default this will be an instance of *RWZoneSimple* created with offsets and zone names from the operating system, with U.S. rules for daylight-saving 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-saving time corrections. By default this is an instance of *RWZoneSimple* with offset and zone name from the operating system.

```
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.

```
static constRWDaylightRule*
dstRule(DstRule rule = NoAm);
```

Returns one of the built-in daylight-saving time rules according to `rule`. Function `dstRule()` is provided for convenience in constructing *RWZoneSimple* instances for time zones in which common daylight-saving 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 (excluding the U.K.). See *RWZoneSimple* for more details. If `DstRule NoDST` is given, then `0` is returned. The result of calling `dstRule()` is normally passed to the *RWZoneSimple* constructor.

*RW**ZoneSimple*** → *RW**Zone***

Synopsis

```
#include <time.h>
#include <rw/zone.h>

RWZoneSimple myZone(USCentral);
```

Description

*RW**ZoneSimple*** is an implementation of the abstract interface defined by class *RW**Zone***. It implements a simple daylight-saving 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 the struct *RW**DaylightRule**,* which is described below.

Direct use of *RW**DaylightRule*** affords the most general interface to *RW**ZoneSimple***. However, a much simpler programmatic interface is offered, as illustrated by the examples below.

Three instances of *RW**ZoneSimple*** 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 member functions are set up according to the time zone facilities provided in the execution environment (typically 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-saving time rules.

Note for developers outside North America: for some time zones this default will not be correct because these time zones rely on the C standard global variable `_daylight`. This variable is set whenever any alternate time zone rule is available, whether it represents daylight-saving time or not. Also the periods of history affected by daylight-saving time may be different in your time zone from those in North America, causing the North American rule to be erroneously invoked. The best way to ensure that these default time zones are correct is to construct an *RW**ZoneSimple*** using an appropriate *RW**DaylightRule*** and initialize `RWZone::local()` and `RWZone::std()` with this value.

Other instances of *RW**ZoneSimple*** may be constructed to represent other time zones, and may be installed globally using *RW**Zone*** static member functions `RWZone::local(const RWZone*)` and `RWZone::standard(const RWZone*)`.

Persistence

None

Examples

To install US Central time as your global “local” time use:

```
RWZone::local(new RWZoneSimple(RWZone::USCentral));
```

To install Hawaiian time (where daylight-saving 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 *RW**Zone::DstRule***:

```
// last Sun in Apr to last in Oct:
     const RWDaylightRule  usRuleAuld =
     { 0, 0000, 1, { 3, 4, 0, 120 }, { 9, 4, 0, 120 } };
// first Sun in Apr to last in Oct
     const RWDaylightRule usRule67 =
     { &usRuleAuld, 1967, 1, { 3, 0, 0, 120 }, { 9, 4, 0, 120 } };
// first Sun in Jan to last in Oct:
     const RWDaylightRule usRule74 =
     { &usRule67, 1974, 1, { 0, 0, 0, 120 }, { 9, 4, 0, 120 } };
// last Sun in Feb to last in Oct
     const RWDaylightRule usRule75 =
    { &usRule74, 1975, 1, { 1, 4, 0, 120 }, { 9, 4, 0, 120 } };
// last Sun in Apr to last in Oct
     const RWDaylightRule usRule76 =
     { &usRule75, 1976, 1, { 3, 4, 0, 120 }, { 9, 4, 0, 120 } };
// first Sun in Apr to last in Oct
     const RWDaylightRule usRuleLate =
     { &usRule76, 1987, 1, { 3, 0, 0, 120 }, { 9, 4, 0, 120 } };

// last Sun in Mar to last in Sep
     const RWDaylightRule euRuleLate =
     { 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-saving time systems that cannot be represented with *RW**DaylightRule*** and *RW**ZoneSimple*** must be modeled by deriving from *RW**Zone*** and implementing its virtual functions.

For example, under Britain's 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 an *RW**Zone*** class all its own.

Constructors

```
RWZoneSimple(RWZone::StdZone zone,
             RWZone::DstRule = RWZone::NoAm);
```

Constructs an *RW**ZoneSimple*** instance using internally held *RW**DaylightRules***. This is the simplest interface to *RW**ZoneSimple***. The first argument is the time zone for which an *RW**ZoneSimple*** is to be constructed. The second argument is the daylight-saving time rule which is to be followed.

```
RWZoneSimple(const RWDaylightRule* rule,
             long tzoff,  const RWCString& tzname,
             long altoff, const RWCString& altname);
```

Constructs an *RW**ZoneSimple*** instance which daylight-saving 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-saving time is in effect. If `rule` is zero, daylight-saving time is not observed.

```
RWZoneSimple(long tzoff, const RWCString& tzname);
```

Constructs an *RW**ZoneSimple*** instance in which daylight-saving 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 *RW**ZoneSimple*** instance in which offsets and names are specified by the `StdZone` argument. Daylight-saving time is computed according to the `rule` argument, if non-zero; otherwise, DST is not observed.

struct RWDaylightRule

The *RW**DaylightRule*** struct passed to *RW**ZoneSimple***'s constructor can be a single rule for all years or can be the head of a chain of rules going backwards in time.

*RW***DaylightRule** is a struct with no constructors. It can be initialized with the syntax used in the Examples section above. The data members of this structure are as follows:

```
struct RWExport RWDaylightRule {
  RWDaylightRule const* next_;
  short firstYear_;
  char observed_;
  RWDaylightBoundary begin_;
  RWDaylightBoundary end_;
}
```

`RWDaylightRule const*`
`next_;`

Points to the next rule in a chain which continues backwards in time.

`short`
`firstYear_;`

Four digit representation of the year in which this rule first goes into effect.

`char`
`observed_;`

A boolean value that can be used to specify a period of years for which daylight-saving time is not observed.

`1` = Daylight-saving time is in effect during this period

`0` = Daylight-saving time is *not* in effect during this period

(Note that these are numeric values as distinguished from '`1`' and '`0`'.)

`RWDaylightBoundary`
`begin_;`

This structure indicates the time of year, to the minute, when DST begins during this period. (See *RW***DaylightBoundary** below.)

`RWDaylightBoundary`
`end_;`

This structure indicates the time of year, to the minute, when standard time resumes during this period. (See *RW***DaylightBoundary** below.)

struct RWDaylight-Boundary

```
struct RWExport RWDaylightBoundary {
  // this struct uses <time.h> struct tm conventions:
  int month_;    // [0..11]
  int week_;     // [0..4], or -1
  int weekday_;  // [0..6], 0=Sunday; or, [1..31] if week_== -1
  int minute_;   // [0..1439]  (Usually 2 AM, = 120)
};
```

`int`
`month_;`

The month from `(0 - 11`), where `0` = January.

`int`
`week_;`

A week of the month from (`0 - 4`), or `-1` if the following field is to represent a day within the month.

`int`
`weekday_;`

A day of the week from (`0 - 6`), where `0` = Sunday, or, if the `week_` field is `-1`, a day of the month from (`1 - 31`).

`int`
`minute_;`

Minutes after 12:00 AM, from (`0 - 1439`). For example, `120` = 2 AM.

Appendix A: Alternate Template Class Interfaces

If you do not have the Standard C++ Library, use the template class interfaces described in this Appendix. If you do have the Standard C++ Library use the interfaces described in the main section of the *Class Reference.*

Synopsis

```
#include <rw/tpdlist.h>
RWTPtrDlist<T> list;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 fundamental type. The class ***T*** must have:

- well-defined equality semantics (`T::operator==(const T&)`).

Persistence

Isomorphic

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 *RWT**ValDlist<T>.***

```
#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) {
    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* const&
operator[](size_t i) const;
```

Returns a pointer to the `i`th value in the list. 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 *RW**BoundsError*** will be thrown.

Public Member Functions

```
void
append(T* a);
```

Appends the item pointed to by `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 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* const&
at(size_t i) const;
```

Returns a pointer to the `i`th value in the list. 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 *RW**BoundsError*** will be thrown.

```
void
clear();
```

Removes all items from the collection.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* deletes them.

```
RWBoolean
contains(const 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`.

```
RWBoolean
contains(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_t
entries() const;
```

Returns the number of items that are currently in the collection.

```
T*
find(const T* target) const;
```

Returns a pointer to the first object encountered which is equal to the object pointed to by `target`, 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();
T* const&
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_t`
`index``(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_t`
`index``(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.

`void`
`insertAt``(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 *RW**BoundsError*** will be thrown.

`RWBoolean`
`isEmpty``() const;`

Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

`T*&`
`last``();`
`T* const&`
`last``() const;`

Returns a pointer to the last item in the list. The behavior is undefined if the list is empty.

`size_t`
`occurrencesOf``(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_t`
`occurrencesOf``(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`.

`void`
`prepend``(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_t`
`removeAll``(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_t`
`removeAll``(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 *RW**BoundsError*** 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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrDlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrDlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrDlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrDlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrDlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrDlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tpdlist.h>
RWTPtrDlist<T> list;
RWTPtrDlistIterator<T> iterator(list);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

None

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.

`RWBoolean`
`operator--``();`

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.

`RWBoolean`
`operator+=``(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.

```
RWBoolean
operator-=(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.

```
void
insertAfterPoint(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 which remains unchanged.

```
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. If the first element of the iterator's associated collection is removed, then the position of the iterator will be 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.

```
void
reset();
```

Resets the iterator to the state it had immediately after construction.

```
void
reset(RWTPtrDlist<T>& c);
```

Resets the iterator to iterate over the collection `c`.

Synopsis

```
#include <rw/tphdict.h>
unsigned hashFun(const K&);
RWTPtrHashDictionary<K,V> dictionary(hashFun);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

Description

RWTPtrHashDictionary<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 *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 fundamental 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 Rogue Wave class, then this requirement is usually trivial because most Rogue Wave objects know how to return a hashing value. In fact, classes *RWCString*, *RWDate*, *RWTime*, and *RWWString* contain static member functions called `hash` that can be supplied to the constructor as is. The function must have 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.

Persistence None

Example

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

main()  {
  RWTPtrHashDictionary<RWCString, RWDate>
    birthdays(RWCString::hash);
  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;

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

Program output:

```
April 12, 1975
```

Public Constructors

```
RWTPtrHashDictionary<K,V>(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<K,V>(const RWTPtrHashDictionary<K,V>& 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<K,V>&
operator=(const RWTPtrHashDictionary<K,V>& 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 lvalue.

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.

```
void
clear();
```

Removes all key value pairs from the collection.

```
void
clearAndDestroy();
```

Removes all key value pairs from the collection *and* deletes both the keys and the values.

```
RWBoolean
contains(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_t
entries() 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`.

```
void
insertKeyAndValue(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.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if the dictionary has no items in it, `FALSE` otherwise.

```
K*
remove(const 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`.

```
void
resize(size_t N);
```

Changes the number of buckets to `N`. This will result in all of the keys being rehashed.

Synopsis

```
#include <rw/tphdict.h>
unsigned hashFun(const K&);
RWTPtrHashDictionary<K,V> dictionary(hashFun);
RWTPtrHashDictionaryIterator<K,V> iterator(dictionary);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

Description

Iterator for class *RWTPtrHashDictionary<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 Rogue Wave 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.

Persistence

None

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.

```
void
reset();
```

Resets the iterator to the state it had immediately after construction.

```
void
reset(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> → *RWTPtrHashTable<T>*

Synopsis

```
#include <rw/tphset.h>
unsigned hashFun(const T&);
RWTPtrHashSet(hashFun) set;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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&)`).

Persistence

None

Example

This examples exercises a set of *RWCStrings*.

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

main()  {
  RWTPtrHashSet<RWCString> set(RWCString::hash);

  set.insert(new RWCString("one"));
  set.insert(new RWCString("two"));
  set.insert(new RWCString("three"));
  set.insert(new RWCString("one"));

  cout << set.entries() << endl;  // Prints "3"

  set.clearAndDestroy();
  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

```
RWTPtrHashSet<T>&
Union(const RWTPtrHashSet<T>& h);
```

Computes the union of self and `h`, modifying self and returning self.

```
RWTPtrHashSet<T>&
difference(const RWTPtrHashSet<T>& h);
```

Computes the disjunction of self and `h`, modifying self and returning self.

```
RWTPtrHashSet<T>&
intersection(const RWTPtrHashSet<T>& h);
```

Computes the intersection of self and `h`, modifying self and returning self.

```
RWTPtrHashSet<T>&
symmetricDifference(const RWTPtrHashSet<T>& h);
```

Computes the symmetric difference between self and `h`, modifying self and returning self.

```
RWBoolean
isSubsetOf(const RWTPtrHashSet<T>& h) const;
```

Returns `TRUE` if self is a subset of `h`.

```
RWBoolean
isProperSubsetOf(const RWTPtrHashSet<T>& h) const;
```

Returns `TRUE` if self is a proper subset of `h`.

```
RWBoolean
isEquivalent(const RWTPtrHashSet<T>& h) const;
```

Returns `TRUE` if self and `h` are identical.

```
RWBoolean
operator!=(const RWTPtrHashSet<T>& h) const;
```

Returns `FALSE` if self and `h` are identical.

```
void
apply(void (*applyFun)(T*, void*), void* d);
```

Inherited from class *RWTPtrHashTable<T>.*

```
void
clear();
```

Inherited from class *RWTPtrHashTable<T>.*

```
void
clearAndDestroy();
```

Inherited from class *RWTPtrHashTable<T>.*

```
RWBoolean
contains(const T* a) const;
```

Inherited from class *RWTPtrHashTable<T>.*

```
size_t
entries() const;
```

Inherited from class *RWTPtrHashTable<T>.*

```
T*
find(const T* target) const;
```

Inherited from class *RWTPtrHashTable<T>.*

```
void
insert(T* a);
```

Redefined from class *RWTPtrHashTable<T>* to allow an object of a given value to be inserted only once.

```
RWBoolean
isEmpty() const;
```

Inherited from class *RWTPtrHashTable<T>.*

```
size_t
occurrencesOf(const T* a) const;
```

Inherited from class *RWTPtrHashTable<T>.*

```
T*
remove(const T* a);
```

Inherited from class *RWTPtrHashTable<T>.*

```
size_t
removeAll(const T* a);
```

Inherited from class *RWTPtrHashTable<T>.*

```
void
resize(size_t N);
```

Inherited from class *RWTPtrHashTable<T>.*

Synopsis

```
#include <rw/tphasht.h>
unsigned hashFun(const T&);
RWTPtrHashTable<T> table(hashFun);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 fundamental 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 Rogue Wave class, then this requirement is usually trivial because most Rogue Wave objects know how to return a hashing value. In fact, classes *RW**CString***, *RW**Date***, *RW**Time***, and *RW**WString*** contain static member functions called `hash` that can be supplied to the constructor as is. The function must have 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.

Persistence

None

Example

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

main()  {
  RWTPtrHashTable<RWCString> table(RWCString::hash);
  RWCString *states[4] = { new RWCString("Alabama"),
                           new RWCString("Pennsylvania"),
                           new RWCString("Oregon"),
                           new RWCString("Montana") };

  table.insert(states[0]);
  table.insert(states[1]);
  table.insert(states[2]);
  table.insert(states[3]);

  RWCString key("Oregon");
  cout << "The table " <<
    (table.contains(&key) ? "does " : "does not ") <<
    "contain Oregon\n";

  table.removeAll(&key);

  cout << "Now the table " <<
    (table.contains(&key) ? "does " : "does not ") <<
    "contain Oregon";

  delete states[0];
  delete states[1];
  delete states[2];
  delete states[3];
  return 0;
}
```

Program output

```
The table does contain Oregon
Now 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&
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.

```
void
clear();
```
Removes all items from the collection.

```
void
clearAndDestroy();
```
Removes all items from the collection *and* deletes them.

```
RWBoolean
contains(const 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_t
entries() const;
```
Returns the number of items currently in the collection.

```
T*
find(const T* target) const;
```
Returns a pointer to the object which is equal to the object pointed to by target, or nil if no such object can be found. Equality is measured by the class-defined equality operator for type T.

```
void
insert(T* a);
```
Adds the object pointed to by a to the collection.

```
RWBoolean
isEmpty() const;
```
Returns TRUE if the collection has no items in it, FALSE otherwise.

`size_t`
`occurrencesOf``(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_t`
`removeAll``(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`.

`void`
`resize``(size_t N);`

Changes the number of buckets to `N`. This will result in all of the objects in the collection being rehashed.

Synopsis

```
#include <rw/tphasht.h>
RWTPtrHashTable<T> table;
RWTPtrHashTableIterator<T> iterator(table);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

None

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.

`void`
`reset();`

Resets the iterator to the state it had immediately after construction.

`void`
`reset(RWTPtrHashTable<T>& c);`

Resets the iterator to iterate over the collection `c`.

Synopsis

```
#include <rw/tpordvec.h>
RWTPtrOrderedVector<T> ordvec;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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&)`).

Persistence

Isomorphic

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* const&
operator()(size_t i) const;

Returns a pointer to 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 zero and the number of items in the collection less one. No bounds checking is performed.

T*&
operator[](size_t i);
T* const&
operator[](size_t i) const;

Returns a pointer to 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 zero and the number of items in the collection less one, or an exception of type *RW***BoundsError** 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* const&
at(size_t i) const;
```

Returns a pointer to 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 zero and the number of items in the collection less one, or an exception of type *RW**BoundsError*** will be thrown.

```
void
clear();
```

Removes all items from the collection.

```
void
clearAndDestroy();
```

Removes all items from the collection *and* deletes them.

```
RWBoolean
contains(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_t
entries() const;
```

Returns the number of items currently in the collection.

```
T*
find(const T* target) const;
```

Returns a pointer to the first object encountered which is equal to the object pointed to by `target`, or `nil` if no such object can be found. Equality is measured by the class-defined equality operator for type `T`.

```
T*&
first();
T* const&
first() const;
```

Returns a pointer to the first item in the vector. An exception of type *RW**BoundsError*** will occur if the vector is empty.

```
size_t
index(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`.

```
void
insert(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.

```
void
insertAt(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 *RW**BoundsError*** will occur.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

```
T*&
last();
T* const&
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 *RW**BoundsError*** will occur.

```
size_t
length() const;
```

Returns the number of items currently in the collection.

```
size_t
occurrencesOf(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`.

```
void
prepend(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_t
removeAll(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 *RW**BoundsError*** 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 *RW**BoundsError*** 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 *RW**BoundsError*** will be thrown if the list is empty.

```
void
resize(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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
      const RWTPtrOrderedVector<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrOrderedVector<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrOrderedVector<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrOrderedVector<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrOrderedVector<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrOrderedVector<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tpslist.h>
RWTPtrSlist<T> list;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 fundamental type. The class ***T*** must have:

- well-defined equality semantics (`T::operator==(const T&)`).

Persistence

Isomorphic

Example

In this example, a singly-linked list of *RW**Dates*** is exercised.

```
#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;
    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<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* const&
operator[](size_t i) const;

Returns a pointer to the `i`th value in the list. 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 *RW**BoundsError*** will be thrown.

Public Member Functions

void
append(T* a);

Appends the item pointed to by `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 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* const;
at(size_t i) const;

Returns a pointer to the `i`th value in the list. 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 *RW**BoundsError*** will be thrown.

```
void
clear();
```
Removes all items from the collection.

```
void
clearAndDestroy();
```
Removes all items from the collection *and* deletes them.

```
RWBoolean
contains(const 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`.

```
RWBoolean
contains(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_t
entries() const;
```
Returns the number of items that are currently in the collection.

```
T*
find(const T* target) const;
```
Returns a pointer to the first object encountered which is equal to the object pointed to by `target`, 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();
T* const&
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_t
index(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_t
index(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.

```
void
insertAt(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 *RW***BoundsError** will be thrown.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

```
T*&
last();
T* const&
last() const;
```

Returns a pointer to the last item in the list. The behavior is undefined if the list is empty.

```
size_t
occurrencesOf(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_t
occurrencesOf(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`.

```
void
prepend(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_t
removeAll(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_t
removeAll(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 *RWBoundsError* 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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTPtrSlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrSlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrSlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrSlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tpslist.h>
RWTPtrSlist<T> list;
RWTPtrSlistIterator<T> iterator(list);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

None

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.

`RWBoolean`
`operator+=``(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.

```
void
insertAfterPoint(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 which remains unchanged.

```
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.

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

Resets the iterator to the state it had immediately after construction.

```
void
```
`reset``(RWTPtrSlist<T>& c);`

Resets the iterator to iterate over the collection `c`.

Synopsis

```
#include <rw/tpsrtvec.h>
RWTPtrSortedVector<T> sortvec;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

Description

RWTPtrSortedVector<T> is a pointer-based *sorted* 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&)`);

Although it is possible to alter objects that are referenced by pointers within a ***RWTPtrSortedVector<T>***, it is dangerous since the changes may affect the way that `operator<()` and `operator==()` behave, causing the ***RWTPtrSortedVector<T>*** to become unsorted.

Persistence

Isomorphic

Example

This example inserts a set of dates into a sorted vector in no particular order, then prints them out in order.

```
#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, "Sep", 1991));
  vec.insert(new RWDate(14, "May", 1990));
  vec.insert(new RWDate(1, "Sep", 1991));  // Add a duplicate
```

```
  vec.insert(new RWDate(2, "June", 1991));

  for (int i=0; i<vec.length(); i++)
    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* const&
operator()(size_t i) const;

Returns a pointer to 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 zero and the number of items in the collection less one. No bounds checking is performed. When used as an `lvalue`, care must be taken so as not to disturb the sortedness of the collection.

T*&
operator[](size_t i);
T* const&
operator[](size_t i) const;

Returns a pointer to 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 zero and the number of items in the collection less one, or an exception of type *RW**BoundsError*** will be thrown. When used as an `lvalue`, care must be taken so as not to disturb the sortedness of the collection.

Public Member Functions

```
T*&
at(size_t i);
T* const&
at(size_t i) const;
```
Returns a pointer to 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 zero and the number of items in the collection less one, or an exception of type *RW***BoundsError** will be thrown. When used as an `lvalue`, care must be taken so as not to disturb the sortedness of the collection.

```
void
clear();
```
Removes all items from the collection.

```
void
clearAndDestroy();
```
Removes all items from the collection *and* deletes them.

```
RWBoolean
contains(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_t
entries() const;
```
Returns the number of items currently in the collection.

```
T*
find(const T* target) const;
```
Returns a pointer to the first object encountered which is equal to the object pointed to by `target`, 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* const&
first() const;
```
Returns a pointer to the first item in the vector. An exception of type *RW***BoundsError** will occur if the vector is empty.

```
size_t
index(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`.

```
void
insert(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.

```
RWBoolean
isEmpty() const;
```
Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

```
T* const&
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 *RW**BoundsError*** will occur.

```
size_t
length() const;
```
Returns the number of items currently in the collection.

```
size_t
occurrencesOf(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_t
removeAll(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 *RWBoundsError* 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 *RWBoundsError* 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 *RWBoundsError* will be thrown if the list is empty.

```
void
resize(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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTPtrSortedVector<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTPtrSortedVector<T>& coll);
```
Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSortedVector<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTPtrSortedVector<T>& coll);
```
Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTPtrSortedVector<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTPtrSortedVector<T>*& p);
```
Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tvdlist.h>
RWTValDlist<T> list;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 fundamental type. The class ***T*** must have:

- A default constructor;
- well-defined copy semantics (`T::T(const T&)` or equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent);
- well-defined equality semantics (`T::operator==(const T&)`).

Persistence

Isomorphic

Example

In this example, a doubly-linked list of user type `Dog` is exercised.

```
#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:
```

```
  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, const Dog& dog){
    str << dog.name;
    return str;}
};

main()  {
  RWTValDlist<Dog> terriers;
  terriers.insert("Cairn Terrier");   // automatic type conversion
  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 *RWBoundsError* 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.

const 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 *RWBoundsError* 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 *RWBoundsError* 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.

```
const 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 *RWBoundsError* 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_t
entries() const;
```

Returns the number of items that are currently in the collection.

```
RWBoolean
find(const T& target, T& k) const;
```

Returns `TRUE` if the list contains an object that is equal to the object `target` 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.

```
RWBoolean
find(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.

```
T&
first();
const T&
first() const;
```

Returns (but does not remove) the first item in the list. The behavior is undefined if the list is empty.

```
T
get();
```

Returns and removes the first item in the list. The behavior is undefined if the list is empty.

```
size_t
index(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_t
index(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.

```
void
insertAt(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 *RW**BoundsError*** will be thrown.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

```
T&
last();
const T&
last() const;
```

Returns (but does not remove) the last item in the list. The behavior is undefined if the list is empty.

```
size_t
occurrencesOf(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`.

```
void
prepend(const T& a);
```

Adds the item `a` to the beginning of the list.

```
RWBoolean
remove(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.

```
RWBoolean
remove(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_t
removeAll(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_t
removeAll(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`.

```
T
removeAt(size_t i);
```
Removes and returns the object at index `i`. An exception of type *RW**BoundsError*** 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 and returns the first item in the list. The behavior is undefined if the list is empty.

```
T
removeLast();
```
Removes and returns the last item in the list. The behavior is undefined if the list is empty.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValDlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValDlist<T>& coll);
```
Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValDlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValDlist<T>& coll);
```
Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValDlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValDlist<T>*& p);
```
Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tvdlist.h>
RWTValDlist<T> list;
RWTValDlistIterator<T> iterator(list);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

Isomorphic

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.

```
RWBoolean
operator--();
```

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.

```
RWBoolean
operator+=(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.

```
RWBoolean
operator-=(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.

```
RWBoolean
operator()();
```
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.

```
RWBoolean
findNext(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`.

```
RWBoolean
findNext(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.

```
void
insertAfterPoint(const T& a);
```
Inserts the value `a` into the iterator's associated collection in the position immediately after the iterator's current position.

```
T
key() const;
```
Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

```
RWBoolean
remove();
```
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`.

Synopsis

```
#include <rw/tvhdict.h>
unsigned hashFun(const K&);
RWTValHashDictionary<K,V> dictionary(hashFun);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

Description

*RW***TValHashDictionary<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 fundamental types. Classes ***K*** and ***V*** must have:

- well-defined copy semantics (`T::T(const T&)` or equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent).

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 Rogue Wave class, then this requirement is usually trivial because most Rogue Wave objects know how to return a hashing value. In fact, classes *RW***CString**, *RW***Date**, *RW***Time**, and *RW***WString** contain static member functions called `hash` that can be supplied to the constructor as is. The function must have 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.

Persistence

None

Example

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

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

  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 constant reference 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`.

```
void
clear();
```

Removes all items from the collection.

```
RWBoolean
contains(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_t
entries() const;
```

Returns the number of key-value pairs currently in the dictionary.

```
RWBoolean
find(const K& target, K& retKey) const;
```

Returns `TRUE` if the dictionary contains a key which is equal to `target` 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***.

`RWBoolean`
`findValue``(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***.

`RWBoolean`
`findKeyAndValue``(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`.

`void`
`insertKeyAndValue``(const K& key, const V& value);`

Inserts the key `key` and value `value` into the dictionary.

`RWBoolean`
`isEmpty``() const;`

Returns `TRUE` if the dictionary has no items in it, `FALSE` otherwise.

`RWBoolean`
`remove``(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`.

`void`
`resize``(size_t N);`

Changes the number of buckets to `N`, a relatively expensive operation if there are many items in the collection.

Synopsis

```
#include <rw/tvhdict.h>
unsigned hashFun(const K&);
RWTValHashDictionary<K,V> dictionary(hashFun);
RWTValHashDictonaryIterator<K,V> iterator(dictionary);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

None

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.

`RWBoolean`
`operator()``();`

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.

`K`
`key``() const;`

Returns the key at the iterator's current position. The results are undefined if the iterator is no longer valid.

```
void
reset();
```

Resets the iterator to the state it had immediately after construction.

```
void
reset(RWTValHashDictionary& c);
```

Resets the iterator to iterate over the collection `c`.

```
V
value() const;
```

Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

RWTValHashSet<T> → *RWTValHashTable<T>*

Synopsis

```
#include <rw/tvhset.h>
unsigned hashFun(const T&);
RWTValHashSet(hashFun) set;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent);
- well-defined equality semantics (`T::operator==(const T&)`).

Persistence

None

Example

This examples exercises a set of *RWCString*s.

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

main(){
  RWTValHashSet<RWCString> set(RWCString::hash);

  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 Member Functions

```
RWTValHashSet<T>&
Union(const RWTValHashSet<T>& h);
```
Computes the union of self and `h`, modifying self and returning self.

```
RWTValHashSet<T>&
difference(const RWTValHashSet<T>& h);
```
Computes the disjunction of self and `h`, modifying self and returning self.

```
RWTValHashSet<T>&
intersection(const RWTValHashSet<T>& h);
```
Computes the intersection of self and `h`, modifying self and returning self.

```
RWTValHashSet<T>&
symmetricDifference(const RWTValHashSet<T>& h);
```
Computes the symmetric difference between self and `h`, modifying self and returning self.

```
RWBoolean
isSubsetOf(const RWTValHashSet<T>& h) const;
```
Returns `TRUE` if self is a subset of `h`.

```
RWBoolean
isProperSubsetOf(const RWTValHashSet<T>& h) const;
```
Returns `TRUE` if self is a proper subset of `h`.

```
RWBoolean
isEquivalent(const RWTValHashSet<T>& h) const;
```
Returns `TRUE` if self and `h` are identical.

```
void
apply(void (*applyFun)(T&, void*), void* d);
```
Inherited from class *RWTValHashTable<T>.*

```
void
clear();
```
Inherited from class *RWTValHashTable<T>.*

```
RWBoolean
contains(const T& val) const;
```
Inherited from class *RWTValHashTable<T>.*

```
size_t
entries() const;
```
Inherited from class *RWTValHashTable<T>.*

```
RWBoolean
find(const T& target, T& k) const;
```
Inherited from class *RWTValHashTable<T>.*

`void`
`insert``(const T& val);`

Redefined from class *RWTValHashTable<T>* to allow an object of a given value to be inserted only once.

`RWBoolean`
`isEmpty``() const;`

Inherited from class *RWTValHashTable<T>.*

`size_t`
`occurrencesOf``(const T& val) const;`

Inherited from class *RWTValHashTable<T>.*

`RWBoolean`
`remove``(const T& val);`

Inherited from class *RWTValHashTable<T>.*

`size_t`
`removeAll``(const T& val);`

Inherited from class *RWTValHashTable<T>.*

`void`
`resize``(size_t N);`

Inherited from class *RWTValHashTable<T>.*

Synopsis

```
#include <rw/tvhasht.h>
unsigned hashFun(const T&);
RWTValHashTable<T> table(hashFun);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 fundamental type. The class ***T*** must have:

- well-defined copy semantics (`T::T(const T&)` or equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent);
- 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 Rogue Wave class, then this requirement is usually trivial because most Rogue Wave objects know how to return a hashing value. In fact, classes *RW**CString***, *RW**Date***, *RW**Time***, and *RW**WString*** contain static member functions called `hash` that can be supplied to the constructor as is. The function must have 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.

Persistence

None

Example

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

main()  {
  RWTValHashTable<RWCString> table(RWCString::hash);

  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 << "Now the table "
       << (table.contains("Oregon") ? "does " : "does not ")
       << "contain Oregon";
  return 0;
}
```

Program output

```
The table does contain Oregon
Now 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)(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`.

`void`
`clear``();`

Removes all items from the collection.

`RWBoolean`
`contains``(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_t`
`entries``() const;`

Returns the number of items currently in the collection.

`RWBoolean`
`find``(const T& target, T& k) const;`

Returns `TRUE` if the collection contains an item which is equal to `target` and puts the matching object into `k`. Returns `FALSE` otherwise and leaves `k` untouched. Equality is measured by the class-defined equality operator.

`void`
`insert``(const T& val);`

Inserts the value `val` into the collection.

`RWBoolean`
`isEmpty``() const;`

Returns `TRUE` if the collection has no items in it, `FALSE` otherwise.

`size_t`
`occurrencesOf``(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.

```
RWBoolean
remove(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_t
removeAll(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.

```
void
resize(size_t N);
```

Changes the number of buckets to `N`, a relatively expensive operation if there are many items in the collection.

Synopsis

```
#include <rw/tvhasht.h>
RWTValHashTable<T> table;
RWTValHashTableIterator<T> iterator(table);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

None

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.

`RWBoolean`
`operator()``();`

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.

`T`
`key``() const;`

Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

```
void
reset();
```

Resets the iterator to the state it had immediately after construction.

```
void
reset(RWTValHashTable<T>& c);
```

Resets the iterator to iterate over the collection `c`.

Synopsis

```
#include <rw/tvordvec.h>
RWTValOrderedVector<T> ordvec;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

Description

*RWT***ValOrderedVector<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 equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent);
- 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 ensure that the collection's capacity is not much greater than its length, otherwise unnecessary resources will be tied up.

Persistence

Isomorphic

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);
const T&
operator()(size_t i) const;

Returns 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 zero and the number of items in the collection less one. No bounds checking is performed.

T&
operator[](size_t i);
const T&
operator[](size_t i) const;

Returns 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 zero and the number of items in the collection less one, or an exception of type *RW**BoundsError*** 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);
const 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 *RW**BoundsError*** will be thrown.

```
void
clear();
```

Removes all items from the collection.

```
RWBoolean
contains(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_t
entries() const;
```

Returns the number of items currently in the collection.

```
RWBoolean
find(const T& target, T& ret) const;
```

Performs a linear search and returns `TRUE` if the vector contains an object that is equal to the object `target` 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.

```
T&
first();
const T&
first() const;
```

Returns the first item in the collection. An exception of type *RW**BoundsError*** will occur if the vector is empty.

```
size_t
index(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.

```
void
insert(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.

```
void
insertAt(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 *RW**BoundsError*** will occur.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

```
T&
last();
const T&
last() const;
```

Returns the last item in the collection. If there are no items in the collection then an exception of type *RW**BoundsError*** will occur.

```
size_t
length() const;
```

Returns the number of items currently in the collection.

```
size_t
occurrencesOf(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.

```
void
prepend(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.

```
RWBoolean
remove(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_t
removeAll(const T& a);
```

Removes all items which are equal to `a`, returning the number removed. Equality is measured by the class-defined equality operator.

```
T
removeAt(size_t i);
```

Removes and returns the object at index `i`. An exception of type *RWBoundsError* 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 and returns the first object in the collection. An exception of type *RWBoundsError* will be thrown if the list is empty.

```
T
removeLast();
```

Removes and returns the last object in the collection. An exception of type *RWBoundsError* will be thrown if the list is empty.

```
void
resize(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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
      const RWTValOrderedVector<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValOrderedVector<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValOrderedVector<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValOrderedVector<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValOrderedVector<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValOrderedVector<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tvslist.h>
RWTValSlist<T> list;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 fundamental type. The class ***T*** must have:

- A default constructor;
- well-defined copy semantics (`T::T(const T&)` or equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent);

well-defined equality semantics (`T::operator==(const T&)`).

Persistence

Isomorphic

Example

In this example, a singly-linked list of *RW**Dates*** is exercised.

```
#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;
  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&
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 *RW**BoundsError*** 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.

```
const 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 *RW**BoundsError*** 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 *RW**BoundsError*** 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.

```
const 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 *RW**BoundsError*** 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_t
entries() const;
```

Returns the number of items that are currently in the collection.

```
RWBoolean
find(const T& target, T& k) const;
```

Returns `TRUE` if the list contains an object that is equal to the object `target` 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.

```
RWBoolean
find(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.

```
T&
first();
const T&
first() const;
```

Returns but does not remove the first item in the list. The behavior is undefined if the list is empty.

```
T
get();
```

Returns and removes the first item in the list. The behavior is undefined if the list is empty.

```
size_t
index(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_t
index(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.

```
void
insertAt(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 *RW**BoundsError*** will be thrown.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if there are no items in the list, `FALSE` otherwise.

```
T&
last();
const T&
last() const;
```

Returns but does not remove the last item in the list. The behavior is undefined if the list is empty.

```
size_t
occurrencesOf(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`.

```
void
prepend(const T& a);
```

Adds the item `a` to the beginning of the list.

```
RWBoolean
remove(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.

```
RWBoolean
remove(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_t
removeAll(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_t
removeAll(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`.

```
T
removeAt(size_t i);
```

Removes and returns the object at index `i`. An exception of type *RWBoundsError* 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 and returns the first item in the list. The behavior is undefined if the list is empty.

```
T
removeLast();
```

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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm, const RWTValSlist<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValSlist<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSlist<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValSlist<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSlist<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValSlist<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Synopsis

```
#include <rw/tvslist.h>
RWTValSlist<T> list;
RWTValSlistIterator<T> iterator(list);
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 Rogue Wave 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.

Persistence

None

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.

```
RWBoolean
operator+=(size_t n);
```

Advances the iterator `n` positions. Returns `TRUE` if the new position is valid, `FALSE` otherwise.

```
RWBoolean
operator()();
```

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.

```
RWBoolean
findNext(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`.

```
RWBoolean
findNext(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 then returns `TRUE`, or `FALSE` if there is no such element.

```
void
insertAfterPoint(const T& a);
```
Inserts the value `a` into the iterator's associated collection in the position immediately after the iterator's current position.

```
T
key() const;
```
Returns the value at the iterator's current position. The results are undefined if the iterator is no longer valid.

```
RWBoolean
remove();
```
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.

```
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(RWTValSlist<T>& c);
```

Resets the iterator to iterate over the collection `c`.

Synopsis

```
#include <rw/tvsrtvec.h>
RWTValSortedVector<T> sortvec;
```

Please Note!

If you do not have the Standard C++ Library, use the interface described here. Otherwise, use the interface described in the Class Reference.

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 equivalent);
- well-defined assignment semantics (`T::operator=(const T&)` or equivalent);
- 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 ensure that the collection's capacity is not much greater than its length, otherwise unnecessary resources will be tied up.

Although it is possible to alter objects that are contained in a ***RWTValSortedVector<T>,*** it is dangerous since the changes may affect the way that `operator<()` and `operator==()` behave, causing the ***RWTValSortedVector<T>*** to become unsorted.

Persistence

Isomorphic

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>

{
  RWTValSortedVector<RWDate> vec;

  vec.insert(RWDate(10, "Aug", 1999));
  vec.insert(RWDate(9, "Aug", 1999));
  vec.insert(RWDate(1, "Sept", 1999));
  vec.insert(RWDate(14, "May", 1999));
  vec.insert(RWDate(1, "Sept", 1999));     // Add a duplicate
  vec.insert(RWDate(2, "June", 1999));

  for (int i=0; i<vec.length(); i++)
    cout << vec[i] << endl;
  return 0;
}
```

Program output

```
May 14, 1999
June 2, 1999
August 9, 1999
August 10, 1999
September 1, 1999
September 1, 1999
```

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);
const T&
operator()(size_t i) const;

Returns 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 zero and the number of items in the collection less one. No bounds checking is performed. When used as an `lvalue`, care must be taken so as not to disturb the sortedness of the collection.

T&
operator[](size_t i);
const T&
operator[](size_t i) const;

Returns 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 zero and the number of items in the collection less one, or an exception of type

RWBoundsError will be thrown. When used as an `lvalue`, care must be taken so as not to disturb the sortedness of the collection.

Public Member Functions

```
T&
at(size_t i);
const 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 *RWBoundsError* will be thrown. When used as an `lvalue`, care must be taken so as not to disturb the sortedness of the collection.

```
void
clear();
```

Removes all items from the collection.

```
RWBoolean
contains(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_t
entries() const;
```

Returns the number of items currently in the collection.

```
RWBoolean
find(const T& target, T& ret) const;
```

Performs a binary search and returns `TRUE` if the vector contains an object that is equal to the object `target` 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.

```
const T&
first() const;
```

Returns the first item in the collection. An exception of type *RWBoundsError* will occur if the vector is empty.

```
size_t
index(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.

```
void
insert(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.

```
RWBoolean
isEmpty() const;
```

Returns `TRUE` if there are no items in the collection, `FALSE` otherwise.

```
const T&
last() const;
```

Returns the last item in the collection. If there are no items in the collection then an exception of type *RW**BoundsError*** will occur.

```
size_t
length() const;
```

Returns the number of items currently in the collection.

```
size_t
occurrencesOf(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.

```
RWBoolean
remove(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_t
removeAll(const T& a);
```

Removes all items which are equal to `a`, returning the number removed. Equality is measured by the class-defined equality operator.

```
T
removeAt(size_t i);
```

Removes and returns the object at index `i`. An exception of type *RW**BoundsError*** 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 and returns the first object in the collection. An exception of type *RW**BoundsError*** will be thrown if the list is empty.

```
T
removeLast();
```

Removes and returns the last object in the collection. An exception of type *RW**BoundsError*** will be thrown if the list is empty.

```
void
resize(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.

Related Global Operators

```
RWvostream&
operator<<(RWvostream& strm,
       const RWTValSortedVector<T>& coll);
RWFile&
operator<<(RWFile& strm, const RWTValSortedVector<T>& coll);
```

Saves the collection `coll` onto the output stream `strm`, or a reference to it if it has already been saved.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSortedVector<T>& coll);
RWFile&
operator>>(RWFile& strm, RWTValSortedVector<T>& coll);
```

Restores the contents of the collection `coll` from the input stream `strm`.

```
RWvistream&
operator>>(RWvistream& strm, RWTValSortedVector<T>*& p);
RWFile&
operator>>(RWFile& strm, RWTValSortedVector<T>*& p);
```

Looks at the next object on the input stream `strm` and either creates a new collection off the heap and sets `p` to point to it, or sets `p` to point to a previously read instance. If a collection is created off the heap, then you are responsible for deleting it.

Information on this document

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.

This document from the document archive refers to a product version which was released a considerable time ago or which is no longer marketed.

Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.

Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format *…@ts.fujitsu.com*.

The Internet pages of Fujitsu Technology Solutions are available at *http://ts.fujitsu.com/*...
and the user documentation at *http://manuals.ts.fujitsu.com*.

Hinweise zum vorliegenden Dokument

Zum 1. April 2009 ist Fujitsu Siemens Computers in den alleinigen Besitz von Fujitsu übergegangen. Diese neue Tochtergesellschaft von Fujitsu trägt seitdem den Namen Fujitsu Technology Solutions.

Das vorliegende Dokument aus dem Dokumentenarchiv bezieht sich auf eine bereits vor längerer Zeit freigegebene oder nicht mehr im Vertrieb befindliche Produktversion.

Bitte beachten Sie, dass alle Firmenbezüge und Copyrights im vorliegenden Dokument rechtlich auf Fujitsu Technology Solutions übergegangen sind.

Kontakt- und Supportadressen werden nun von Fujitsu Technology Solutions angeboten und haben die Form *…@ts.fujitsu.com*.

Die Internetseiten von Fujitsu Technology Solutions finden Sie unter *http://de.ts.fujitsu.com/*..., und unter *http://manuals.ts.fujitsu.com* finden Sie die Benutzerdokumentation.