rptr Reference Manual

1.1.0

Generated by Doxygen 1.3.3

Wed May 12 15:35:34 2004

Contents

Chapter 1

rptr Main Page

This page documents a simple and straight-forward reference counting pointer implementation (**rptr::Rptr**(p. 5)). There is also a container that acts as a 'vector with gaps' (**rptr::Rvec**(p. 9)). Such a container is not a part of STL, but I find it quite useful in some situations.

This package is copyright protected under the GPL terms, which means that it is what is called 'free software'. (see `COPYING`)

Defining the macro RPTR_DEBUG activates a runtime check for null pointer dereferencing. On error an assertion fails and the program aborts. Together with a standard debugger this will hopefully help in finding your bugs easily.

Beware that the runtime checks will give an efficiency penalty.

Also beware that compiling only some files with RPTR_DEBUG may sometimes produce strange results because inlined functions may be defined differently in different object files. It is recommended not to link together files compiled with different RPTR_DEBUG macro settings.

Chapter 2

rptr Compound Index

2.1 rptr Compound List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 3

rptr Class Documentation

3.1 rptr::Rptr< T > Class Template Reference

General, reference counting smart pointer class.

`#include <rptr.h>`

Public Member Functions

Construction

- **Rptr** ()
- **Rptr** (const **Rptr** &src)
- **Rptr** (T ∗data0, bool owner0=true)
- **Rptr** & **Set** (T ∗src, bool owner0=true)
- ∼**Rptr** ()

const

- T & **operator** ∗ () const
- T ∗ **operator** → () const
- template<class TT> **operator const Rptr** () const
- T ∗ **Ptr** () const
- bool **IsOwner** () const
- T ∗ **Drop** () const
- bool **operator==** (const **Rptr** &rhs) const
- bool **operator==** (const T ∗rhs) const
- bool **operator!=** (const **Rptr** &rhs) const
- bool **operator!=** (const T ∗rhs) const

nonconst

- **Rptr** & **operator=** (const **Rptr** &src)
- **Rptr** & **operator=** (T ∗src)
- template<class TT> **Rptr** & **DynamicCast** (const **Rptr**< TT > &src)
- template<class TT> **Rptr** & **StaticCast** (const **Rptr**< TT > &src)

3.1.1 Detailed Description

template<class T> class rptr::Rptr< T >

General, reference counting smart pointer class.

This smart pointer provides the following features:

- Data is automatically deleted when (and only when) the last refering **Rptr**(p. 5) is either destructed or set to point to other data. (Standard reference counting pointer behaviour)
- The automatic deletion can be disabled so that data with other storage than dynamic can be allowed as destination for pointer object.
- Implicit cast from **Rptr**(p. 5)<A> & to const **Rptr**(p. 5)<B> & is supported iff implicit conversion from A ∗ & to B ∗ const & is legal (thus implicit cast rules for built-in pointer types are imitated). The casting is done using a reinterpret_cast. In theory the result is undefined according to ANSII c++, but in practice this works with today's compilers. It increases efficiency since in many cases pointer copying can be avoided. (Usually, copying a reference counting pointer always causes some performance penalty because the constructors and destructors must modify the reference counter object.)
- RTTI dynamic_cast be done using member function DynamicCast(TT ∗).
- static_cast be done using member function StaticCast(TT ∗).
- Self assignment is safe.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 template<class T> rptr::Rptr< T >::Rptr () `[inline]`

Default constructor

3.1.2.2 template<class T> rptr::Rptr< T >::Rptr (const Rptr< T > & *src*) `[inline]`

Copy constructor

3.1.2.3 template<class T> rptr::Rptr< T >::Rptr (T ∗ *data0*, bool *owner0* = true) `[inline]`

Constructor from built-in pointer type

- `data0` Pointer to data
- `owner0` Set this argument to false if object is not to be deleted by the reference counting mechanism e.g. when initializing a pointer object with the address of an object that has not been allocated with operator new.

3.1.2.4 template<class T> rptr::Rptr< T >::∼Rptr () `[inline]`

Destructor

3.1.3 Member Function Documentation

3.1.3.1 template<class T> T∗ rptr::Rptr< T >::Drop () const [inline]

Disable reference counting mechanism for the object this pointer points to.

3.1.3.2 template<class T> template<class TT> Rptr& rptr::Rptr< T >::DynamicCast (const Rptr< TT > & *src*) [inline]

Dynamic cast (assigns with 0 on failure)

3.1.3.3 template<class T> bool rptr::Rptr< T >::IsOwner () const [inline]

Returns:

true if the object pointed to will be deleted by the reference counting mechanism.

3.1.3.4 template<class T> T& rptr::Rptr< T >::operator ∗ () const [inline]

Dereferencing

3.1.3.5 template<class T> template<class TT> rptr::Rptr< T >::operator const Rptr () const [inline]

Pointer reference cast operator

3.1.3.6 template<class T> bool rptr::Rptr< T >::operator!= (const T ∗ *rhs*) const [inline]

Inequality, built-in pointer right-hand side

3.1.3.7 template<class T> bool rptr::Rptr< T >::operator!= (const Rptr< T > & *rhs*) const [inline]

Inequality

3.1.3.8 template<class T> T∗ rptr::Rptr< T >::operator → () const [inline]

Member dereferencing

3.1.3.9 template<class T> Rptr& rptr::Rptr< T >::operator= (T ∗ *src*) [inline]

Allow assignment from built-in pointer type

3.1.3.10 template<class T> Rptr& rptr::Rptr< T >::operator= (const Rptr< T > & *src*) [inline]

Assignment operator

3.1.3.11 template<class T> bool rptr::Rptr< T >::operator== (const T ∗ *rhs*) const `[inline]`

Equality, built-in pointer right-hand side

3.1.3.12 template<class T> bool rptr::Rptr< T >::operator== (const Rptr< T > & *rhs*) const `[inline]`

Equality

3.1.3.13 template<class T> T∗ rptr::Rptr< T >::Ptr () const `[inline]`

Allow access to built-in pointer

3.1.3.14 template<class T> Rptr& rptr::Rptr< T >::Set (T ∗ *src*, bool *owner0* = true) `[inline]`

Reinitialize the pointer

See also:
Rptr(T ∗, bool)(p. 6)

3.1.3.15 template<class T> template<class TT> Rptr& rptr::Rptr< T >::StaticCast (const Rptr< TT > & *src*) `[inline]`

Static cast

The documentation for this class was generated from the following file:

- rptr.h

3.2 rptr::Rvec< T > Class Template Reference

A vector class template allowing gaps.

`#include <rvec.h>`

Public Member Functions

Construction

- **Rvec** ()

const

- const_reference **operator[ ]** (size_type n) const
- const_pointer **ptr** (size_type n) const
- **const_iterator begin** () const
- **const_iterator end** () const
- int **ibegin** () const
- int **inext** (int id) const
- int **iprev** (int id) const
- size_type **id** (const **iterator** &it) const
- size_type **id** (const **const_iterator** &it) const
- size_type **size** () const
- bool **defined** (size_type n) const
- bool **empty** () const

nonconst

- **iterator begin** ()
- **iterator end** ()
- pointer **ptr** (size_type n)
- reference **operator[ ]** (size_type n)
- void **swap** (const **Rvec** &src)
- size_type **insert** (pointer x)
- size_type **set** (size_type pos, pointer x)
- void **erase** (size_type pos)
- void **erase** (const **iterator** &i)
- void **clear** ()

3.2.1 Detailed Description

template<class T> class rptr::Rvec< T >

A vector class template allowing gaps.

This template class behaves much like a vector<T> class, but it also allows 'gaps' within valid index range. When erasing an element a gap is created rather than shifting all elements with higher index one step down as in a vector<T> container. In addition, the elements may be objects of any type derived from the template argument type T.

This container is a good choice if you want a convenient vector for polymorphic types and if robustness has higher priority than efficiency.

The template class fulfills the requirements for a STL container (hopefully, this should perhaps be more thorougly tested),

Feature summary:

- Random access with constant time using operator []
- Reference counting pointer used to hold elements (**rptr::Rptr**(p. 5))
- Can hold elements of any type derived from template argument T
- Supports gaps in valid index range
- exception mechanism is used to handle

3.2.2 Constructor & Destructor Documentation

3.2.2.1 template<class T> rptr::Rvec< T >::Rvec () `[inline]`

Default constructor

3.2.3 Member Function Documentation

3.2.3.1 template<class T> iterator rptr::Rvec< T >::begin () `[inline]`

Returns:

const_iterator(p. 13) pointing to first non-empty element position or **end()**(p. 10) if container is empty

3.2.3.2 template<class T> const_iterator rptr::Rvec< T >::begin () const `[inline]`

Returns:

const_iterator(p. 13) pointing to first non-empty element position or **end()**(p. 10) if container is empty

3.2.3.3 template<class T> void rptr::Rvec< T >::clear () `[inline]`

Erase all elements

3.2.3.4 template<class T> bool rptr::Rvec< T >::defined (size_type *n*) const `[inline]`

Returns:

true iff position *n* contains a defined element

3.2.3.5 template<class T> bool rptr::Rvec< T >::empty () const `[inline]`

Returns:

true iff no elements exist in container

3.2.3.6 template<class T> iterator rptr::Rvec< T >::end () `[inline]`

Returns:

const_iterator(p. 13) after last defined element position

3.2.3.7 template<class T> const_iterator rptr::Rvec< T >::end () const `[inline]`

Returns:

const_iterator(p. 13) after last defined element position

3.2.3.8 template<class T> void rptr::Rvec< T >::erase (const iterator & *i*)

Erase an element

3.2.3.9 template<class T> void rptr::Rvec< T >::erase (size_type *pos*) `[inline]`

Erase an element

3.2.3.10 template<class T> int rptr::Rvec< T >::ibegin () const `[inline]`

Returns:

position of first defined element

3.2.3.11 template<class T> size_type rptr::Rvec< T >::id (const const_iterator & *it*) const `[inline]`

Returns:

integer index of given iterator *it*

3.2.3.12 template<class T> size_type rptr::Rvec< T >::id (const iterator & *it*) const `[inline]`

Returns:

integer index of given iterator *it*

3.2.3.13 template<class T> int rptr::Rvec< T >::inext (int *id*) const `[inline]`

Returns:

position of next defined element

3.2.3.14 template<class T> size_type rptr::Rvec< T >::insert (pointer *x*) `[inline]`

Insert a new object into container.

Returns:

index of inserted object.

3.2.3.15 template<class T> int rptr::Rvec< T >::iprev (int *id*) const `[inline]`

Returns:

position of previous defined element

3.2.3.16 template<class T> reference rptr::Rvec< T >::operator[] (size_type *n*) [inline]

Returns:

element at position n. Throws `std::out_of_range` if no element exists at n

3.2.3.17 template<class T> const_reference rptr::Rvec< T >::operator[] (size_type *n*) const [inline]

Returns:

element at position n. Throws `std::out_of_range` if no element exists at n

3.2.3.18 template<class T> pointer rptr::Rvec< T >::ptr (size_type *n*) [inline]

Returns:

pointer at position n. Throws `std::out_of_range` if no element exists at n

3.2.3.19 template<class T> const_pointer rptr::Rvec< T >::ptr (size_type *n*) const [inline]

Returns:

pointer at position n. Throws `std::out_of_range` if no element exists at n

3.2.3.20 template<class T> Rvec< T >::size_type rptr::Rvec< T >::set (size_type *pos*, pointer *x*)

Insert a new object into container at given position. If an object existed at *pos* that object is replaced by the new object x. If x is NULL and element at *pos* is defined, then that element is erased and **end()**(p. 10) is returned.

- *pos* Position where the new object will be inserted

  Returns:

  index of inserted object or **end()**(p. 10) if x was NULL.

3.2.3.21 template<class T> size_type rptr::Rvec< T >::size () const [inline]

Returns:

number of elements in container (not counting empty positions)

3.2.3.22 template<class T> void rptr::Rvec< T >::swap (const Rvec< T > & *src*) [inline]

Swap contents of *src* and this container

The documentation for this class was generated from the following file:

- rvec.h

3.3 rptr::Rvec< T >::const_iterator Class Reference

`#include <rvec.h>`

Public Member Functions

- reference **operator** ∗ () const
- pointer **operator** → () const
- pointer **ptr** () const
- bool **operator==** (const **const_iterator** &right) const
- bool **operator!=** (const **const_iterator** &right) const
- bool **operator<** (const **const_iterator** &right) const
- **const_iterator** & **operator++** ()
- **const_iterator operator++** (int)
- **const_iterator** & **operator–** ()
- **const_iterator operator–** (int)

3.3.1 Detailed Description

template<class T> class rptr::Rvec< T >::const_iterator

STL-type Const-iterator implementation

3.3.2 Member Function Documentation

3.3.2.1 template<class T> reference rptr::Rvec< T >::const_iterator::operator ∗ () const `[inline]`

Dereferencing

3.3.2.2 template<class T> bool rptr::Rvec< T >::const_iterator::operator!= (const const_iterator & *right*) const `[inline]`

Inequality

3.3.2.3 template<class T> const_iterator rptr::Rvec< T >::const_iterator::operator++ (int) `[inline]`

Post-increment

3.3.2.4 template<class T> const_iterator& rptr::Rvec< T >::const_iterator::operator++ () `[inline]`

Pre-increment

3.3.2.5 template<class T> const_iterator rptr::Rvec< T >::const_iterator::operator– (int) `[inline]`

Post-decrement

3.3.2.6 template<class T> const_iterator& rptr::Rvec< T >::const_iterator::operator– () `[inline]`

Pre-decrement

3.3.2.7 template<class T> pointer rptr::Rvec< T >::const_iterator::operator → () const `[inline]`

Member-dereferencing

3.3.2.8 template<class T> bool rptr::Rvec< T >::const_iterator::operator< (const const_iterator & *right*) const `[inline]`

Comparison

3.3.2.9 template<class T> bool rptr::Rvec< T >::const_iterator::operator== (const const_iterator & *right*) const `[inline]`

Equality

3.3.2.10 template<class T> pointer rptr::Rvec< T >::const_iterator::ptr () const `[inline]`

Get pointer

The documentation for this class was generated from the following file:

- rvec.h

3.4 rptr::Rvec< T >::iterator Class Reference

`#include <rvec.h>`

Public Member Functions

- reference **operator** ∗ () const
- pointer **operator** → () const
- pointer **ptr** () const
- bool **operator==** (const **iterator** &right) const
- bool **operator!=** (const **iterator** &right) const
- bool **operator<** (const **iterator** &right) const
- **iterator** & **operator++** ()
- **iterator operator++** (int)
- **iterator** & **operator–** ()
- **iterator operator–** (int)

3.4.1 Detailed Description

template<class T> class rptr::Rvec< T >::iterator

STL-type iterator implementation

3.4.2 Member Function Documentation

3.4.2.1 template<class T> reference rptr::Rvec< T >::iterator::operator ∗ () const `[inline]`

Dereferencing

3.4.2.2 template<class T> bool rptr::Rvec< T >::iterator::operator!= (const iterator & *right*) const `[inline]`

Inequality

3.4.2.3 template<class T> iterator rptr::Rvec< T >::iterator::operator++ (int) `[inline]`

Post-increment

3.4.2.4 template<class T> iterator& rptr::Rvec< T >::iterator::operator++ () `[inline]`

Pre-increment

3.4.2.5 template<class T> iterator rptr::Rvec< T >::iterator::operator– (int) `[inline]`

Post-decrement

3.4.2.6 template<class T> iterator& rptr::Rvec< T >::iterator::operator– () `[inline]`

Pre-decrement

3.4.2.7 template<class T> pointer rptr::Rvec< T >::iterator::operator → () const `[inline]`

Member-dereferencing

3.4.2.8 template<class T> bool rptr::Rvec< T >::iterator::operator< (const iterator & *right*) const `[inline]`

Comparison

3.4.2.9 template<class T> bool rptr::Rvec< T >::iterator::operator== (const iterator & *right*) const `[inline]`

Equality

3.4.2.10 template<class T> pointer rptr::Rvec< T >::iterator::ptr () const `[inline]`

Get pointer

The documentation for this class was generated from the following file:

- rvec.h

Index