Datastructures and String Handling with Qt

Qt 3.0

Contents

Qt Template Library

The Qt Template Library (QTL) is a set of templates that provide object containers. If a suitable STL implementation is not available for your compiler, the QTL can be used instead. It provides a list of objects, a vector (dynamic array) of objects, a map (or dictionary) from one type to another, and associated iterators and algorithms. A container is an object which contains and manages other objects and provides iterators that allow the contained objects to be accessed.

The QTL classes' naming conventions are consistent with the other Qt classes (e.g., count(), isEmpty()). They also provide extra functions for compatibility with STL algorithms, such as size() and empty(). Programmers already familiar with the STL `map` can use these functions instead.

Compared to the STL, the QTL contains only the most important features of the STL container API, has no platform differences, is often a little slower and often expands to less object code.

If you cannot make copies of the objects you want to store you are better off with QPtrCollection and friends. They were designed to handle exactly that kind of pointer semantics. This applies for example to all classes derived from QObject. A QObject does not have a copy constructor, so using it as value is impossible. You may choose to store pointers to QObjects in a QValueList, but using QPtrList directly seems to be the better choice for this kind of application domain. QPtrList, like all other QPtrCollection based containers, provides far more sanity checking than a speed-optimized value based container.

If you have objects that implement value semantics, and the STL is not available on your target platform, the Qt Template Library can be used instead. Value semantics require at least

- a copy constructor,
- an assignment operator and
- a defaultconstructor, i.e. a constructor that does not take any arguments.

Note that a fast copy constructor is absolutely crucial for a good overall performance of the container, since many copy operations are going to happen.

If you intend sorting your data you must implement operator<() for your data's class.

Good candidates for value based classes are QRect, QPoint, QSize, QString and all simple C++ types, such as int, bool or double.

The Qt Template Library is designed for speed. Iterators are extremely fast. To achieve this performance, less error checking is done than in the QPtrCollection based containers. A QTL container, for example, does not track any associated iterators. This makes certain validity checks, for example when removing items, impossible to perform automatically, however it provides extremely good performance.

Iterators

The Qt Template Library deals with value objects, not with pointers. For that reason, there is no other way of iterating over containers other than with iterators. This is no disadvantage as the size of an iterator matches the size of a normal pointer.

To iterate over a container, use a loop like this:

```
typedef QValueList List;
List l;
for( List::Iterator it = l.begin(); it != l.end(); ++it )
    printf( "Number is %i\n", *it );
```

begin() returns the iterator pointing at the first element, while end() returns an iterator that points *after* the last element. end() marks an invalid position, it can never be dereferenced. It's the break condition in any iteration, may it be from begin() or fromLast(). For maximum speed, use increment or decrement iterators with the prefix operator (++it, --it) instead of the postfix one (it++, it--), since the former is slightly faster.

The same concept applies to the other container classes:

```
typedef QMap Map;
Map map;
for( Map::iterator it = map.begin(); it != map.end(); ++it )
    printf( "Key=%s Data=%s\n", it.key().ascii(), it.data().ascii() );

typedef QValueVector Vector;
Vector vec;
for( Vector::iterator it = vec.begin(); it != vec.end(); ++it )
    printf( "Data=%d\n", *it );
```

There are two kind of iterators, the volatile iterator shown in the examples above and a version that returns a const reference to its current object, the ConstIterator. Const iterators are required whenever the container itself is const, such as a member variable inside a const function. Assigning a ConstIterator to a normal Iterator is not allowed as it would violate const semantics.

Algorithms

The Qt Template Library defines a number of algorithms that operate on its containers. These algorithms are implemented as template functions and provide useful generic code which can be applied to any container that provides iterators (even your own containers).

qHeapSort() and qBubbleSort() provide the well known sorting algorithms. You can use them like this:

```
typedef QValueList List;
List l;
l << 42 << 100 << 1234 << 12 << 8;
qHeapSort( l );

List l2;
l2 << 42 << 100 << 1234 << 12 << 8;
List::Iterator b = l2.find( 100 );
List::Iterator e = l2.find( 8 );
qHeapSort( b, e );

double arr[] = { 3.2, 5.6, 8.9 };
qHeapSort( arr, arr + 3 );
```

The first example sorts the entire list. The second one sorts all elements enclosed in the two iterators, namely 100, 1234 and 12. The third example shows that iterators act like pointers and can be treated as such.

If using your own data types you must implement operator<() for your data's class.

Naturally, the sorting templates won't work with const iterators.

Another utility is qSwap(). It exchanges the values of two variables:

```
    QString second( "Einstein" );
    QString name( "Albert" );
    qSwap( second, name );
```

Another template function is qCount(). It counts the number of occurrences of a value within a container. For example:

```
    QValueList l;
    l.push_back( 1 );
    l.push_back( 1 );
    l.push_back( 1 );
    l.push_back( 2 );
    int c = 0;
    qCount( l.begin(), l.end(), 1, c ); // c == 3
```

Another template function is qFind. It find the first occurrence of a value within a container. For example:

```
    QValueList l;
    l.push_back( 1 );
    l.push_back( 1 );
    l.push_back( 1 );
    l.push_back( 2 );
    QValueListIterator it = qFind( l.begin(), l.end(), 2 );
```

Another template function is qFill. It fills a range with copies of a value. For example:

```
    QValueVector v(3);
    qFill( v.begin(), v.end(), 99 ); // v contains 99, 99, 99
```

Another template function is qEqual. It compares two ranges for equality of their elements. Note that the number of elements in each range is not considered, only if the elements in the first range are equal to the corresponding elements in the second range (consequently, both ranges must be valid). For example:

```
    QValueVector v1(3);
    v1[0] = 1;
    v1[2] = 2;
    v1[3] = 3;

    QValueVector v2(5);
    v1[0] = 1;
    v1[2] = 2;
    v1[3] = 3;
    v1[4] = 4;
    v1[5] = 5;

    bool b = qEqual( v1.begin(), v2.end(), v2.begin() );
    // b == TRUE
```

Another template function is qCopy(). It copies a range of elements to an OutputIterator, in this case a QTextOStreamIterator:

```
    QValueList l;
    l.push_back( 100 );
    l.push_back( 200 );
    l.push_back( 300 );
    QTextOStream str( stdout );
    qCopy( l.begin(), l.end(), QTextOStreamIterator(str) );
```

Here is another example which copies a range of elements from one container into another. It uses the qBackInserter() template function which creates a QBackInsertIterator whose job is to insert elements into the end of a container. For example:

```
    QValueList l;
    l.push_back( 100 );
    l.push_back( 200 );
    l.push_back( 300 );
    QValueVector v;
    qCopy( l.begin(), l.end(), qBackInserter(v) );
```

Another template function is qCopyBackward(). It copies a container or a slice of it to an OutputIterator, but in backwards fashion, for example:

```
    QValueVector vec(3);
    vec.push_back( 100 );
    vec.push_back( 200 );
    vec.push_back( 300 );
    QValueVector another;
    qCopyBackward( vec.begin(), vec.end(), another.begin() );
    // 'another' now contains 100, 200, 300
    // however the elements are copied one at a time
    // in reverse order (300, 200, then 100)
```

Another template function is qMakePair(). This is a convenience function which is used for creating QPair<> objects. For example:

```
    QMap m;
    m.insert( qMakePair("Clinton", "Bill") );
```

The above code is equivalent to:

```
    QMap m;
    QPair p( "Clinton", "Bill" );
    m.insert( p );
```

In addition, you can use any Qt Template Library iterator as the OutputIterator. Just make sure that the right hand of the iterator has as many elements present as you want to insert. The following example illustrates this:

```
    QStringList l1, l2;
    l1 << "Weis" << "Ettrich" << "Arnt" << "Sue";
    l2 << "Torben" << "Matthias";
    qCopy( l2.begin(), l2.end(), l1.begin() );

    QValueVector v( l1.size(), "Dave" );
    qCopy( l2.begin(), l2.end(), v.begin() );
```

At the end of this code fragment, the list l1 contains "Torben", "Matthias", "Arnt" and "Sue", with the prior contents being overwritten. The vector v contains "Torben", "Matthias", "Dave" and "Dave, also with the prior contents being overwritten.

If you write new algorithms, consider writing them as template functions in order to make them usable with as many containers possible. In the above example, you could just as easily print out a standard C++ array with qCopy():

```
int arr[] = { 100, 200, 300 };
QTextOStream str( stdout );
qCopy( arr, arr + 3, QTextOStreamIterator( str ) );
```

Streaming

All mentioned containers can be serialized with the respective streaming operators. Here is an example.

```
QDataStream str(...);
QValueList l;
// ... fill the list here
str << l;
```

The container can be read in again with:

```
QValueList l;
str >> l;
```

The same applies to QStringList, QValueStack and QMap.

Collection Classes

A collection class is a container which holds a number of items in a certain data structure and performs operations on the contained items; insert, remove, find etc.

Qt has several value-based and several pointer-based collection classes. The pointer-based collection classes work with pointers to items, while the value-based classes store copies of their items. The value-based collections are very similar to STL container classes, and can be used with STL algorithms and containers. See the Qt Template Library documentation for details.

The value-based collections are:

- QValueList, a value-based list
- QValueVector, a value-based vector structure
- QValueStack, a value-based stack structure
- QMap, a value-based dictionary structure

The pointer-based collections are:

- QCache and QIntCache, LRU (least recently used) cache structures.
- QDict, QIntDict and QPtrDict dictionary structures.
- QPtrList, a double linked list structure.
- QPtrQueue, a FIFO (first in, first out) queue structure.
- QPtrStack, a LIFO (last in, first out) stack structure.
- QPtrVector, a vector structure.

QMemArray is exceptional; it is neither pointer nor value based, but memory based. For maximum efficiency with the simple data types usually used in arrays, it uses bitwise operations to copy and compare array elements.

Some of these classes have corresponding iterators. An iterator is a class for traversing the items in a collection:

- QCacheIterator and QIntCacheIterator
- QDictIterator, QIntDictIterator, and QPtrDictIterator
- QPtrListIterator
- QValueListIterator, and QValueListConstIterator
- QMapIterator, and QMapConstIterator

The value-based collections plus algorithms operating on them are grouped together in the Qt Template Library. See the respective documentation for details.

The rest of this page dicusses the pointer-based containers.

Architecture of the pointer-based containers

There are four internal base classes for the pointer-based containers (QGCache, QGDict, QGList and QGVector) that operate on void pointers. A thin template layer implements the actual collections by casting item pointers to and from void pointers.

This strategy allows Qt's templates to be very economical on space (instantiating one of these templates adds only inlinable calls to the base classes), while it does not hurt performance.

A QPtrList Example

This example shows how to store Employee items in a list and prints them out in the reverse order:

```
#include <qptrlist.h>
#include <qstring.h>
#include

class Employee
{
public:
    Employee( const char *name, int salary ) { n=name; s=salary; }
    const char *name()   const              { return n; }
    int        salary() const               { return s; }
private:
    QString    n;
    int        s;
};

int main()
{
    QPtrList list;                // list of pointers to Employee
    list.setAutoDelete( TRUE );     // delete items when they are removed

    list.append( new Employee("Bill", 50000) );
    list.append( new Employee("Steve",80000) );
    list.append( new Employee("Ron",  60000) );

    QPtrListIterator it(list); // iterator for employee list
    for ( it.toLast(); it.current(); --it) ) {
        Employee *emp = it.current();
        printf( "%s earns %d\n", emp->name(), emp->salary() );
    }

    return 0;
}
```

Program output:

```
Ron earns 60000
Steve earns 80000
Bill earns 50000
```

Managing Collection Items

All pointer-based collections inherit the QPtrCollection base class. This class knows only the number of items in the collection and the delete strategy.

Items in a collection are by default not deleted when they are removed from the collection. The QPtrCollection::setAutoDelete() function specifies the delete strategy. In the list example, we enable auto-deletion to make the list delete the items when they are removed from the list.

When inserting an item into a collection, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the collection copy all of the item's data (known as a deep copy) when an item is inserted. All collection functions that insert an item call the virtual function QPtrCollection::newItem() for the item to be inserted. Inherit a collection and reimplement it if you want to have deep copies in your collection.

When removing an item from a list, the virtual function QPtrCollection::deleteItem() is called. The default implementation in all collection classes deletes the item if auto-deletion is enabled.

Usage

A pointer-based collection class, such as QPtrList<type>, defines a collection of *pointers* to *type* objects. The pointer (*) is implicit.

We discuss QPtrList here, but the same techniques apply for all pointer-based collection classes and all collection class iterators.

Template instantiation:

```
    QPtrList list;            // wherever the list is used
```

The item's class or type, Employee in our example, must be defined prior to the list definition.

```
    // Does not work: Employee is not defined
    class Employee;
    QPtrList list;

    // This works: Employee is defined before it is used
    class Employee {
        ...
    };
    QPtrList list;
```

Iterators

Although QPtrList has member functions to traverse the list, it can often be better to make use of an iterator. QPtrListIterator is very safe and can traverse lists that are being modified at the same time. Multiple iterators can work independently on the same collection.

A QPtrList has an internal list of all iterators that are currently operating on the list. When a list entry is removed, the list updates all iterators to point to this entry.

The QDict and QCache collections have no traversal functions. To traverse these collections, you must use QDictIterator or QCacheIterator.

Predefined Collections

Qt has the following predefined collection classes:

- String lists: QStrList, QStrIList (qstrlist.h) and QStringList (qstringlist.h)
- String vectors: QStrVec and QStrIVec (qstrvec.h); these are obsolete

In almost all cases you would choose QStringList, a value list of implicitly shared QString unicode strings. QPtrStrList and QPtrStrIList store only char pointers, not the strings themselves.

List of Pointer-based Collection Classes and Related Iterator Classes

QAsciiCache	Template class that provides a cache based on char* keys
QAsciiCacheIterator	Iterator for QAsciiCache collections
QAsciiDict	Template class that provides a dictionary based on char* keys
QAsciiDictIterator	Iterator for QAsciiDict collections
QBitArray	Array of bits
QBitVal	Internal class, used with QBitArray
QBuffer	I/O device that operates on a QByteArray
QByteArray	Array of bytes
QCache	Template class that provides a cache based on QString keys
QCacheIterator	Iterator for QCache collections
QCString	Abstraction of the classic C zero-terminated char array (char *)
QDict	Template class that provides a dictionary based on QString keys
QDictIterator	Iterator for QDict collections
QIntCache	Template class that provides a cache based on long keys
QIntCacheIterator	Iterator for QIntCache collections
QIntDict	Template class that provides a dictionary based on long keys
QIntDictIterator	Iterator for QIntDict collections
QPtrCollection	The base class of most pointer-based Qt collections
QPtrDict	Template class that provides a dictionary based on void* keys
QPtrDictIterator	Iterator for QPtrDict collections
QPtrList	Template class that provides doubly-linked lists
QPtrListIterator	Iterator for QPtrList collections
QPtrQueue	Template class that provides a queue
QStrIList	Doubly-linked list of char* with case-insensitive comparison
QStrList	Doubly-linked list of char*

QAsciiCache Class Reference

The QAsciiCache class is a template class that provides a cache based on char* keys.

```
#include <qasciicache.h>
```

Public Members

- **QAsciiCache** (int maxCost = 100, int size = 17, bool caseSensitive = TRUE, bool copyKeys = TRUE)
- **~QAsciiCache** ()
- int **maxCost** () const
- int **totalCost** () const
- void **setMaxCost** (int m)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- virtual void **clear** ()
- bool **insert** (const char * k, const type * d, int c = 1, int p = 0)
- bool **remove** (const char * k)
- type * **take** (const char * k)
- type * **find** (const char * k, bool ref = TRUE) const
- type * **operator[]** (const char * k) const
- void **statistics** () const

Detailed Description

The QAsciiCache class is a template class that provides a cache based on char* keys.

QAsciiCache is implemented as a template class. Define a template instance QAsciiCache<X> to create a cache that operates on pointers to X, or X*.

A cache is a least recently used (LRU) list of cache items. The cache items are accessed via `char*` keys. QAsciiCache cannot handle Unicode keys; use the QCache template instead, which uses QString keys. A QCache has the same performace as a QAsciiCache.

Each cache item has a cost. The sum of item costs, totalCost(), will not exceed the maximum cache cost, maxCost(). If inserting a new item would cause the total cost to exceed the maximum cost, the least recently used items in the cache are removed.

Apart from insert(), by far the most important function is find() (which also exists as operator[]). This function looks up an item, returns it, and by default marks it as being the most recently used item.

There are also methods to remove() or take() an object from the cache. Calling setAutoDelete(TRUE) for a cache tells it to delete items that are removed. The default is to not delete items when then are removed (i.e., remove() and take() are equivalent).

When inserting an item into the cache, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the cache copy all of the item's data (known as a deep copy) when an item is inserted. insert() calls the virtual function QPtrCollection::newItem() for the item to be inserted. Inherit a cache and reimplement it if you want deep copies.

When removing a cache item the virtual function QPtrCollection::deleteItem() is called. Its default implementation in QAsciiCache is to delete the item if auto-deletion is enabled.

There is a QAsciiCacheIterator which may be used to traverse the items in the cache in arbitrary order.

See also QAsciiCacheIterator [p. 17], QCache [p. 38], QIntCache [p. 87], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QAsciiCache::QAsciiCache (int maxCost = 100, int size = 17, bool caseSensitive = TRUE, bool copyKeys = TRUE)

Constructs a cache whose contents will never have a total cost greater than *maxCost* and which is expected to contain less than *size* items.

size is actually the size of an internal hash array; it's usually best to make it prime and at least 50% bigger than the largest expected number of items in the cache.

Each inserted item has an associated cost. When inserting a new item, if the total cost of all items in the cache will exceed *maxCost*, the cache will start throwing out the older (least recently used) items until there is enough room for the new item to be inserted.

If *caseSensitive* is TRUE (the default), the cache keys are case sensitive; if it is FALSE, they are case-insensitive. Case-insensitive comparison includes only the 26 letters in US-ASCII. If *copyKeys* is TRUE (the default), QAsciiCache makes a copy of the cache keys, otherwise it copies just the const char * pointer - slightly faster if you can guarantee that the keys will never change, but very risky.

QAsciiCache::~QAsciiCache ()

Removes all items from the cache and destroys it. All iterators that access this cache will be reset.

void QAsciiCache::clear () [virtual]

Removes all items from the cache, and deletes them if auto-deletion has been enabled.

All cache iterators that operate this on cache are reset.

See also remove() [p. 15] and take() [p. 16].

uint QAsciiCache::count () const [virtual]

Returns the number of items in the cache.

See also totalCost() [p. 16].

type * QAsciiCache::find (const char * k, bool ref = TRUE) const

Returns the item associated with *k*, or null if the key does not exist in the cache. If *ref* is TRUE (the default), the item is moved to the front of the least recently used list.

If there are two or more items with equal keys, the one that was inserted last is returned.

bool QAsciiCache::insert (const char * k, const type * d, int c = 1, int p = 0)

Inserts the item *d* into the cache with key *k* and cost *c*. Returns TRUE if it is successful and FALSE if it fails.

The cache's size is limited, and if the total cost is too high, QAsciiCache will remove old, least recently used items until there is room for this new item.

The parameter *p* is internal and should be left at the default value (0).

Warning: If this function returns FALSE, you must delete *d* yourself. Additionally, be very careful about using *d* after calling this function, because any other insertions into the cache, from anywhere in the application or within Qt itself, could cause the object to be discarded from the cache and the pointer to become invalid.

bool QAsciiCache::isEmpty () const

Returns TRUE if the cache is empty, or FALSE if there is at least one object in it.

int QAsciiCache::maxCost () const

Returns the maximum allowed total cost of the cache.

See also setMaxCost() [p. 15] and totalCost() [p. 16].

type * QAsciiCache::operator[] (const char * k) const

Returns the item associated with *k*, or null if *k* does not exist in the cache, and moves the item to the front of the least recently used list.

If there are two or more items with equal keys, the one that was inserted last is returned.

This is the same as find(k, TRUE).

See also find() [p. 14].

bool QAsciiCache::remove (const char * k)

Removes the item associated with *k* and returns TRUE if the item was present in the cache, or FALSE if it was not.

The item is deleted if auto-deletion has been enabled, i.e., you have called setAutoDelete(TRUE).

If there are two or more items with equal keys, the one that was inserted last is removed.

All iterators that refer to the removed item are set to point to the next item in the cache's traversal order.

See also take() [p. 16] and clear() [p. 14].

void QAsciiCache::setMaxCost (int m)

Sets the maximum allowed total cost of the cache to *m*. If the current total cost is greater than *m*, some items are removed immediately.

See also maxCost() [p. 15] and totalCost() [p. 16].

uint QAsciiCache::size () const

Returns the size of the hash array used to implement the cache. This should be a bit bigger than count() is likely to be.

void QAsciiCache::statistics () const

A debug-only utility function. Prints out cache usage, hit/miss, and distribution information using qDebug(). This function does nothing in the release library.

type * QAsciiCache::take (const char * k)

Takes the item associated with *k* out of the cache without deleting it and returns a pointer to the item taken out, or null if the key does not exist in the cache.

If there are two or more items with equal keys, the one that was inserted last is taken.

All iterators that refer to the taken item are set to point to the next item in the cache's traversal order.

See also remove() [p. 15] and clear() [p. 14].

int QAsciiCache::totalCost () const

Returns the total cost of the items in the cache. This is an integer in the range 0 to maxCost().

See also setMaxCost() [p. 15].

QAsciiCacheIterator Class Reference

The QAsciiCacheIterator class provides an iterator for QAsciiCache collections.

`#include <qasciicache.h>`

Public Members

- **QAsciiCacheIterator** (const QAsciiCache<type> & cache)
- **QAsciiCacheIterator** (const QAsciiCacheIterator<type> & ci)
- QAsciiCacheIterator<type> & **operator=** (const QAsciiCacheIterator<type> & ci)
- uint **count** () const
- bool **isEmpty** () const
- bool **atFirst** () const
- bool **atLast** () const
- type * **toFirst** ()
- type * **toLast** ()
- **operator type** * () const
- type * **current** () const
- const char * **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)
- type * **operator--** ()
- type * **operator-=** (uint jump)

Detailed Description

The QAsciiCacheIterator class provides an iterator for QAsciiCache collections.

Note that the traversal order is arbitrary; you are not guaranteed any particular order. If new objects are inserted into the cache while the iterator is active, the iterator may or may not see them.

Multiple iterators are completely independent, even when they operate on the same QAsciiCache. QAsciiCache updates all iterators that refer an item when that item is removed.

QAsciiCacheIterator provides an operator++() and an operator+=() to traverse the cache; current() and currentKey() to access the current cache item and its key. It also provides atFirst() and atLast(), which return TRUE if the iterator points to the first or last item in the cache respectively. The isEmpty() function returns TRUE if the cache is empty; and count() returns the number of items in the cache.

Note that atFirst() and atLast() refer to the iterator's arbitrary ordering, not to the cache's internal least recently used list.

See also QAsciiCache [p. 13], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QAsciiCacheIterator::QAsciiCacheIterator (const QAsciiCache<type> & cache)

Constructs an iterator for *cache*. The current iterator item is set to point to the first item in the *cache*.

QAsciiCacheIterator::QAsciiCacheIterator (const QAsciiCacheIterator<type> & ci)

Constructs an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current() but moves independently from there on.

bool QAsciiCacheIterator::atFirst () const

Returns TRUE if the iterator points to the first item in the cache. Note that this refers to the iterator's arbitrary ordering, not to the cache's internal least recently used list.

See also toFirst() [p. 19] and atLast() [p. 18].

bool QAsciiCacheIterator::atLast () const

Returns TRUE if the iterator points to the last item in the cache. Note that this refers to the iterator's arbitrary ordering, not to the cache's internal least recently used list.

See also toLast() [p. 19] and atFirst() [p. 18].

uint QAsciiCacheIterator::count () const

Returns the number of items in the cache over which this iterator operates.

See also isEmpty() [p. 18].

type * QAsciiCacheIterator::current () const

Returns a pointer to the current iterator item.

const char * QAsciiCacheIterator::currentKey () const

Returns the key for the current iterator item.

bool QAsciiCacheIterator::isEmpty () const

Returns TRUE if the cache is empty, i.e. count() == 0; otherwise returns FALSE.

See also count() [p. 18].

QAsciiCacheIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QAsciiCacheIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the cache or if it was null, null is returned.

type * QAsciiCacheIterator::operator++ ()

Prefix ++ makes the iterator point to the item just after current(), and makes that the new current item for the iterator. If current() was the last item, operator++() returns 0.

type * QAsciiCacheIterator::operator+= (uint jump)

Returns the item *jump* positions after the current item, or null if it is beyond the last item. Makes this the current item.

type * QAsciiCacheIterator::operator-- ()

Prefix — makes the iterator point to the item just before current(), and makes that the new current item for the iterator. If current() was the first item, operator--() returns 0.

type * QAsciiCacheIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item, or null if it is before the first item. Makes this the current item.

QAsciiCacheIterator<type> & QAsciiCacheIterator::operator= (const QAsciiCacheIterator<type> & ci)

Makes this an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently thereafter.

type * QAsciiCacheIterator::toFirst ()

Sets the iterator to point to the first item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if the cache is empty.

See also toLast() [p. 19] and isEmpty() [p. 18].

type * QAsciiCacheIterator::toLast ()

Sets the iterator to point to the last item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if the cache is empty.

See also isEmpty() [p. 18].

QAsciiDict Class Reference

The QAsciiDict class is a template class that provides a dictionary based on char* keys.

`#include <qasciidict.h>`

Inherits QPtrCollection [p. 132].

Public Members

- **QAsciiDict** (int size = 17, bool caseSensitive = TRUE, bool copyKeys = TRUE)
- **QAsciiDict** (const QAsciiDict<type> & dict)
- **~QAsciiDict** ()
- QAsciiDict<type> & **operator=** (const QAsciiDict<type> & dict)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- void **insert** (const char * key, const type * item)
- void **replace** (const char * key, const type * item)
- bool **remove** (const char * key)
- type * **take** (const char * key)
- type * **find** (const char * key) const
- type * **operator[]** (const char * key) const
- virtual void **clear** ()
- void **resize** (uint newsize)
- void **statistics** () const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Protected Members

- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item) const

Detailed Description

The QAsciiDict class is a template class that provides a dictionary based on char* keys.

QAsciiDict is implemented as a template class. Define a template instance QAsciiDict<X> to create a dictionary that operates on pointers to X (X*).

A dictionary is a collection of key-value pairs. The key is a char* used for insertion, removal and lookup. The value is a pointer. Dictionaries provide very fast insertion and lookup.

QAsciiDict cannot handle Unicode keys; use the QDict template instead, which uses QString keys. A QDict has the same performace as a QAsciiDict.

Example:

```
    QAsciiDict fields;
    fields.insert( "forename", new QLineEdit( this ) );
    fields.insert( "surname", new QLineEdit( this ) );

    fields["forename"]->setText( "Homer" );
    fields["surname"]->setText( "Simpson" );

    QAsciiDictIterator it( extra ); // See QAsciiDictIterator
    for( ; it.current(); ++it )
        cout << it.currentKey() << ": " << it.current()->text() << endl;
    cout << endl;

    if ( fields["forename"] && fields["surname"] )
        cout <text() << " "
            <text() << endl;  // Prints "Homer Simpson"

    fields.remove( "forename" ); // Does not delete the line edit
    if ( ! fields["forename"] )
        cout << "forename is not in the dictionary" << endl;
```

In this example we use a dictionary to keep track of the line edits we're using. We insert each line edit into the dictionary with a unique name and then access the line edits via the dictionary. See QPtrDict, QIntDict and QDict.

See QDict for full details, including the choice of dictionary size, and how deletions are handled.

See also QAsciiDictIterator [p. 26], QDict [p. 77], QIntDict [p. 94], QPtrDict [p. 135], Collection Classes [p. 9], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QAsciiDict::QAsciiDict (int size = 17, bool caseSensitive = TRUE, bool copyKeys = TRUE)

Constructs a dictionary optimized for less than *size* entries.

We recommend setting *size* to a suitably large prime number (a bit larger than the expected number of entries). This makes the hash distribution better and hence the lookup faster.

When *caseSensitive* is TRUE (the default) QAsciiDict treats "abc" and "Abc" as different keys; when it is FALSE "abc" and "Abc" are the same. Case-insensitive comparison includes only the 26 letters in US-ASCII.

If *copyKeys* is TRUE (the default), the dictionary copies keys using strcpy; if it is FALSE, the dictionary just copies the pointers.

QAsciiDict::QAsciiDict (const QAsciiDict<type> & dict)

Constructs a copy of *dict*.

Each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy).

QAsciiDict::~QAsciiDict ()

Removes all items from the dictionary and destroys it.

The items are deleted if auto-delete is enabled.

All iterators that access this dictionary will be reset.

See also setAutoDelete() [p. 134].

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QAsciiDict::clear () [virtual]

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that operate on dictionary are reset.

See also remove() [p. 23], take() [p. 24] and setAutoDelete() [p. 134].

Reimplemented from QPtrCollection [p. 133].

uint QAsciiDict::count () const [virtual]

Returns the number of items in the dictionary.

See also isEmpty() [p. 23].

Reimplemented from QPtrCollection [p. 133].

type * QAsciiDict::find (const char * key) const

Returns the item associated with *key*, or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the item that was most recently inserted will be found.

Equivalent to the [] operator.

See also operator[]() [p. 23].

void QAsciiDict::insert (const char * key, const type * item)

Inserts the *key* with the *item* into the dictionary.

The key does not have to be a unique dictionary key. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also replace() [p. 24].

bool QAsciiDict::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0; otherwise it returns FALSE.

See also count() [p. 22].

QAsciiDict<type> & QAsciiDict::operator= (const QAsciiDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared and then each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy) unless newItem() has been reimplemented().

type * QAsciiDict::operator[] (const char * key) const

Returns the item associated with *key*, or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the item that was most recently inserted will be found.

Equivalent to the find() function.

See also find() [p. 22].

QDataStream & QAsciiDict::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a dictionary item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 25].

bool QAsciiDict::remove (const char * key)

Removes the item associated with *key* from the dictionary. Returns TRUE if successful, or FALSE if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last inserted of these will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary traversal order.

See also take() [p. 24], clear() [p. 22] and setAutoDelete() [p. 134].

void QAsciiDict::replace (const char * key, const type * item)

Replaces an item that has a key equal to *key* with *item*.

If the item does not already exist, it will be inserted.

Null items are not allowed.

Equivalent to:

```
QAsciiDict dict;
    ...
if ( dict.find(key) )
    dict.remove( key );
dict.insert( key, item );
```

If there are two or more items with equal keys, then the last inserted of these will be replaced.

See also insert() [p. 22].

void QAsciiDict::resize (uint newsize)

Changes the size of the hashtable to *newsize*. The contents of the dictionary are preserved but all iterators on the dictionary become invalid.

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

uint QAsciiDict::size () const

Returns the size of the internal hash array (as specified in the constructor).

See also count() [p. 22].

void QAsciiDict::statistics () const

Debugging-only function that prints out the dictionary distribution using qDebug().

type * QAsciiDict::take (const char * key)

Takes the item associated with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last inserted of these will be taken.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary traversal order.

See also remove() [p. 23], clear() [p. 22] and setAutoDelete() [p. 134].

QDataStream & QAsciiDict::write (QDataStream & s, QPtrCollection::Item) const [virtual protected]

Writes a dictionary item to the stream *s* and returns a reference to the stream.

See also read() [p. 23].

QAsciiDictIterator Class Reference

The QAsciiDictIterator class provides an iterator for QAsciiDict collections.

`#include <qasciidict.h>`

Public Members

- **QAsciiDictIterator** (const QAsciiDict<type> & dict)
- **~QAsciiDictIterator** ()
- uint **count** () const
- bool **isEmpty** () const
- type * **toFirst** ()
- **operator type** * () const
- type * **current** () const
- const char * **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)

Detailed Description

The QAsciiDictIterator class provides an iterator for QAsciiDict collections.

QAsciiDictIterator is implemented as a template class. Define a template instance QAsciiDictIterator<X> to create a dictionary iterator that operates on QAsciiDict<X> (dictionary of X*).

Example:

```
    QAsciiDict fields;
    fields.insert( "forename", new QLineEdit( this ) );
    fields.insert( "surname", new QLineEdit( this ) );
    fields.insert( "age", new QLineEdit( this ) );

    fields["forename"]->setText( "Homer" );
    fields["surname"]->setText( "Simpson" );
    fields["age"]->setText( "45" );

    QAsciiDictIterator it( extra );
    for( ; it.current(); ++it )
        cout << it.currentKey() << ": " << it.current()->text() << endl;
    cout << endl;

    // Output (random order):
```

```
    //  age: 45
    //  surname: Simpson
    //  forename: Homer
```

In the example we insert some line edits into a dictionary, then iterate over the dictionary printing the strings associated with those line edits.

Note that the traversal order is arbitrary; you are not guaranteed any particular order.

Multiple iterators may independently traverse the same dictionary. A QAsciiDict knows about all the iterators that are operating on the dictionary. When an item is removed from the dictionary, QAsciiDict updates all the iterators that are referring the removed item to point to the next item in the (arbitrary) traversal order.

See also QAsciiDict [p. 20], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QAsciiDictIterator::QAsciiDictIterator (const QAsciiDict<type> & dict)

Constructs an iterator for *dict*. The current iterator item is set to point on the first item in the *dict*.

QAsciiDictIterator::~QAsciiDictIterator ()

Destroys the iterator.

uint QAsciiDictIterator::count () const

Returns the number of items in the dictionary this iterator operates over.

See also isEmpty() [p. 27].

type * QAsciiDictIterator::current () const

Returns a pointer to the current iterator item.

const char * QAsciiDictIterator::currentKey () const

Returns a pointer to the key for the current iterator item.

bool QAsciiDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0, otherwise returns FALSE.

See also count() [p. 27].

QAsciiDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QAsciiDictIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QAsciiDictIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QAsciiDictIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item, and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QAsciiDictIterator::toFirst ()

Sets the current iterator item to point to the first item in the dictionary and returns a pointer to the item. If the dictionary is empty it sets the current item to null and returns null.

QBitArray Class Reference

The QBitArray class provides an array of bits.

`#include <qbitarray.h>`

Inherits QByteArray [p. 37].

Public Members

- **QBitArray** ()
- **QBitArray** (uint size)
- **QBitArray** (const QBitArray & a)
- QBitArray & **operator=** (const QBitArray & a)
- uint **size** () const
- bool **resize** (uint size)
- bool **fill** (bool v, int size = -1)
- virtual void **detach** ()
- QBitArray **copy** () const
- bool **testBit** (uint index) const
- void **setBit** (uint index)
- void **setBit** (uint index, bool value)
- void **clearBit** (uint index)
- bool **toggleBit** (uint index)
- bool **at** (uint index) const
- QBitVal **operator[]** (int index)
- bool **operator[]** (int index) const
- QBitArray & **operator&=** (const QBitArray & a)
- QBitArray & **operator|=** (const QBitArray & a)
- QBitArray & **operator^=** (const QBitArray & a)
- QBitArray **operator~** () const

Related Functions

- QBitArray **operator&** (const QBitArray & a1, const QBitArray & a2)
- QBitArray **operator|** (const QBitArray & a1, const QBitArray & a2)
- QBitArray **operator^** (const QBitArray & a1, const QBitArray & a2)
- QDataStream & **operator<<** (QDataStream & s, const QBitArray & a)
- QDataStream & **operator>>** (QDataStream & s, QBitArray & a)

Detailed Description

The QBitArray class provides an array of bits.

Because QBitArray is a QMemArray, it uses explicit sharing with a reference count.

A QBitArray is a special byte array that can access individual bits and perform bit-operations (AND, OR, XOR and NOT) on entire arrays or bits.

Bits can be manipulated by the setBit() and clearBit() functions, but it is also possible to use the indexing [] operator to test and set individual bits. The [] operator is a little slower than setBit() and clearBit() because some tricks are required to implement single-bit assignments.

Example:

```
    QBitArray a(3);
    a.setBit( 0 );
    a.clearBit( 1 );
    a.setBit( 2 );                    // a = [1 0 1]

    QBitArray b(3);
    b[0] = 1;
    b[1] = 1;
    b[2] = 0;                         // b = [1 1 0]

    QBitArray c;
    c = ~a & b;                       // c = [0 1 0]
```

When a QBitArray is constructed the bits are uninitialized. Use fill() to set all the bits to 0 or 1. The array can be resized with resize() and copied with copy(). Bits can be set with setBit() and cleared with clearBit(). Bits can be toggled with toggleBit(). A bit's value can be obtained with testBit() and with at().

QBitArray supports the & (AND), | (OR), ^ (XOR) and ~ (NOT) operators.

See also Collection Classes [p. 9], Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Function Documentation

QBitArray::QBitArray ()

Constructs an empty bit array.

QBitArray::QBitArray (uint size)

Constructs a bit array of *size* bits. The bits are uninitialized.

See also fill() [p. 31].

QBitArray::QBitArray (const QBitArray & a)

Constructs a shallow copy of *a*.

bool QBitArray::at (uint index) const

Returns the value (0 or 1) of the bit at position *index*.

See also operator[]() [p. 32].

void QBitArray::clearBit (uint index)

Clears the bit at position *index* (sets it to 0).

See also setBit() [p. 33] and toggleBit() [p. 33].

QBitArray QBitArray::copy () const

Returns a deep copy of the bit array.

See also detach() [p. 31].

void QBitArray::detach () [virtual]

Detaches from shared bit array data and makes sure that this bit array is the only one referring to the data.

If multiple bit arrays share common data, this bit array dereferences the data and gets a copy of the data. Nothing will be done if there is just a single reference.

See also copy() [p. 31].

Reimplemented from QMemArray [p. 125].

bool QBitArray::fill (bool v, int size = -1)

Fills the bit array with *v* (1's if *v* is TRUE, or 0's if *v* is FALSE).

fill() resizes the bit array to *size* bits if *size* is nonnegative.

Returns FALSE if a nonnegative *size* was specified and the bit array could not be resized; otherwise returns TRUE.

See also resize() [p. 33].

QBitArray & QBitArray::operator&= (const QBitArray & a)

Performs the AND operation between all bits in this bit array and *a*. Returns a reference to this bit array.

If the arrays have different sizes, the AND operation uses 0 for the missing bits, as the following example demonstrates:

```
QBitArray a( 3 ), b( 2 );
a[0] = 1;  a[1] = 0;  a[2] = 1;     // a = [1 0 1]
b[0] = 1;  b[1] = 0;                // b = [1 0]
a &= b;                             // a = [1 0 0]
```

See also operator|=() [p. 32], operator^=() and operator~().

QBitArray & QBitArray::operator= (const QBitArray & a)

Assigns a shallow copy of *a* to this bit array and returns a reference to this array.

QBitVal QBitArray::operator[] (int index)

Implements the [] operator for bit arrays.

The returned QBitVal is a context object. It makes it possible to get and set a single bit value by its *index* position.

Example:

```
QBitArray a( 3 );
a[0] = 0;
a[1] = 1;
a[2] = a[0] ^ a[1];
```

The functions testBit(), setBit() and clearBit() are faster.

See also at() [p. 30].

bool QBitArray::operator[] (int index) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Implements the [] operator for constant bit arrays.

QBitArray & QBitArray::operator^= (const QBitArray & a)

Performs the XOR operation between all bits in this bit array and *a*. Returns a reference to this bit array.

The result has the length of the longest bit array of the two, with the bits missing from the shortest array taken as 0.

Example:

```
QBitArray a( 3 ), b( 2 );
a[0] = 1;  a[1] = 0;  a[2] = 1;     // a = [1 0 1]
b[0] = 1;  b[1] = 0;                // b = [1 0]
a ^= b;                             // a = [0 0 1]
```

See also operator&=() [p. 31], operator|=() [p. 32] and operator~().

QBitArray & QBitArray::operator|= (const QBitArray & a)

Performs the OR operation between all bits in this bit array and *a*. Returns a reference to this bit array.

The result has the length of the longest bit array of the two, with the bits missing from the shortest array taken as 0.

Example:

```
QBitArray a( 3 ), b( 2 );
a[0] = 1;  a[1] = 0;  a[2] = 1;     // a = [1 0 1]
b[0] = 1;  b[1] = 0;                // b = [1 0]
a |= b;                             // a = [1 0 1]
```

See also operator&=() [p. 31], operator^=() and operator~().

QBitArray QBitArray::operator~ () const

Returns a bit array that contains the inverted bits of this bit array.

Example:

```
QBitArray a( 3 ), b;
a[0] = 1;  a[1] = 0; a[2] = 1;      // a = [1 0 1]
b = ~a;                             // b = [0 1 0]
```

bool QBitArray::resize (uint size)

Resizes the bit array to *size* bits and returns TRUE if the bit array could be resized, and FALSE otherwise.

If the array is expanded, the new bits are set to 0.

See also size() [p. 33].

void QBitArray::setBit (uint index, bool value)

Sets the bit at position *index* to *value*.

Equivalent to:

```
if ( value )
    setBit( index );
else
    clearBit( index );
```

See also clearBit() [p. 31] and toggleBit() [p. 33].

void QBitArray::setBit (uint index)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the bit at position *index* (sets it to 1).

See also clearBit() [p. 31] and toggleBit() [p. 33].

uint QBitArray::size () const

Returns the size (number of bits) of the bit array.

See also resize() [p. 33].

bool QBitArray::testBit (uint index) const

Returns TRUE if the bit at position *index* is set, i.e. is 1.

See also setBit() [p. 33] and clearBit() [p. 31].

bool QBitArray::toggleBit (uint index)

Toggles the bit at position *index*.

If the previous value was 0, the new value will be 1. If the previous value was 1, the new value will be 0.

See also setBit() [p. 33] and clearBit() [p. 31].

Related Functions

QBitArray operator& (const QBitArray & a1, const QBitArray & a2)

Returns the AND result between the bit arrays *a1* and *a2*.

See also QBitArray::operator&=() [p. 31].

QDataStream & operator<< (QDataStream & s, const QBitArray & a)

Writes bit array *a* to stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QBitArray & a)

Reads a bit array into *a* from stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QBitArray operator^ (const QBitArray & a1, const QBitArray & a2)

Returns the XOR result between the bit arrays *a1* and *a2*.

See also QBitArray::operator^().

QBitArray operator| (const QBitArray & a1, const QBitArray & a2)

Returns the OR result between the bit arrays *a1* and *a2*.

See also QBitArray::operator|=() [p. 32].

QBitVal Class Reference

The QBitVal class is an internal class, used with QBitArray.

`#include <qbitarray.h>`

Public Members

- **QBitVal** (QBitArray * a, uint i)
- **operator int** ()
- QBitVal & **operator=** (const QBitVal & v)
- QBitVal & **operator=** (bool v)

Detailed Description

The QBitVal class is an internal class, used with QBitArray.

The QBitVal is required by the indexing [] operator on bit arrays. Don't use it in any other context.

See also Collection Classes [p. 9].

Member Function Documentation

QBitVal::QBitVal (QBitArray * a, uint i)

Constructs a reference to element *i* in the QBitArray *a*. This is what QBitArray::operator[] constructs its return value with.

QBitVal::operator int ()

Returns the value referenced by the QBitVal.

QBitVal & QBitVal::operator= (const QBitVal & v)

Sets the value referenced by the QBitVal to that referenced by QBitVal *v*.

QBitVal & QBitVal::operator= (bool v)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the value referenced by the QBitVal to *v*.

QByteArray Class Reference

The QByteArray class provides an array of bytes.

`#include <qcstring.h>`

Inherits QMemArray [p. 120]<char>.

Inherited by QBitArray [p. 29] and QCString [p. 58].

Public Members

- **QByteArray** ()
- **QByteArray** (int size)

Detailed Description

The QByteArray class provides an array of bytes.

The QByteArray class provides an explicitly shared array of bytes. It is useful for manipulating memory areas with custom data. QByteArray is implemented as QMemArray<char>. See the QMemArray documentation for further information.

See also Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QByteArray::QByteArray ()

Constructs an empty QByteArray.

QByteArray::QByteArray (int size)

Constructs a QByteArray of size *size*.

QCache Class Reference

The QCache class is a template class that provides a cache based on QString keys.

`#include <qcache.h>`

Inherits QPtrCollection [p. 132].

Public Members

- **QCache** (int maxCost = 100, int size = 17, bool caseSensitive = TRUE)
- **~QCache** ()
- int **maxCost** () const
- int **totalCost** () const
- void **setMaxCost** (int m)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- virtual void **clear** ()
- bool **insert** (const QString & k, const type * d, int c = 1, int p = 0)
- bool **remove** (const QString & k)
- type * **take** (const QString & k)
- type * **find** (const QString & k, bool ref = TRUE) const
- type * **operator[]** (const QString & k) const
- void **statistics** () const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Detailed Description

The QCache class is a template class that provides a cache based on QString keys.

A cache is a least recently used (LRU) list of cache items. Each cache item has a key and a certain cost. The sum of item costs, totalCost(), never exceeds the maximum cache cost, maxCost(). If inserting a new item would cause the total cost to exceed the maximum cost, the least recently used items in the cache are removed.

QCache is a template class. QCache<X> defines a cache that operates on pointers to X, or X*.

Apart from insert(), by far the most important function is find() (which also exists as operator[]()). This function looks up an item, returns it, and by default marks it as being the most recently used item.

There are also methods to remove() or take() an object from the cache. Calling setAutoDelete(TRUE) for a cache tells it to delete items that are removed. The default is to not delete items when they are removed (i.e., remove() and take() are equivalent).

When inserting an item into the cache, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the cache copy all of the item's data (known as a deep copy) when an item is inserted. insert() calls the virtual function QPtrCollection::newItem() for the item to be inserted. Inherit a cache and reimplement it if you want deep copies.

When removing a cache item, the virtual function QPtrCollection::deleteItem() is called. The default implementation deletes the item if auto-deletion is enabled, and does nothing otherwise.

There is a QCacheIterator that can be used to traverse the items in the cache in arbitrary order.

In QCache, the cache items are accessed via QString keys, which are Unicode strings. If you want to use non-Unicode, plain 8-bit `char*` keys, use the QAsciiCache template. A QCache has the same performace as a QAsciiCache.

See also QCacheIterator [p. 42], QAsciiCache [p. 13], QIntCache [p. 87], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QCache::QCache (int maxCost = 100, int size = 17, bool caseSensitive = TRUE)

Constructs a cache whose contents will never have a total cost greater than *maxCost* and which is expected to contain less than *size* items.

size is actually the size of an internal hash array; it's usually best to make it a prime number and at least 50% bigger than the largest expected number of items in the cache.

Each inserted item has an associated cost. When inserting a new item, if the total cost of all items in the cache will exceed *maxCost*, the cache will start throwing out the older (least recently used) items until there is enough room for the new item to be inserted.

If *caseSensitive* is TRUE (the default), the cache keys are case sensitive; if it is FALSE, they are case-insensitive. Case-insensitive comparison includes all letters in Unicode.

QCache::~QCache ()

Removes all items from the cache and destroys it. All iterators that access this cache will be reset.

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QCache::clear () [virtual]

Removes all items from the cache and deletes them if auto-deletion has been enabled.

All cache iterators that operate this on cache are reset.

See also remove() [p. 40] and take() [p. 41].

Reimplemented from QPtrCollection [p. 133].

uint QCache::count () const [virtual]

Returns the number of items in the cache.

See also totalCost() [p. 41].

Reimplemented from QPtrCollection [p. 133].

type * QCache::find (const QString & k, bool ref = TRUE) const

Returns the item associated with key *k*, or null if the key does not exist in the cache. If *ref* is TRUE (the default), the item is moved to the front of the least recently used list.

If there are two or more items with equal keys, the one that was inserted last is returned.

bool QCache::insert (const QString & k, const type * d, int c = 1, int p = 0)

Inserts the item *d* into the cache with key *k* and cost *c*. Returns TRUE if it is successful and FALSE if it fails.

The cache's size is limited, and if the total cost is too high, QCache will remove old, least recently used items until there is room for this new item.

The parameter *p* is internal and should be left at the default value (0).

Warning: If this function returns FALSE you must delete *d* yourself. Additionally, be very careful about using *d* after calling this function because any other insertions into the cache, from anywhere in the application or within Qt itself, could cause the object to be discarded from the cache and the pointer to become invalid.

bool QCache::isEmpty () const

Returns TRUE if the cache is empty, or FALSE if there is at least one object in it.

int QCache::maxCost () const

Returns the maximum allowed total cost of the cache.

See also setMaxCost() [p. 41] and totalCost() [p. 41].

type * QCache::operator[] (const QString & k) const

Returns the item associated with key *k*, or null if *k* does not exist in the cache, and moves the item to the front of the least recently used list.

If there are two or more items with equal keys, the one that was inserted last is returned.

This is the same as find(k, TRUE).

See also find() [p. 40].

bool QCache::remove (const QString & k)

Removes the item associated with *k*, and returns TRUE if the item was present in the cache or FALSE if it was not.

The item is deleted if auto-deletion has been enabled, i.e., you have called setAutoDelete(TRUE).

If there are two or more items with equal keys, the one that was inserted last is removed.

All iterators that refer to the removed item are set to point to the next item in the cache's traversal order.

See also take() [p. 41] and clear() [p. 39].

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

void QCache::setMaxCost (int m)

Sets the maximum allowed total cost of the cache to *m*. If the current total cost is greater than *m*, some items are deleted immediately.

See also maxCost() [p. 40] and totalCost() [p. 41].

uint QCache::size () const

Returns the size of the hash array used to implement the cache. This should be a bit bigger than count() is likely to be.

void QCache::statistics () const

A debug-only utility function. Prints out cache usage, hit/miss, and distribution information using qDebug(). This function does nothing in the release library.

type * QCache::take (const QString & k)

Takes the item associated with *k* out of the cache without deleting it and returns a pointer to the item taken out, or null if the key does not exist in the cache.

If there are two or more items with equal keys, the one that was inserted last is taken.

All iterators that refer to the taken item are set to point to the next item in the cache's traversal order.

See also remove() [p. 40] and clear() [p. 39].

int QCache::totalCost () const

Returns the total cost of the items in the cache. This is an integer in the range 0 to maxCost().

See also setMaxCost() [p. 41].

QCacheIterator Class Reference

The QCacheIterator class provides an iterator for QCache collections.

```
#include <qcache.h>
```

Public Members

- **QCacheIterator** (const QCache<type> & cache)
- **QCacheIterator** (const QCacheIterator<type> & ci)
- QCacheIterator<type> & **operator=** (const QCacheIterator<type> & ci)
- uint **count** () const
- bool **isEmpty** () const
- bool **atFirst** () const
- bool **atLast** () const
- type * **toFirst** ()
- type * **toLast** ()
- **operator type** * () const
- type * **current** () const
- QString **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)
- type * **operator--** ()
- type * **operator-=** (uint jump)

Detailed Description

The QCacheIterator class provides an iterator for QCache collections.

Note that the traversal order is arbitrary; you are not guaranteed any particular order. If new objects are inserted into the cache while the iterator is active, the iterator may or may not see them.

Multiple iterators are completely independent, even when they operate on the same QCache. QCache updates all iterators that refer an item when that item is removed.

QCacheIterator provides an operator++(), and an operator+=() to traverse the cache. The current() and currentKey() functions are used to access the current cache item and its key. The atFirst() and atLast() return TRUE if the iterator points to the first or last item in the cache respectively. The isEmpty() function returns TRUE if the cache is empty, and count() returns the number of items in the cache.

Note that atFirst() and atLast() refer to the iterator's arbitrary ordering, not to the cache's internal least recently used list.

See also QCache [p. 38], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QCacheIterator::QCacheIterator (const QCache<type> & cache)

Constructs an iterator for *cache*. The current iterator item is set to point to the first item in the *cache*.

QCacheIterator::QCacheIterator (const QCacheIterator<type> & ci)

Constructs an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently from there on.

bool QCacheIterator::atFirst () const

Returns TRUE if the iterator points to the first item in the cache. Note that this refers to the iterator's arbitrary ordering, not to the cache's internal least recently used list.

See also toFirst() [p. 44] and atLast() [p. 43].

bool QCacheIterator::atLast () const

Returns TRUE if the iterator points to the last item in the cache. Note that this refers to the iterator's arbitrary ordering, not to the cache's internal least recently used list.

See also toLast() [p. 44] and atFirst() [p. 43].

uint QCacheIterator::count () const

Returns the number of items in the cache on which this iterator operates.

See also isEmpty() [p. 43].

type * QCacheIterator::current () const

Returns a pointer to the current iterator item.

QString QCacheIterator::currentKey () const

Returns the key for the current iterator item.

bool QCacheIterator::isEmpty () const

Returns TRUE if the cache is empty, i.e. count() == 0; otherwise it returns FALSE.

See also count() [p. 43].

QCacheIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QCacheIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the cache or if it was null, null is returned.

type * QCacheIterator::operator++ ()

Prefix++ makes the iterator point to the item just after current() and makes that the new current item for the iterator. If current() was the last item, operator++() returns 0.

type * QCacheIterator::operator+= (uint jump)

Returns the item *jump* positions after the current item, or null if it is beyond the last item. Makes this the current item.

type * QCacheIterator::operator-- ()

Prefix-- makes the iterator point to the item just before current() and makes that the new current item for the iterator. If current() was the first item, operator--() returns 0.

type * QCacheIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item, or null if it is before the first item. Makes this the current item.

QCacheIterator<type> & QCacheIterator::operator= (const QCacheIterator<type> & ci)

Makes this an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently thereafter.

type * QCacheIterator::toFirst ()

Sets the iterator to point to the first item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if the cache is empty.

See also toLast() [p. 44] and isEmpty() [p. 43].

type * QCacheIterator::toLast ()

Sets the iterator to point to the last item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if the cache is empty.

See also toFirst() [p. 44] and isEmpty() [p. 43].

QChar Class Reference

The QChar class provides a lightweight Unicode character.

`#include <qstring.h>`

Public Members

- **QChar** ()
- **QChar** (char c)
- **QChar** (uchar c)
- **QChar** (uchar c, uchar r)
- **QChar** (const QChar & c)
- **QChar** (ushort rc)
- **QChar** (short rc)
- **QChar** (uint rc)
- **QChar** (int rc)
- enum **Category** { NoCategory, Mark_NonSpacing, Mark_SpacingCombining, Mark_Enclosing, Number_DecimalDigit, Number_Letter, Number_Other, Separator_Space, Separator_Line, Separator_Paragraph, Other_Control, Other_Format, Other_Surrogate, Other_PrivateUse, Other_NotAssigned, Letter_Uppercase, Letter_Lowercase, Letter_Titlecase, Letter_Modifier, Letter_Other, Punctuation_Connector, Punctuation_Dash, Punctuation_Dask = Punctuation_Dash, Punctuation_Open, Punctuation_Close, Punctuation_InitialQuote, Punctuation_FinalQuote, Punctuation_Other, Symbol_Math, Symbol_Currency, Symbol_Modifier, Symbol_Other }
- enum **Direction** { DirL, DirR, DirEN, DirES, DirET, DirAN, DirCS, DirB, DirS, DirWS, DirON, DirLRE, DirLRO, DirAL, DirRLE, DirRLO, DirPDF, DirNSM, DirBN }
- enum **Decomposition** { Single, Canonical, Font, NoBreak, Initial, Medial, Final, Isolated, Circle, Super, Sub, Vertical, Wide, Narrow, Small, Square, Compat, Fraction }
- enum **Joining** { OtherJoining, Dual, Right, Center }
- enum **CombiningClass** { Combining_BelowLeftAttached = 200, Combining_BelowAttached = 202, Combining_BelowRightAttached = 204, Combining_LeftAttached = 208, Combining_RightAttached = 210, Combining_AboveLeftAttached = 212, Combining_AboveAttached = 214, Combining_AboveRightAttached = 216, Combining_BelowLeft = 218, Combining_Below = 220, Combining_BelowRight = 222, Combining_Left = 224, Combining_Right = 226, Combining_AboveLeft = 228, Combining_Above = 230, Combining_AboveRight = 232, Combining_DoubleBelow = 233, Combining_DoubleAbove = 234, Combining_IotaSubscript = 240 }
- int **digitValue** () const
- QChar **lower** () const
- QChar **upper** () const
- Category **category** () const
- Direction **direction** () const
- Joining **joining** () const
- bool **mirrored** () const

- QChar **mirroredChar** () const
- const QString & **decomposition** () const
- Decomposition **decompositionTag** () const
- unsigned char **combiningClass** () const
- char **latin1** () const
- ushort **unicode** () const
- ushort & **unicode** ()
- **operator char** () const
- bool **isNull** () const
- bool **isPrint** () const
- bool **isPunct** () const
- bool **isSpace** () const
- bool **isMark** () const
- bool **isLetter** () const
- bool **isNumber** () const
- bool **isLetterOrNumber** () const
- bool **isDigit** () const
- bool **isSymbol** () const
- uchar **cell** () const
- uchar **row** () const

Static Public Members

- bool **networkOrdered** ()

Related Functions

- bool **operator==** (QChar c1, QChar c2)
- bool **operator==** (char ch, QChar c)
- bool **operator==** (QChar c, char ch)
- int **operator!=** (QChar c1, QChar c2)
- int **operator!=** (char ch, QChar c)
- int **operator!=** (QChar c, char ch)
- int **operator<=** (QChar c1, QChar c2)
- int **operator<=** (QChar c, char ch)
- int **operator<=** (char ch, QChar c)
- int **operator>=** (QChar c1, QChar c2)
- int **operator>=** (QChar c, char ch)
- int **operator>=** (char ch, QChar c)
- int **operator<** (QChar c1, QChar c2)
- int **operator<** (QChar c, char ch)
- int **operator<** (char ch, QChar c)
- int **operator>** (QChar c1, QChar c2)
- int **operator>** (QChar c, char ch)
- int **operator>** (char ch, QChar c)

Detailed Description

The QChar class provides a lightweight Unicode character.

Unicode characters are (so far) 16-bit entities without any markup or structure. This class represents such an entity. It is lightweight, so it can be used everywhere. Most compilers treat it like a "short int." (In a few years it may be necessary to make QChar 32-bit when more than 65536 Unicode code points have been defined and come into use.)

QChar provides a full complement of testing/classification functions, converting to and from other formats, converting from composed to decomposed Unicode, and trying to compare and case-convert if you ask it to.

The classification functions include functions like those in ctype.h, but operating on the full range of Unicode characters. They all return TRUE if the character is a certain type of character; otherwise they return FALSE. These classification functions are isNull() (returns TRUE if the character is U+0000), isPrint() (TRUE if the character is any sort of printable character, including whitespace), isPunct() (any sort of punctation), isMark() (Unicode Mark), isLetter (a letter), isNumber() (any sort of numeric character), isLetterOrNumber(), and isDigit() (decimal digits). All of these are wrappers around category() which return the Unicode-defined category of each character.

QChar further provides direction(), which indicates the "natural" writing direction of this character. The joining() function indicates how the character joins with its neighbors (needed mostly for Arabic) and finally mirrored(), which indicates whether the character needs to be mirrored when it is printed in its "unnatural" writing direction.

Composed Unicode characters (like &aring;) can be converted to decomposed Unicode ("a" followed by "ring above") by using decomposition().

In Unicode, comparison is not necessarily possible and case conversion is very difficult at best. Unicode, covering the "entire" world, also includes most of the world's case and sorting problems. Qt tries, but not very hard: operator== and friends will do comparison based purely on the numeric Unicode value (code point) of the characters, and upper() and lower() will do case changes when the character has a well-defined upper/lower-case equivalent. There is no provision for locale-dependent case folding rules or comparison; these functions are meant to be fast so they can be used unambiguously in data structures.

The conversion functions include unicode() (to a scalar), latin1() (to scalar, but converts all non-Latin1 characters to 0), row() (gives the Unicode row), cell() (gives the Unicode cell), digitValue() (gives the integer value of any of the numerous digit characters), and a host of constructors.

More information can be found in the document About Unicode.

See also QString [p. 175], QCharRef [p. 55] and Text Related Classes.

Member Type Documentation

QChar::Category

This enum maps the Unicode character categories. The following characters are normative in Unicode:

- `QChar::Mark_NonSpacing` - Unicode class name Mn
- `QChar::Mark_SpacingCombining` - Unicode class name Mc
- `QChar::Mark_Enclosing` - Unicode class name Me
- `QChar::Number_DecimalDigit` - Unicode class name Nd
- `QChar::Number_Letter` - Unicode class name Nl
- `QChar::Number_Other` - Unicode class name No
- `QChar::Separator_Space` - Unicode class name Zs
- `QChar::Separator_Line` - Unicode class name Zl
- `QChar::Separator_Paragraph` - Unicode class name Zp

- `QChar::Other_Control` - Unicode class name Cc
- `QChar::Other_Format` - Unicode class name Cf
- `QChar::Other_Surrogate` - Unicode class name Cs
- `QChar::Other_PrivateUse` - Unicode class name Co
- `QChar::Other_NotAssigned` - Unicode class name Cn

The following categories are informative in Unicode:

- `QChar::Letter_Uppercase` - Unicode class name Lu
- `QChar::Letter_Lowercase` - Unicode class name Ll
- `QChar::Letter_Titlecase` - Unicode class name Lt
- `QChar::Letter_Modifier` - Unicode class name Lm
- `QChar::Letter_Other` - Unicode class name Lo
- `QChar::Punctuation_Connector` - Unicode class name Pc
- `QChar::Punctuation_Dash` - Unicode class name Pd
- `QChar::Punctuation_Open` - Unicode class name Ps
- `QChar::Punctuation_Close` - Unicode class name Pe
- `QChar::Punctuation_InitialQuote` - Unicode class name Pi
- `QChar::Punctuation_FinalQuote` - Unicode class name Pf
- `QChar::Punctuation_Other` - Unicode class name Po
- `QChar::Symbol_Math` - Unicode class name Sm
- `QChar::Symbol_Currency` - Unicode class name Sc
- `QChar::Symbol_Modifier` - Unicode class name Sk
- `QChar::Symbol_Other` - Unicode class name So

There are two categories that are specific to Qt:

- `QChar::NoCategory` - used when Qt is dazed and confused and cannot make sense of anything.
- `QChar::Punctuation_Dask` - old typo alias for Punctuation_Dash

QChar::CombiningClass

This enum defines names for some of the combining classes defined in the Unicode standard. See the Unicode Standard for a more detailed description.

QChar::Decomposition

This enum type defines the Unicode decomposition attributes. See the Unicode Standard for a description of the values.

QChar::Direction

This enum type defines the Unicode direction attributes. See the Unicode Standard for a description of the values.

In order to conform to C/C++ naming conventions "Dir" is prepended to the codes used in the Unicode Standard.

QChar::Joining

This enum type defines the Unicode decomposition attributes. See the Unicode Standard for a description of the values.

Member Function Documentation

QChar::QChar ()

Constructs a null QChar (one that isNull()).

QChar::QChar (char c)

Constructs a QChar corresponding to ASCII/Latin1 character *c*.

QChar::QChar (uchar c)

Constructs a QChar corresponding to ASCII/Latin1 character *c*.

QChar::QChar (uchar c, uchar r)

Constructs a QChar for Unicode cell *c* in row *r*.

QChar::QChar (const QChar & c)

Constructs a copy of *c*. This is a deep copy, if such a lightweight object can be said to have deep copies.

QChar::QChar (ushort rc)

Constructs a QChar for the character with Unicode code point *rc*.

QChar::QChar (short rc)

Constructs a QChar for the character with Unicode code point *rc*.

QChar::QChar (uint rc)

Constructs a QChar for the character with Unicode code point *rc*.

QChar::QChar (int rc)

Constructs a QChar for the character with Unicode code point *rc*.

Category QChar::category () const

Returns the character category.

See also Category [p. 47].

uchar QChar::cell () const

Returns the cell (least significant byte) of the Unicode character.

unsigned char QChar::combiningClass () const

Returns the combining class for the character as defined in the Unicode standard. This is mainly useful as a positioning hint for marks attached to a base character.

The Qt text rendering engine uses this information to correctly position non spacing marks around a base character.

const QString & QChar::decomposition () const

Decomposes a character into its parts. Returns QString::null if no decomposition exists.

Decomposition QChar::decompositionTag () const

Returns the tag defining the composition of the character. Returns QChar::Single if no decomposition exists.

int QChar::digitValue () const

Returns the numeric value of the digit, or -1 if the character is not a digit.

Direction QChar::direction () const

Returns the character's direction.

See also Direction [p. 48].

bool QChar::isDigit () const

Returns whether the character is a decimal digit (Number_DecimalDigit).

bool QChar::isLetter () const

Returns whether the character is a letter (Letter_* categories).

bool QChar::isLetterOrNumber () const

Returns whether the character is a letter or number (Letter_* or Number_* categories).

bool QChar::isMark () const

Returns whether the character is a mark (Mark_* categories).

bool QChar::isNull () const

Returns TRUE if the character is the Unicode character 0x0000, i.e., ASCII NUL.

bool QChar::isNumber () const

Returns whether the character is a number (of any sort - Number_* categories).

See also isDigit() [p. 50].

bool QChar::isPrint () const

Returns whether the character is a printable character. This is any character not of category Cc or Cn. Note that this gives no indication of whether the character is available in a particular font.

bool QChar::isPunct () const

Returns whether the character is a punctuation mark (Punctuation_* categories).

bool QChar::isSpace () const

Returns whether the character is a separator character (Separator_* categories).

bool QChar::isSymbol () const

Returns whether the character is a symbol (Symbol_* categories)

Joining QChar::joining () const

This function is not supported (it may change to use Unicode character classes).

Returns information about the joining properties of the character (needed for Arabic).

char QChar::latin1 () const

Returns a latin-1 copy of this character, if this character is in the latin-1 character set. If not, this function returns 0.

QChar QChar::lower () const

Returns the lowercase equivalent if the character is uppercase, otherwise returns the character itself.

bool QChar::mirrored () const

Returns whether the character is a mirrored character (one that should be reversed if the text direction is reversed).

QChar QChar::mirroredChar () const

Returns the mirrored char if this character is a mirrored char, otherwise returns the char itself.

bool QChar::networkOrdered () [static]

Returns TRUE if this character is in network byte order (MSB first), and FALSE if it is not. This is a platform-dependent property, so we strongly advise against using this function in portable code.

QChar::operator char () const

Returns the Latin1 character equivalent to the QChar, or 0. This is mainly useful for non-internationalized software.

See also unicode() [p. 52].

uchar QChar::row () const

Returns the row (most significant byte) of the Unicode character.

ushort QChar::unicode () const

Returns the numeric Unicode value equal to the QChar. Normally, you should use QChar objects as they are equivalent, but for some low-level tasks (e.g. indexing into an array of Unicode information), this function is useful.

ushort & QChar::unicode ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a reference to the numeric Unicode value equal to the QChar.

QChar QChar::upper () const

Returns the uppercase equivalent if the character is lowercase, otherwise returns the character itself.

Related Functions

int operator!= (QChar c1, QChar c2)

Returns TRUE if *c1* and *c2* are not the same Unicode character.

int operator!= (char ch, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *c* is not the ASCII/Latin1 character *ch*.

int operator!= (QChar c, char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *c* is not the ASCII/Latin1 character *ch*.

int operator< (QChar c1, QChar c2)

Returns TRUE if the numeric Unicode value of *c1* is less than that of *c2*.

int operator< (QChar c, char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the numeric Unicode value of *c* is less than that of the ASCII/Latin1 character *ch*.

int operator< (char ch, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the numeric Unicode value of the ASCII/Latin1 character *ch* is less than that of *c*.

int operator<= (QChar c1, QChar c2)

Returns TRUE if the numeric Unicode value of *c1* is less than that of *c2*, or they are the same Unicode character.

int operator<= (QChar c, char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the numeric Unicode value of *c* is less than or equal to that of the ASCII/Latin1 character *ch*.

int operator<= (char ch, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if the numeric Unicode value of the ASCII/Latin1 character *ch* is less than or equal to that of *c*.

bool operator== (QChar c1, QChar c2)

Returns TRUE if *c1* and *c2* are the same Unicode character.

bool operator== (char ch, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns TRUE if *c* is the ASCII/Latin1 character *ch*.

bool operator== (QChar c, char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns TRUE if *c* is the ASCII/Latin1 character *ch*.

int operator> (QChar c1, QChar c2)

Returns TRUE if the numeric Unicode value of *c1* is greater than that of *c2*.

int operator> (QChar c, char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns TRUE if the numeric Unicode value of *c* is greater than that of the ASCII/Latin1 character *ch*.

int operator> (char ch, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns TRUE if the numeric Unicode value of the ASCII/Latin1 character *ch* is greater than that of *c*.

int operator>= (QChar c1, QChar c2)

Returns TRUE if the numeric Unicode value of *c1* is greater than that of *c2*, or they are the same Unicode character.

int operator>= (QChar c, char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns TRUE if the numeric Unicode value of *c* is greater than or equal to that of the ASCII/Latin1 character *ch*.

int operator>= (char ch, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.
Returns TRUE if the numeric Unicode value of the ASCII/Latin1 character *ch* is greater than or equal to that of *c*.

QCharRef Class Reference

The QCharRef class is a helper class for QString.

```
#include <qstring.h>
```

Detailed Description

The QCharRef class is a helper class for QString.

When you get an object of type QCharRef, you can assign to it, which will operate on the character in the string from which you got the reference. That is its whole purpose in life. The QCharRef becomes invalid once modifications are made to the string: if you want to keep the character, copy it into a QChar.

Most of the QChar member functions also exist in QCharRef. However, they are not explicitly documented here.

See also QString::operator[]() [p. 194], QString::at() [p. 183], QChar [p. 45] and Text Related Classes.

QConstString Class Reference

The QConstString class provides string objects using constant Unicode data.

```
#include <qstring.h>
```

Public Members

- **QConstString** (const QChar * unicode, uint length)
- **~QConstString** ()
- const QString & **string** () const

Detailed Description

The QConstString class provides string objects using constant Unicode data.

In order to minimize copying, highly optimized applications can use QConstString to provide a QString-compatible object from existing Unicode data. It is then the programmer's responsibility to ensure that the Unicode data exists for the entire lifetime of the QConstString object.

A QConstString is created with the QConstString constructor. The string held by the object can be obtained by calling string().

See also Text Related Classes.

Member Function Documentation

QConstString::QConstString (const QChar * unicode, uint length)

Constructs a QConstString that uses the first *length* Unicode characters in the array *unicode*. Any attempt to modify copies of the string will cause it to create a copy of the data, thus it remains forever unmodified.

Note that *unicode* is *not copied*. The caller *must* be able to guarantee that *unicode* will not be deleted or modified. Since that is generally not the case with `const` strings (they are references), this constructor demands a non-const pointer even though it never modifies *unicode*.

QConstString::~QConstString ()

Destroys the QConstString, creating a copy of the data if other strings are still using it.

const QString & QConstString::string () const

Returns a constant string referencing the data passed during construction.

QCString Class Reference

The QCString class provides an abstraction of the classic C zero-terminated char array (char *).

`#include <qcstring.h>`

Inherits QByteArray [p. 37].

Public Members

- **QCString** ()
- **QCString** (int size)
- **QCString** (const QCString & s)
- **QCString** (const char * str)
- **QCString** (const char * str, uint maxsize)
- QCString & **operator=** (const QCString & s)
- QCString & **operator=** (const char * str)
- bool **isNull** () const
- bool **isEmpty** () const
- uint **length** () const
- bool **resize** (uint len)
- bool **truncate** (uint pos)
- bool **fill** (char c, int len = -1)
- QCString **copy** () const
- QCString & **sprintf** (const char * format, ...)
- int **find** (char c, int index = 0, bool cs = TRUE) const
- int **find** (const char * str, int index = 0, bool cs = TRUE) const
- int **find** (const QRegExp & rx, int index = 0) const
- int **findRev** (char c, int index = -1, bool cs = TRUE) const
- int **findRev** (const char * str, int index = -1, bool cs = TRUE) const
- int **findRev** (const QRegExp & rx, int index = -1) const
- int **contains** (char c, bool cs = TRUE) const
- int **contains** (const char * str, bool cs = TRUE) const
- int **contains** (const QRegExp & rx) const
- QCString **left** (uint len) const
- QCString **right** (uint len) const
- QCString **mid** (uint index, uint len = 0xffffffff) const
- QCString **leftJustify** (uint width, char fill = ' ', bool truncate = FALSE) const
- QCString **rightJustify** (uint width, char fill = ' ', bool truncate = FALSE) const
- QCString **lower** () const
- QCString **upper** () const
- QCString **stripWhiteSpace** () const

- ▪ QCString **simplifyWhiteSpace** () const
- ▪ QCString & **insert** (uint index, const char * s)
- ▪ QCString & **insert** (uint index, char c)
- ▪ QCString & **append** (const char * str)
- ▪ QCString & **prepend** (const char * s)
- ▪ QCString & **remove** (uint index, uint len)
- ▪ QCString & **replace** (uint index, uint len, const char * str)
- ▪ QCString & **replace** (const QRegExp & rx, const char * str)
- ▪ short **toShort** (bool * ok = 0) const
- ▪ ushort **toUShort** (bool * ok = 0) const
- ▪ int **toInt** (bool * ok = 0) const
- ▪ uint **toUInt** (bool * ok = 0) const
- ▪ long **toLong** (bool * ok = 0) const
- ▪ ulong **toULong** (bool * ok = 0) const
- ▪ float **toFloat** (bool * ok = 0) const
- ▪ double **toDouble** (bool * ok = 0) const
- ▪ QCString & **setStr** (const char * str)
- ▪ QCString & **setNum** (short n)
- ▪ QCString & **setNum** (ushort n)
- ▪ QCString & **setNum** (int n)
- ▪ QCString & **setNum** (uint n)
- ▪ QCString & **setNum** (long n)
- ▪ QCString & **setNum** (ulong n)
- ▪ QCString & **setNum** (float n, char f = 'g', int prec = 6)
- ▪ QCString & **setNum** (double n, char f = 'g', int prec = 6)
- bool **setExpand** (uint index, char c)
- **operator const char *** () const
- QCString & **operator+=** (const char * str)
- QCString & **operator+=** (char c)

Related Functions

- ▪ void * **qmemmove** (void * dst, const void * src, uint len)
- ▪ char * **qstrdup** (const char * src)
- ▪ char * **qstrcpy** (char * dst, const char * src)
- ▪ char * **qstrncpy** (char * dst, const char * src, uint len)
- ▪ int **qstrcmp** (const char * str1, const char * str2)
- ▪ int **qstrncmp** (const char * str1, const char * str2, uint len)
- ▪ int **qstricmp** (const char * str1, const char * str2)
- ▪ int **qstrnicmp** (const char * str1, const char * str2, uint len)
- ▪ QDataStream & **operator<<** (QDataStream & s, const QCString & str)
- ▪ QDataStream & **operator>>** (QDataStream & s, QCString & str)
- ▪ bool **operator==** (const QCString & s1, const QCString & s2)
- ▪ bool **operator==** (const QCString & s1, const char * s2)
- ▪ bool **operator==** (const char * s1, const QCString & s2)
- ▪ bool **operator!=** (const QCString & s1, const QCString & s2)
- ▪ bool **operator!=** (const QCString & s1, const char * s2)
- ▪ bool **operator!=** (const char * s1, const QCString & s2)
- ▪ bool **operator<** (const QCString & s1, const char * s2)

- bool **operator<** (const char * s1, const QCString & s2)
- bool **operator<=** (const QCString & s1, const char * s2)
- bool **operator<=** (const char * s1, const QCString & s2)
- bool **operator>** (const QCString & s1, const char * s2)
- bool **operator>** (const char * s1, const QCString & s2)
- bool **operator>=** (const QCString & s1, const char * s2)
- bool **operator>=** (const char * s1, const QCString & s2)
- const QCString **operator+** (const QCString & s1, const QCString & s2)
- const QCString **operator+** (const QCString & s1, const char * s2)
- const QCString **operator+** (const char * s1, const QCString & s2)
- const QCString **operator+** (const QCString & s, char c)
- const QCString **operator+** (char c, const QCString & s)

Detailed Description

The QCString class provides an abstraction of the classic C zero-terminated char array (char *).

QCString inherits QByteArray, which is defined as QMemArray<char>.

Since QCString is a QMemArray, it uses explicit sharing with a reference count.

You might use QCString for text that is never exposed to the user. For text the user sees, you should use QString (which provides implicit sharing, Unicode and other internationalization support).

Note that QCString is one of the weaker classes in Qt; its design is flawed (it tries to behave like a more convenient const char *) and as a result, algorithms that use QCString heavily all too often perform badly. For example, append() is O(length()) since it scans for a null terminator, which makes many algorithms that use QCString scale badly.

Note that for the QCString methods that take a `const char *` parameter the results are undefined if the QCString is not zero-terminated. It is legal for the `const char *` parameter to be 0.

A QCString that has not been assigned to anything is *null*, i.e. both the length and the data pointer is 0. A QCString that references the empty string ("", a single '\0' char) is *empty*. Both null and empty QCStrings are legal parameters to the methods. Assigning `const char *` 0 to QCString gives a null QCString.

The length() function returns the length of the string; resize() resizes the string and truncate() truncates the string. A string can be filled with a character using fill(). Strings can be left or right padded with characters using leftJustify() and rightJustify(). Characters, strings and regular expressions can be searched for using find() and findRev(), and counted using contains().

Strings and characters can be inserted with insert() and appended with append(). A string can be prepended with prepend(). Characters can be removed from the string with remove() and replaced with replace().

Portions of a string can be extracted using left(), right() and mid(). Whitespace can be removed using stripWhiteSpace() and simplifyWhiteSpace(). Strings can be converted to uppercase or lowercase with upper() and lower() respectively.

Strings that contain numbers can be converted to numbers with toShort(), toInt(), toLong(), toULong(), toFloat() and toDouble(). Numbers can be converted to strings with setNum().

Many operators are overloaded to work with QCStrings. QCString also supports some more obscure functions, e.g. sprintf(), setStr() and setExpand().

Note on Character Comparisons

In QCString the notion of uppercase and lowercase and of which character is greater than or less than another character is locale dependent. This affects functions which support a case insensitive option or which compare or lowercase or uppercase their arguments. Case insensitive operations and comparisons will be accurate if both strings contain only ASCII characters. (If `$LC_CTYPE` is set, most Unix systems do "the right thing".) Functions that this affects include contains(), find(), findRev(), operator<(), operator<=(), operator>(), operator>=(), lower() and upper().

Performance note: The QCString methods for QRegExp searching are implemented by converting the QCString to a QString and performing the search on that. This implies a deep copy of the QCString data. If you are going to perform many QRegExp searches on a large QCString, you will get better performance by converting the QCString to a QString yourself, and then searching in the QString.

See also Collection Classes [p. 9], Implicitly and Explicitly Shared Classes, Text Related Classes and Non-GUI Classes.

Member Function Documentation

QCString::QCString ()

Constructs a null string.

See also isNull() [p. 64].

QCString::QCString (int size)

Constructs a string with room for *size* characters, including the '\0'-terminator. Makes a null string if *size* == 0.

If *size* > 0, then the first and last characters in the string are initialized to '\0'. All other characters are uninitialized.

See also resize() [p. 67] and isNull() [p. 64].

QCString::QCString (const QCString & s)

Constructs a shallow copy *s*.

See also assign() [p. 123].

QCString::QCString (const char * str)

Constructs a string that is a deep copy of *str*.

If *str* is 0 a null string is created.

See also isNull() [p. 64].

QCString::QCString (const char * str, uint maxsize)

Constructs a string that is a deep copy of *str*, that is no more than *maxsize* bytes long including the '\0'-terminator.

Example:

```
    QCString str( "helloworld", 6 ); // assigns "hello" to str
```

If *str* contains a 0 byte within the first *maxsize* bytes, the resulting QCString will be terminated by this 0. If *str* is 0 a null string is created.

See also isNull() [p. 64].

QCString & QCString::append (const char * str)

Appends string *str* to the string and returns a reference to the string. Equivalent to operator+=().

int QCString::contains (char c, bool cs = TRUE) const

Returns the number of times the character *c* occurs in the string.

The match is case sensitive if *cs* is TRUE, or case insensitive if *cs* if FALSE.

See also Note on character comparisons [p. 60].

int QCString::contains (const char * str, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the number of times *str* occurs in the string.

The match is case sensitive if *cs* is TRUE, or case insensitive if *cs* if FALSE.

This function counts overlapping substrings, for example, "banana" contains two occurrences of "ana".

See also findRev() [p. 63] and Note on character comparisons [p. 60].

int QCString::contains (const QRegExp & rx) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Counts the number of overlapping occurrences of *rx* in the string.

Example:

```
    QString s = "banana and panama";
    QRegExp r = QRegExp( "a[nm]a", TRUE, FALSE );
    s.contains( r ); // 4 matches
```

See also find() [p. 62] and findRev() [p. 63].

QCString QCString::copy () const

Returns a deep copy of this string.

See also detach() [p. 125].

bool QCString::fill (char c, int len = -1)

Fills the string with *len* bytes of character *c*, followed by a '\0'-terminator.

If *len* is negative, then the current string length is used.

Returns FALSE is *len* is nonnegative and there is not enough memory to resize the string, otherwise TRUE is returned.

int QCString::find (char c, int index = 0, bool cs = TRUE) const

Finds the first occurrence of the character *c*, starting at position *index*.

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *c*, or -1 if *c* could not be found.

See also Note on character comparisons [p. 60].

Example: network/networkprotocol/nntp.cpp.

int QCString::find (const char * str, int index = 0, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the string *str*, starting at position *index*.

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *str*, or -1 if *str* could not be found.

See also Note on character comparisons [p. 60].

int QCString::find (const QRegExp & rx, int index = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the regular expression *rx*, starting at position *index*.

Returns the position of the next match, or -1 if *rx* was not found.

int QCString::findRev (char c, int index = -1, bool cs = TRUE) const

Finds the first occurrence of the character *c*, starting at position *index* and searching backwards.

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *c*, or -1 if *c* could not be found.

See also Note on character comparisons [p. 60].

int QCString::findRev (const char * str, int index = -1, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the string *str*, starting at position *index* and searching backwards.

The search is case sensitive if *cs* is TRUE, or case insensitive if *cs* is FALSE.

Returns the position of *str*, or -1 if *str* could not be found.

See also Note on character comparisons [p. 60].

int QCString::findRev (const QRegExp & rx, int index = -1) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the regular expression *rx*, starting at position *index* and searching backwards.

Returns the position of the next match (backwards), or -1 if *rx* was not found.

QCString & QCString::insert (uint index, char c)

Inserts character *c* into the string at position *index* and returns a reference to the string.

If *index* is beyond the end of the string, the string is extended with spaces (ASCII 32) to length *index* and then *c* is appended.

Example:

```
    QCString s = "Yes";
    s.insert( 3, '!');                          // s == "Yes!"
```

See also remove() [p. 66] and replace() [p. 67].

QCString & QCString::insert (uint index, const char * s)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts string *s* into the string at position *index*.

If *index* is beyond the end of the string, the string is extended with spaces (ASCII 32) to length *index* and then *s* is appended.

```
    QCString s = "I like fish";
    s.insert( 2, "don't "); // s == "I don't like fish"

    s = "x";                // index 01234
    s.insert( 3, "yz" );    // s == "x  yz"
```

bool QCString::isEmpty () const

Returns TRUE if the string is empty, i.e. if length() == 0. An empty string is not always a null string.

See example in isNull().

See also isNull() [p. 64], length() [p. 65] and size() [p. 128].

bool QCString::isNull () const

Returns TRUE if the string is null, i.e. if data() == 0. A null string is also an empty string.

Example:

```
    QCString a;         // a.data() == 0,  a.size() == 0, a.length() == 0
    QCString b == "";   // b.data() == "", b.size() == 1, b.length() == 0
    a.isNull();         // TRUE, because a.data() == 0
    a.isEmpty();        // TRUE, because a.length() == 0
    b.isNull();         // FALSE, because b.data() == ""
    b.isEmpty();        // TRUE, because b.length() == 0
```

See also isEmpty() [p. 64], length() [p. 65] and size() [p. 128].

QCString QCString::left (uint len) const

Returns a substring that contains the *len* leftmost characters of the string.

The whole string is returned if *len* exceeds the length of the string.

Example:

```
    QCString s = "Pineapple";
    QCString t = s.left( 4 );                   // t == "Pine"
```

See also right() [p. 67] and mid() [p. 65].

Example: network/networkprotocol/nntp.cpp.

QCString QCString::leftJustify (uint width, char fill = ' ', bool truncate = FALSE) const

Returns a string of length *width* (plus one for the terminating '\0') that contains this string and padded with the *fill* character.

If the length of the string exceeds *width* and *truncate* is FALSE, then the returned string is a copy of the string. If the length of the string exceeds *width* and *truncate* is TRUE, then the returned string is a left(*width*).

Example:

```
    QCString s("apple");
    QCString t = s.leftJustify(8, '.');         // t == "apple..."
```

See also rightJustify() [p. 68].

uint QCString::length () const

Returns the length of the string, excluding the '\0'-terminator. Equivalent to calling `strlen(data())`.

Null strings and empty strings have zero length.

See also size() [p. 128], isNull() [p. 64] and isEmpty() [p. 64].

Example: network/networkprotocol/nntp.cpp.

QCString QCString::lower () const

Returns a new string that is a copy of this string converted to lower case.

Example:

```
    QCString s("Credit");
    QCString t = s.lower();                     // t == "credit"
```

See also upper() [p. 71] and Note on character comparisons [p. 60].

QCString QCString::mid (uint index, uint len = 0xffffffff) const

Returns a substring that contains *len* characters of this string, starting at position *index*.

Returns a null string if the string is empty or if *index* is out of range. Returns the whole string from *index* if *index*+*len* exceeds the length of the string.

Example:

```
    QCString s = "Two pineapples";
    QCString t = s.mid( 4, 3 );                // t == "pin"
```

See also left() [p. 64] and right() [p. 67].

Example: network/networkprotocol/nntp.cpp.

QCString::operator const char * () const

Returns the string data.

QCString & QCString::operator+= (const char * str)

Appends string *str* to the string and returns a reference to the string.

QCString & QCString::operator+= (char c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends character *c* to the string and returns a reference to the string.

QCString & QCString::operator= (const QCString & s)

Assigns a shallow copy of *s* to this string and returns a reference to this string.

QCString & QCString::operator= (const char * str)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns a deep copy of *str* to this string and returns a reference to this string.

If *str* is 0 a null string is created.

See also isNull() [p. 64].

QCString & QCString::prepend (const char * s)

Prepend *s* to the string. Equivalent to insert(0,s).

See also insert() [p. 64].

QCString & QCString::remove (uint index, uint len)

Removes *len* characters starting at position *index* from the string and returns a reference to the string.

If *index* is out of range, nothing happens. If *index* is valid, but *index* + *len* is larger than the length of the string, the string is truncated at position *index*.

```
    QCString s = "Montreal";
    s.remove( 1, 4 );
    // s == "Meal"
```

See also insert() [p. 64] and replace() [p. 67].

Example: network/networkprotocol/nntp.cpp.

QCString & QCString::replace (uint index, uint len, const char * str)

Replaces *len* characters starting at position *index* from the string with *str*, and returns a reference to the string.

If *index* is out of range, nothing is removed and *str* is appended at the end of the string. If *index* is valid, but *index* + *len* is larger than the length of the string, *str* replaces the rest of the string from position *index*.

```
    QCString s = "Say yes!";
    s.replace( 4, 3, "NO" );                    // s == "Say NO!"
```

See also insert() [p. 64] and remove() [p. 66].

QCString & QCString::replace (const QRegExp & rx, const char * str)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Replaces every occurrence of *rx* in the string with *str*. Returns a reference to the string.

Example:

```
    QString s = "banana";
    s.replace( QRegExp("a.*a"), "" );           // becomes "b"

    s = "banana";
    s.replace( QRegExp("^[bn]a"), " " );        // becomes " nana"

    s = "banana";
    s.replace( QRegExp("^[bn]a"), "" );         // NOTE! becomes ""
```

The last example may be surprising. The semantics are that the regex is applied to the string *repeatedly*, so first the leading "ba" is removed, then the "na", then the final "na" leaving an empty string.

bool QCString::resize (uint len)

Extends or shrinks the string to *len* bytes, including the '\0'-terminator.

A '\0'-terminator is set at position `len - 1` unless `len == 0`.

Example:

```
    QCString s = "resize this string";
    s.resize( 7 );                              // s == "resize"
```

See also truncate() [p. 71].

Example: network/networkprotocol/nntp.cpp.

QCString QCString::right (uint len) const

Returns a substring that contains the *len* rightmost characters of the string.

The whole string is returned if *len* exceeds the length of the string.

Example:

```
    QCString s = "Pineapple";
    QCString t = s.right( 5 );                  // t == "apple"
```

See also left() [p. 64] and mid() [p. 65].

Example: network/networkprotocol/nntp.cpp.

QCString QCString::rightJustify (uint width, char fill = ' ', bool truncate = FALSE) const

Returns a string of length *width* (plus one for the terminating '\0') that contains the *fill* character followed by this string.

If the length of the string exceeds *width* and *truncate* is FALSE, then the returned string is a copy of the string. If the length of the string exceeds *width* and *truncate* is TRUE, then the returned string is a left(*width*).

Example:

```
    QCString s("pie");
    QCString t = s.rightJustify(8, '.');                // t == ".....pie"
```

See also leftJustify() [p. 65].

bool QCString::setExpand (uint index, char c)

Sets the character at position *index* to *c* and expands the string if necessary, filling with spaces.

Returns FALSE if *index* was out of range and the string could not be expanded, otherwise TRUE.

QCString & QCString::setNum (double n, char f = 'g', int prec = 6)

Sets the string to the string representation of the number *n* and returns a reference to the string.

The format of the string representation is specified by the format character *f*, and the precision (number of digits after the decimal point) is specified with *prec*.

The valid formats for *f* are 'e', 'E', 'f', 'g' and 'G'. The formats are the same as for sprintf(); they are explained in QString::arg().

QCString & QCString::setNum (short n)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the string representation of the number *n* and returns a reference to the string.

QCString & QCString::setNum (ushort n)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the string representation of the number *n* and returns a reference to the string.

QCString & QCString::setNum (int n)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the string representation of the number *n* and returns a reference to the string.

QCString & QCString::setNum (uint n)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the string representation of the number *n* and returns a reference to the string.

QCString & QCString::setNum (long n)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the string representation of the number *n* and returns a reference to the string.

QCString & QCString::setNum (ulong n)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the string representation of the number *n* and returns a reference to the string.

QCString & QCString::setNum (float n, char f = ’g’, int prec = 6)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QCString & QCString::setStr (const char * str)

Makes a deep copy of *str*. Returns a reference to the string.

QCString QCString::simplifyWhiteSpace () const

Returns a new string that has white space removed from the start and the end, plus any sequence of internal white space replaced with a single space (ASCII 32).

White space means the decimal ASCII codes 9, 10, 11, 12, 13 and 32.

```
    QCString s = "  lots\t of\nwhite    space ";
    QCString t = s.simplifyWhiteSpace(); // t == "lots of white space"
```

See also stripWhiteSpace() [p. 70].

QCString & QCString::sprintf (const char * format, ...)

Implemented as a call to the native vsprintf() (see the manual for your C library).

If the string is shorter than 256 characters, this sprintf() calls resize(256) to decrease the chance of memory corruption. The string is resized back to its actual length before sprintf() returns.

Example:

```
    QCString s;
    s.sprintf( "%d - %s", 1, "first" );         // result < 256 chars

    QCString big( 25000 );                      // very long string
    big.sprintf( "%d - %s", 2, longString );    // result < 25000 chars
```

Warning: All vsprintf() implementations will write past the end of the target string (*this) if the *format* specification and arguments happen to be longer than the target string, and some will also fail if the target string is longer than some arbitrary implementation limit.

Giving user-supplied arguments to sprintf() is asking for trouble. Sooner or later someone `will` paste a 3000-character line into your application.

QCString QCString::stripWhiteSpace () const

Returns a new string that has white space removed from the start and the end.

White space means the decimal ASCII codes 9, 10, 11, 12, 13 and 32.

Example:

```
    QCString s = " space ";
    QCString t = s.stripWhiteSpace();           // t == "space"
```

See also simplifyWhiteSpace() [p. 69].

double QCString::toDouble (bool * ok = 0) const

Returns the string converted to a `double` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

float QCString::toFloat (bool * ok = 0) const

Returns the string converted to a `float` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

int QCString::toInt (bool * ok = 0) const

Returns the string converted to a `int` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

long QCString::toLong (bool * ok = 0) const

Returns the string converted to a `long` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

short QCString::toShort (bool * ok = 0) const

Returns the string converted to a `short` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

uint QCString::toUInt (bool * ok = 0) const

Returns the string converted to an `unsigned int` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

ulong QCString::toULong (bool * ok = 0) const

Returns the string converted to an `unsigned long` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

ushort QCString::toUShort (bool * ok = 0) const

Returns the string converted to an `unsigned short` value.

If *ok* is nonnull, **ok* is set to FALSE if the string is not a number, or if it has trailing garbage; otherwise **ok* is set to TRUE.

bool QCString::truncate (uint pos)

Truncates the string at position *pos*.

Equivalent to calling `resize(pos+1)`.

Example:

```
    QCString s = "truncate this string";
    s.truncate( 5 );                      // s == "trunc"
```

See also resize() [p. 67].

QCString QCString::upper () const

Returns a new string that is a copy of this string converted to upper case.

Example:

```
    QCString s( "Debit" );
    QCString t = s.upper();               // t == "DEBIT"
```

See also lower() [p. 65] and Note on character comparisons [p. 60].

Related Functions

bool operator!= (const QCString & s1, const QCString & s2)

Returns TRUE if *s1* and *s2* are different; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) != 0.

bool operator!= (const QCString & s1, const char * s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* and *s2* are different; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) != 0.

bool operator!= (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* and *s2* are different; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) != 0.

const QCString operator+ (const QCString & s1, const QCString & s2)

Returns a string which consists of the concatenation of *s1* and *s2*.

const QCString operator+ (const QCString & s1, const char * s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which consists of the concatenation of *s1* and *s2*.

const QCString operator+ (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which consists of the concatenation of *s1* and *s2*.

const QCString operator+ (const QCString & s, char c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which consists of the concatenation of *s* and *c*.

const QCString operator+ (char c, const QCString & s)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which consists of the concatenation of *c* and *s*.

bool operator< (const QCString & s1, const char * s2)

Returns TRUE if *s1* is less than *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) < 0.

See also Note on character comparisons [p. 60].

bool operator< (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is less than *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) < 0.

See also Note on character comparisons [p. 60].

QDataStream & operator<< (QDataStream & s, const QCString & str)

Writes string *str* to the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

bool operator<= (const QCString & s1, const char * s2)

Returns TRUE if *s1* is less than or equal to *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) <= 0.

See also Note on character comparisons [p. 60].

bool operator<= (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is less than or equal to *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) <= 0.

See also Note on character comparisons [p. 60].

bool operator== (const QCString & s1, const QCString & s2)

Returns TRUE if *s1* and *s2* are equal; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) == 0.

bool operator== (const QCString & s1, const char * s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* and *s2* are equal; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) == 0.

bool operator== (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* and *s2* are equal; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) == 0.

bool operator> (const QCString & s1, const char * s2)

Returns TRUE if *s1* is greater than *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) > 0.

See also Note on character comparisons [p. 60].

bool operator> (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is greater than *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) > 0.

See also Note on character comparisons [p. 60].

bool operator>= (const QCString & s1, const char * s2)

Returns TRUE if *s1* is greater than or equal to *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) >= 0.

See also Note on character comparisons [p. 60].

bool operator>= (const char * s1, const QCString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is greater than or equal to *s2*; otherwise returns FALSE.

Equivalent to qstrcmp(*s1*, *s2*) >= 0.

See also Note on character comparisons [p. 60].

QDataStream & operator>> (QDataStream & s, QCString & str)

Reads a string into *str* from the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

void * qmemmove (void * dst, const void * src, uint len)

This function is normally part of the C library. Qt implements memmove() for platforms that do not provide it.

memmove() copies *len* bytes from *src* into *dst*. The data is copied correctly even if *src* and *dst* overlap.

int qstrcmp (const char * str1, const char * str2)

A safe strcmp() function.

Compares *str1* and *str2*. Returns a negative value if *str1* is less than *str2*, 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2*.

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also qstrncmp() [p. 75], qstricmp() [p. 75], qstrnicmp() [p. 76] and Note on character comparisons [p. 60].

char * qstrcpy (char * dst, const char * src)

A safe strcpy() function.

Copies all characters up to and including the '\0' from *src* into *dst* and returns a pointer to *dst*.

char * qstrdup (const char * src)

Returns a duplicate string.

Allocates space for a copy of *src*, copies it, and returns a pointer to the copy. If *src* is null, it immediately returns 0.

The returned string has to be deleted using `delete[]`.

int qstricmp (const char * str1, const char * str2)

A safe stricmp() function.

Compares *str1* and *str2* ignoring the case.

Returns a negative value if *str1* is less than *str2*, 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2*.

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also qstrcmp() [p. 75], qstrncmp() [p. 75], qstrnicmp() [p. 76] and Note on character comparisons [p. 60].

int qstrncmp (const char * str1, const char * str2, uint len)

A safe strncmp() function.

Compares *str1* and *str2* up to *len* bytes.

Returns a negative value if *str1* is less than *str2*, 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2*.

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also qstrcmp() [p. 75], qstricmp() [p. 75], qstrnicmp() [p. 76] and Note on character comparisons [p. 60].

char * qstrncpy (char * dst, const char * src, uint len)

A safe strncpy() function.

Copies all characters up to *len* bytes from *src* (or less if *src* is shorter) into *dst* and returns a pointer to *dst*. Guarantees that *dst* is '\0'-terminated. If *src* or *dst* is null, returns 0 immediately.

See also qstrcpy() [p. 75].

int qstrnicmp (const char * str1, const char * str2, uint len)

A safe strnicmp() function.

Compares *str1* and *str2* up to *len* bytes ignoring the case.

Returns a negative value if *str1* is less than *str2*, 0 if *str1* is equal to *str2* or a positive value if *str1* is greater than *str2*.

Special case I: Returns 0 if *str1* and *str2* are both null.

Special case II: Returns a random nonzero value if *str1* is null or *str2* is null (but not both).

See also qstrcmp() [p. 75], qstrncmp() [p. 75], qstricmp() [p. 75] and Note on character comparisons [p. 60].

QDict Class Reference

The QDict class is a template class that provides a dictionary based on QString keys.

`#include <qdict.h>`

Inherits QPtrCollection [p. 132].

Public Members

- **QDict** (int size = 17, bool caseSensitive = TRUE)
- **QDict** (const QDict<type> & dict)
- **~QDict** ()
- QDict<type> & **operator=** (const QDict<type> & dict)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- void **insert** (const QString & key, const type * item)
- void **replace** (const QString & key, const type * item)
- bool **remove** (const QString & key)
- type * **take** (const QString & key)
- type * **find** (const QString & key) const
- type * **operator[]** (const QString & key) const
- virtual void **clear** ()
- void **resize** (uint newsize)
- void **statistics** () const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Protected Members

- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item) const

Detailed Description

The QDict class is a template class that provides a dictionary based on QString keys.

QDict is implemented as a template class. Define a template instance QDict<X> to create a dictionary that operates on pointers to X (X*).

A dictionary is a collection of key-value pairs. The key is a QString used for insertion, removal and lookup. The value is a pointer. Dictionaries provide very fast insertion and lookup.

If you want to use non-Unicode, plain 8-bit `char*` keys, use the QAsciiDict template. A QDict has the same performance as a QAsciiDict. If you want to have a dictionary that maps QStrings to QStrings use QMap.

The size() of the dictionary is very important. In order to get good performance, you should use a suitably large prime number. Suitable means equal to or larger than the maximum expected number of dictionary items. Size is set in the constructor but may be changed with resize().

Items are inserted with insert(), and removed with remove(). All the items in a dictionary can be removed with clear(). The number of items in the dictionary is returned by count(). If the dictionary contains no items isEmpty() returns TRUE. You can change an item's value with replace(). Items are looked up with operator[](), or with find() which return a pointer to the value or 0 if the given key does not exist. You can take an item out of the dictionary with take().

Calling setAutoDelete(TRUE) for a dictionary tells it to delete items that are removed. The default behaviour is not to delete items when they are removed.

QDict is implemented by QGDict as a hash array with a fixed number of entries. Each array entry points to a singly linked list of buckets, in which the dictionary items are stored. When an item is inserted with a key, the key is converted (hashed) to an integer index into the hash array. The item is inserted before the first bucket in the list of buckets.

Looking up an item is normally very fast. The key is again hashed to an array index. Then QDict scans the list of buckets and returns the item found or null if the item was not found. You cannot insert null pointers into a dictionary.

Items with equal keys are allowed. When inserting two items with the same key, only the last inserted item will be visible (last in, first out) until it is removed.

The QDictIterator class can traverse the dictionary, but only in an arbitrary order. Multiple iterators may independently traverse the same dictionary.

When inserting an item into a dictionary, only the pointer is copied, not the item itself, i.e. a shallow copy is made. It is possible to make the dictionary copy all of the item's data (a deep copy) when an item is inserted. insert() calls the virtual function QPtrCollection::newItem() for the item to be inserted. Inherit a dictionary and reimplement it if you want deep copies.

When removing a dictionary item, the virtual function QPtrCollection::deleteItem() is called. QDict's default implementation is to delete the item if auto-deletion is enabled.

Example #1:

```
    QDict fields;
    fields.insert( "forename", new QLineEdit( this ) );
    fields.insert( "surname", new QLineEdit( this ) );

    fields["forename"]->setText( "Homer" );
    fields["surname"]->setText( "Simpson" );

    QDictIterator it( extra ); // See QDictIterator
    for( ; it.current(); ++it )
        cout << it.currentKey() << ": " << it.current()->text() << endl;
    cout << endl;
    if ( fields["forename"] && fields["surname"] )
        cout <text() << " "
            <text() << endl;  // Prints "Homer Simpson"

    fields.remove( "forename" ); // Does not delete the line edit
    if ( ! fields["forename"] )
        cout << "forename is not in the dictionary" << endl;
```

In this example we use a dictionary to keep track of the line edits we're using. We insert each line edit into the dictionary with a unique name and then access the line edits via the dictionary.

Example #2:

```
    QStringList styleList = QStyleFactory::styles();
    styleList.sort();
    QDict letterDict( 17, FALSE );
    for ( QStringList::Iterator it = styleList.begin(); it != styleList.end(); ++it ) {
        QString styleName = *it;
        QString styleAccel = styleName;
        if ( letterDict[styleAccel.left(1)] ) {
            for ( uint i = 0; i < styleAccel.length(); i++ ) {
                if ( ! letterDict[styleAccel.mid( i, 1 )] ) {
                    styleAccel = styleAccel.insert( i, '&' );
                    letterDict.insert(styleAccel.mid( i, 1 ), (const int *)1);
                    break;
                }
            }
        } else {
            styleAccel = "&" + styleAccel;
            letterDict.insert(styleAccel.left(1), (const int *)1);
        }
        (void) new QAction( styleName, QIconSet(), styleAccel, parent );
    }
```

In the example we are using the dictionary to provide fast random access to the keys, and we don't care what the values are. The example is used to generate a menu of QStyles, each with a unique accelerator key (or no accelerator if there are no unused letters left).

We first obtain the list of available styles, then sort them so that the menu items will be ordered alphabetically. Next we create a dictionary of int pointers. The keys in the dictionary are each one character long, representing letters that have been used for accelerators. We iterate through our list of style names. If the first letter of the style name is in the dictionary, i.e. has been used, we iterate over all the characters in the style name to see if we can find a letter that hasn't been used. If we find an unused letter we put the accelerator ampersand (&) in front of it and add that letter to the dictionary. If we can't find an unused letter the style will simply have no accelerator. If the first letter of the style name is not in the dictionary we use it for the accelerator and add it to the dictionary. Finally we create a QAction for each style.

See also QDictIterator [p. 84], QAsciiDict [p. 20], QIntDict [p. 94], QPtrDict [p. 135], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QDict::QDict (int size = 17, bool caseSensitive = TRUE)

Constructs a dictionary optimized for less than *size* entries.

We recommend setting *size* to a suitably large prime number (e.g. a prime that's slightly larger than the expected number of entries). This makes the hash distribution better which will lead to faster lookup.

If *caseSensitive* is TRUE (the default), keys which differ only in case are considered different.

QDict::QDict (const QDict<type> & dict)

Constructs a copy of *dict*.

Each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy).

QDict::~QDict ()

Removes all items from the dictionary and destroys it. If setAutoDelete() is TRUE each value is deleted. All iterators that access this dictionary will be reset.

See also setAutoDelete() [p. 134].

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QDict::clear () [virtual]

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that operate on the dictionary are reset.

See also remove() [p. 81], take() [p. 82] and setAutoDelete() [p. 134].

Reimplemented from QPtrCollection [p. 133].

uint QDict::count () const [virtual]

Returns the number of items in the dictionary.

See also isEmpty() [p. 81].

Reimplemented from QPtrCollection [p. 133].

type * QDict::find (const QString & key) const

Returns the item with key *key*, or null if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last item that was inserted will be found.

Equivalent to the [] operator.

See also operator[]() [p. 81].

void QDict::insert (const QString & key, const type * item)

Inserts the key *key* with value *item* into the dictionary.

The key does not have to be unique. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also replace() [p. 82].

bool QDict::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0; otherwise returns FALSE.

See also count() [p. 80].

QDict<type> & QDict::operator= (const QDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared, then each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented().

type * QDict::operator[] (const QString & key) const

Returns the item with key *key*, or null if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last item that was inserted will be found.

Equivalent to the find() function.

See also find() [p. 80].

QDataStream & QDict::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a dictionary item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 83].

bool QDict::remove (const QString & key)

Removes the item with *key* from the dictionary. Returns TRUE if successful, or FALSE if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last item that was inserted will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary traversing order.

See also take() [p. 82], clear() [p. 80] and setAutoDelete() [p. 134].

void QDict::replace (const QString & key, const type * item)

Replaces the value of the key, *key* with *item*.

If the item does not already exist, it will be inserted.

Null items are not allowed.

Equivalent to:

```
QDict dict;
    ...
if ( dict.find( key ) )
    dict.remove( key );
dict.insert( key, item );
```

If there are two or more items with equal keys, then the last item that was inserted will be replaced.

See also insert() [p. 81].

void QDict::resize (uint newsize)

Changes the size of the hashtable the *newsize*. The contents of the dictionary are preserved, but all iterators on the dictionary become invalid.

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

uint QDict::size () const

Returns the size of the internal hash array (as specified in the constructor).

See also count() [p. 80].

void QDict::statistics () const

Debugging-only function that prints out the dictionary distribution using qDebug().

type * QDict::take (const QString & key)

Takes the item with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last item that was inserted will be taken.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary traversal order.

See also remove() [p. 81], clear() [p. 80] and setAutoDelete() [p. 134].

QDataStream & QDict::write (QDataStream & s, QPtrCollection::Item) const [virtual protected]

Writes a dictionary item to the stream *s* and returns a reference to the stream.

See also read() [p. 81].

QDictIterator Class Reference

The QDictIterator class provides an iterator for QDict collections.

`#include <qdict.h>`

Public Members

- **QDictIterator** (const QDict<type> & dict)
- **~QDictIterator** ()
- uint **count** () const
- bool **isEmpty** () const
- type * **toFirst** ()
- **operator type** * () const
- type * **current** () const
- QString **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()

Detailed Description

The QDictIterator class provides an iterator for QDict collections.

QDictIterator is implemented as a template class. Define a template instance QDictIterator<X> to create a dictionary iterator that operates on QDict<X> (dictionary of X*).

The traversal order is arbitrary; when we speak of the "first", "last" and "next" item we are talking in terms of this arbitrary order.

Multiple iterators may independently traverse the same dictionary. A QDict knows about all iterators that are operating on the dictionary. When an item is removed from the dictionary, QDict update all iterators that are referring to the removed item to point to the next item in the traversal order.

Example:

```
    QDict fields;
    fields.insert( "forename", new QLineEdit( this ) );
    fields.insert( "surname", new QLineEdit( this ) );
    fields.insert( "age", new QLineEdit( this ) );

    fields["forename"]->setText( "Homer" );
    fields["surname"]->setText( "Simpson" );
    fields["age"]->setText( "45" );

    QDictIterator it( extra );
    for( ; it.current(); ++it )
        cout << it.currentKey() << ": " << it.current()->text() << endl;
    cout << endl;

    // Output (random order):
    //  age: 45
    //  surname: Simpson
    //  forename: Homer
```

In the example we insert some line edits into a dictionary, then iterate over the dictionary printing the strings associated with those line edits.

See also QDict [p. 77], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QDictIterator::QDictIterator (const QDict<type> & dict)

Constructs an iterator for *dict*. The current iterator item is set to point to the first item in the dictionary, *dict*. First in this context means first in the arbitrary traversal order.

QDictIterator::~QDictIterator ()

Destroys the iterator.

uint QDictIterator::count () const

Returns the number of items in the dictionary over which the iterator is operating.

See also isEmpty() [p. 85].

type * QDictIterator::current () const

Returns a pointer to the current iterator item's value.

QString QDictIterator::currentKey () const

Returns the current iterator item's key.

bool QDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty, i.e. count() == 0; otherwise returns FALSE.

See also count() [p. 85].

QDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QDictIterator::operator() ()

Makes the next item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was 0, 0 is returned.

type * QDictIterator::operator++ ()

Prefix ++ makes the next item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was 0, 0 is returned.

type * QDictIterator::toFirst ()

Resets the iterator, making the first item the first current item. First in this context means first in the arbitrary traversal order. Returns a pointer to this item.

If the dictionary is empty it sets the current item to 0 and returns 0.

QIntCache Class Reference

The QIntCache class is a template class that provides a cache based on long keys.

```
#include <qintcache.h>
```

Public Members

- **QIntCache** (int maxCost = 100, int size = 17)
- **~QIntCache** ()
- int **maxCost** () const
- int **totalCost** () const
- void **setMaxCost** (int m)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- bool **insert** (long k, const type * d, int c = 1, int p = 0)
- bool **remove** (long k)
- type * **take** (long k)
- virtual void **clear** ()
- type * **find** (long k, bool ref = TRUE) const
- type * **operator[]** (long k) const
- void **statistics** () const

Detailed Description

The QIntCache class is a template class that provides a cache based on long keys.

QIntCache is implemented as a template class. Define a template instance QIntCache<X> to create a cache that operates on pointers to X, or X*.

A cache is a least recently used (LRU) list of cache items, accessed via `long` keys. Each cache item has a cost. The sum of item costs, totalCost(), will not exceed the maximum cache cost, maxCost(). If inserting a new item would cause the total cost to exceed the maximum cost, the least recently used items in the cache are removed.

Apart from insert(), by far the most important function is find() (which also exists as operator[]). This function looks up an item, returns it, and by default marks it as being the most recently used item.

There are also methods to remove() or take() an object from the cache. Calling setAutoDelete(TRUE) for a cache tells it to delete items that are removed. The default is to not delete items when they are removed (i.e. remove() and take() are equivalent).

When inserting an item into the cache, only the pointer is copied, not the item itself. This is called a shallow copy. It is possible to make the dictionary copy all of the item's data (known as a deep copy) when an item is inserted.

insert() calls the virtual function QPtrCollection::newItem() for the item to be inserted. Inherit a dictionary and reimplement it if you want deep copies.

When removing a cache item if auto-deletion is enabled the item will be automatically deleted.

There is a QIntCacheIterator which may be used to traverse the items in the cache in arbitrary order.

See also QIntCacheIterator [p. 91], QCache [p. 38], QAsciiCache [p. 13], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QIntCache::QIntCache (int maxCost = 100, int size = 17)

Constructs a cache whose contents will never have a total cost greater than *maxCost* and which is expected to contain less than *size* items.

size is actually the size of an internal hash array; it's usually best to make it prime and at least 50% bigger than the largest expected number of items in the cache.

Each inserted item is associated with a cost. When inserting a new item, if the total cost of all items in the cache will exceed *maxCost*, the cache will start throwing out the older (least recently used) items until there is enough room for the new item to be inserted.

QIntCache::~QIntCache ()

Removes all items from the cache and then destroys the int cache. If auto-deletion is enabled the cache's items are deleted. All iterators that access this cache will be reset.

void QIntCache::clear () [virtual]

Removes all items from the cache, and deletes them if auto-deletion has been enabled.

All cache iterators that operate this on cache are reset.

See also remove() [p. 89] and take() [p. 90].

uint QIntCache::count () const [virtual]

Returns the number of items in the cache.

See also totalCost() [p. 90].

type * QIntCache::find (long k, bool ref = TRUE) const

Returns the item associated with *k*, or null if the key does not exist in the cache. If *ref* is TRUE (the default), the item is moved to the front of the LRU list.

If there are two or more items with equal keys, the one that was inserted last is returned.

bool QIntCache::insert (long k, const type * d, int c = 1, int p = 0)

Inserts the item *d* into the cache with key *k* and cost *c* (default 1). Returns TRUE if it succeeds and FALSE if it fails.

The cache's size is limited, and if the total cost is too high, QIntCache will remove old, least-used items until there is room for this new item.

The parameter *p* is internal and should be left at the default value (0).

Warning: If this function returns FALSE, you must delete *d* yourself. Additionally, be very careful about using *d* after calling this function. Any other insertions into the cache, from anywhere in the application or within Qt itself, could cause the object to be discarded from the cache and the pointer to become invalid.

bool QIntCache::isEmpty () const

Returns TRUE if the cache is empty; otherwise returns FALSE.

int QIntCache::maxCost () const

Returns the maximum allowed total cost of the cache.

See also setMaxCost() [p. 89] and totalCost() [p. 90].

type * QIntCache::operator[] (long k) const

Returns the item associated with *k*, or null if *k* does not exist in the cache, and moves the item to the front of the LRU list.

If there are two or more items with equal keys, the one that was inserted last is returned.

This is the same as find(k, TRUE).

See also find() [p. 88].

bool QIntCache::remove (long k)

Removes the item associated with *k*, and returns TRUE if the item was present in the cache or FALSE if it was not.

The item is deleted if auto-deletion has been enabled, i.e. if you have called setAutoDelete(TRUE).

If there are two or more items with equal keys, the one that was inserted last is removed.

All iterators that refer to the removed item are set to point to the next item in the cache's traversal order.

See also take() [p. 90] and clear() [p. 88].

void QIntCache::setMaxCost (int m)

Sets the maximum allowed total cost of the cache to *m*. If the current total cost is above *m*, some items are removed immediately.

See also maxCost() [p. 89] and totalCost() [p. 90].

uint QIntCache::size () const

Returns the size of the hash array used to implement the cache. This should be a bit larger than count() is likely to be.

void QIntCache::statistics () const

A debug-only utility function. Prints out cache usage, hit/miss, and distribution information using qDebug(). This function does nothing in the release library.

type * QIntCache::take (long k)

Takes the item associated with *k* out of the cache without deleting it, and returns a pointer to the item taken out or null if the key does not exist in the cache.

If there are two or more items with equal keys, the one that was inserted last is taken.

All iterators that refer to the taken item are set to point to the next item in the cache's traversal order.

See also remove() [p. 89] and clear() [p. 88].

int QIntCache::totalCost () const

Returns the total cost of the items in the cache. This is an integer in the range 0 to maxCost().

See also setMaxCost() [p. 89].

QIntCacheIterator Class Reference

The QIntCacheIterator class provides an iterator for QIntCache collections.

```
#include <qintcache.h>
```

Public Members

- **QIntCacheIterator** (const QIntCache<type> & cache)
- **QIntCacheIterator** (const QIntCacheIterator<type> & ci)
- QIntCacheIterator<type> & **operator=** (const QIntCacheIterator<type> & ci)
- uint **count** () const
- bool **isEmpty** () const
- bool **atFirst** () const
- bool **atLast** () const
- type * **toFirst** ()
- type * **toLast** ()
- **operator type** * () const
- type * **current** () const
- long **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)
- type * **operator--** ()
- type * **operator-=** (uint jump)

Detailed Description

The QIntCacheIterator class provides an iterator for QIntCache collections.

Note that the traversal order is arbitrary; you are not guaranteed any particular order. If new objects are inserted into the cache while the iterator is active, the iterator may or may not see them.

Multiple iterators are completely independent, even when they operate on the same QIntCache. QIntCache updates all iterators that refer an item when that item is removed.

QIntCacheIterator provides an operator++(), and an operator+=() to traverse the cache; current() and currentKey() to access the current cache item and its key; atFirst() atLast(), which return TRUE if the iterator points to the first/last item in the cache; isEmpty(), which returns TRUE if the cache is empty; and count(), which returns the number of items in the cache.

Note that atFirst() and atLast() refer to the iterator's arbitrary ordering, not to the cache's internal LRU list.

See also QIntCache [p. 87], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QIntCacheIterator::QIntCacheIterator (const QIntCache<type> & cache)

Constructs an iterator for *cache*. The current iterator item is set to point to the first item in the *cache* (or rather, the first item is defined to be the item at which this constructor sets the iterator to point).

QIntCacheIterator::QIntCacheIterator (const QIntCacheIterator<type> & ci)

Constructs an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently from there on.

bool QIntCacheIterator::atFirst () const

Returns TRUE if the iterator points to the first item in the cache; otherwise returns FALSE. Note that this refers to the iterator's arbitrary ordering, not to the cache's internal LRU list.

See also toFirst() [p. 93] and atLast() [p. 92].

bool QIntCacheIterator::atLast () const

Returns TRUE if the iterator points to the last item in the cache; otherwise returns FALSE. Note that this refers to the iterator's arbitrary ordering, not to the cache's internal LRU list.

See also toLast() [p. 93] and atFirst() [p. 92].

uint QIntCacheIterator::count () const

Returns the number of items in the cache on which this iterator operates.

See also isEmpty() [p. 92].

type * QIntCacheIterator::current () const

Returns a pointer to the current iterator item.

long QIntCacheIterator::currentKey () const

Returns the key for the current iterator item.

bool QIntCacheIterator::isEmpty () const

Returns TRUE if the cache is empty; otherwise returns FALSE.

See also count() [p. 92].

QIntCacheIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QIntCacheIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the cache or if it was null, null is returned.

type * QIntCacheIterator::operator++ ()

Prefix ++ makes the iterator point to the item just after current(), and makes it the new current item for the iterator. If current() was the last item, operator--() returns 0.

type * QIntCacheIterator::operator+= (uint jump)

Returns the item *jump* positions after the current item, or null if it is beyond the last item. Makes this the current item.

type * QIntCacheIterator::operator-- ()

Prefix — makes the iterator point to the item just before current(), and makes it the new current item for the iterator. If current() was the first item, operator--() returns 0.

type * QIntCacheIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item, or null if it is beyond the first item. Makes this the current item.

QIntCacheIterator<type> & QIntCacheIterator::operator= (const QIntCacheIterator<type> & ci)

Makes this an iterator for the same cache as *ci*. The new iterator starts at the same item as ci.current(), but moves independently thereafter.

type * QIntCacheIterator::toFirst ()

Sets the iterator to point to the first item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if if the cache is empty.

See also toLast() [p. 93] and isEmpty() [p. 92].

type * QIntCacheIterator::toLast ()

Sets the iterator to point to the last item in the cache and returns a pointer to the item.

Sets the iterator to null and returns null if if the cache is empty.

See also toFirst() [p. 93] and isEmpty() [p. 92].

QIntDict Class Reference

The QIntDict class is a template class that provides a dictionary based on long keys.

`#include <qintdict.h>`

Inherits QPtrCollection [p. 132].

Public Members

- **QIntDict** (int size = 17)
- **QIntDict** (const QIntDict<type> & dict)
- **~QIntDict** ()
- QIntDict<type> & **operator=** (const QIntDict<type> & dict)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- void **insert** (long key, const type * item)
- void **replace** (long key, const type * item)
- bool **remove** (long key)
- type * **take** (long key)
- type * **find** (long key) const
- type * **operator[]** (long key) const
- virtual void **clear** ()
- void **resize** (uint newsize)
- void **statistics** () const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Protected Members

- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item) const

Detailed Description

The QIntDict class is a template class that provides a dictionary based on long keys.

QIntDict is implemented as a template class. Define a template instance QIntDict<X> to create a dictionary that operates on pointers to X (X*).

A dictionary is a collection of key-value pairs. The key is an `long` used for insertion, removal and lookup. The value is a pointer. Dictionaries provide very fast insertion and lookup.

Example:

```
QIntDict fields;
for ( int i = 0; i < 3; i++ )
    fields.insert( i, new QLineEdit( this ) );

fields[0]->setText( "Homer" );
fields[1]->setText( "Simpson" );
fields[2]->setText( "45" );

QIntDictIterator it( fields ); // See QIntDictIterator
for ( ; it.current(); ++it )
    cout << it.currentKey() << ": " << it.current()->text() << endl;

for ( int i = 0; i < 3; i++ )
    cout <text() << " "; // Prints "Homer Simpson 45"
cout << endl;

fields.remove( 1 ); // Does not delete the line edit
for ( int i = 0; i < 3; i++ )
    if ( fields[i] )
        cout <text() << " "; // Prints "Homer 45"
```

See QDict for full details, including the choice of dictionary size, and how deletions are handled.

See also QIntDictIterator [p. 100], QDict [p. 77], QAsciiDict [p. 20], QPtrDict [p. 135], Collection Classes [p. 9], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QIntDict::QIntDict (int size = 17)

Constructs a dictionary using an internal hash array of size *size*.

Setting *size* to a suitably large prime number (equal to or greater than the expected number of entries) makes the hash distribution better and hence the lookup faster.

QIntDict::QIntDict (const QIntDict<type> & dict)

Constructs a copy of *dict*.

Each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy).

QIntDict::~QIntDict ()

Removes all items from the dictionary and destroys it.

All iterators that access this dictionary will be reset.

See also setAutoDelete() [p. 134].

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QIntDict::clear () [virtual]

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that access this dictionary will be reset.

See also remove() [p. 97], take() [p. 98] and setAutoDelete() [p. 134].

Reimplemented from QPtrCollection [p. 133].

uint QIntDict::count () const [virtual]

Returns the number of items in the dictionary.

See also isEmpty() [p. 97].

Reimplemented from QPtrCollection [p. 133].

type * QIntDict::find (long key) const

Returns the item associated with *key*, or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the [] operator.

See also operator[]() [p. 97].

Example: table/bigtable/main.cpp.

void QIntDict::insert (long key, const type * item)

Insert item *item* into the dictionary using key *key*.

The key does not have to be unique. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also replace() [p. 97].

Example: scribble/scribble.cpp.

bool QIntDict::isEmpty () const

Returns TRUE if the dictionary is empty; otherwise returns FALSE.

See also count() [p. 96].

QIntDict<type> & QIntDict::operator= (const QIntDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared and then each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented.

type * QIntDict::operator[] (long key) const

Returns the item associated with *key*, or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last inserted of these will be found.

Equivalent to the find() function.

See also find() [p. 96].

QDataStream & QIntDict::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a dictionary item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 99].

bool QIntDict::remove (long key)

Removes the item associated with *key* from the dictionary. Returns TRUE if successful; otherwise returns FALSE, e.g. if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last inserted of these will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary's traversing order.

See also take() [p. 98], clear() [p. 96] and setAutoDelete() [p. 134].

Example: table/bigtable/main.cpp.

void QIntDict::replace (long key, const type * item)

If the dictionary has key *key*, this key's item is replaced with *item*. If the dictionary doesn't contain key *key*, *item* is inserted into the dictionary using key *key*.

Null items are not allowed.

Equivalent to:

```
QIntDict dict;
//  ...
if ( dict.find(key) )
    dict.remove( key );
dict.insert( key, item );
```

If there are two or more items with equal keys, then the last inserted of these will be replaced.

See also insert() [p. 96].

Example: table/bigtable/main.cpp.

void QIntDict::resize (uint newsize)

Changes the size of the hashtable to *newsize*. The contents of the dictionary are preserved, but all iterators on the dictionary become invalid.

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

uint QIntDict::size () const

Returns the size of the internal hash array (as specified in the constructor).

See also count() [p. 96].

void QIntDict::statistics () const

Debugging-only function that prints out the dictionary distribution using qDebug().

type * QIntDict::take (long key)

Takes the item associated with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last inserted of these will be taken.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary's traversing order.

See also remove() [p. 97], clear() [p. 96] and setAutoDelete() [p. 134].

QDataStream & QIntDict::write (QDataStream & s, QPtrCollection::Item) const [virtual protected]

Writes a dictionary item to the stream *s* and returns a reference to the stream.

See also read() [p. 97].

QIntDictIterator Class Reference

The QIntDictIterator class provides an iterator for QIntDict collections.

`#include <qintdict.h>`

Public Members

- **QIntDictIterator** (const QIntDict<type> & dict)
- **~QIntDictIterator** ()
- uint **count** () const
- bool **isEmpty** () const
- type * **toFirst** ()
- **operator type** * () const
- type * **current** () const
- long **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)

Detailed Description

The QIntDictIterator class provides an iterator for QIntDict collections.

QIntDictIterator is implemented as a template class. Define a template instance QIntDictIterator<X> to create a dictionary iterator that operates on QIntDict<X> (dictionary of X*).

Example:

```
    QIntDict fields;
    for ( int i = 0; i < 3; i++ )
        fields.insert( i, new QLineEdit( this ) );

    fields[0]->setText( "Homer" );
    fields[1]->setText( "Simpson" );
    fields[2]->setText( "45" );

    QIntDictIterator it( fields );
    for ( ; it.current(); ++it )
        cout << it.currentKey() << ": " << it.current()->text() << endl;

    // Output (random order):
    //  0: Homer
    //  1: Simpson
    //  2: 45
```

Note that the traversal order is arbitrary; you are not guaranteed the order above.

Multiple iterators may independently traverse the same dictionary. A QIntDict knows about all the iterators that are operating on the dictionary. When an item is removed from the dictionary, QIntDict updates all iterators that refer the removed item to point to the next item in the traversing order.

See also QIntDict [p. 94], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QIntDictIterator::QIntDictIterator (const QIntDict<type> & dict)

Constructs an iterator for *dict*. The current iterator item is set to point to the 'first' item in the *dict*. The first item refers to the first item in the dictionary's arbitrary internal ordering.

QIntDictIterator::~QIntDictIterator ()

Destroys the iterator.

uint QIntDictIterator::count () const

Returns the number of items in the dictionary this iterator operates over.

See also isEmpty() [p. 101].

type * QIntDictIterator::current () const

Returns a pointer to the current iterator item.

long QIntDictIterator::currentKey () const

Returns the key for the current iterator item.

bool QIntDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty; otherwise eturns FALSE.

See also count() [p. 101].

QIntDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QIntDictIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QIntDictIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QIntDictIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item, and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QIntDictIterator::toFirst ()

Sets the current iterator item to point to the first item in the dictionary and returns a pointer to the item. The first item refers to the first item in the dictionary's arbitrary internal ordering. If the dictionary is empty it sets the current item to null and returns null.

QMap Class Reference

The QMap class is a value-based template class that provides a dictionary.

`#include <qmap.h>`

Public Members

- typedef Key **key_type**
- typedef T **mapped_type**
- typedef QPair<const key_type, mapped_type> **value_type**
- typedef value_type * **pointer**
- typedef const value_type * **const_pointer**
- typedef value_type & **reference**
- typedef const value_type & **const_reference**
- typedef size_t **size_type**
- typedef QMapIterator<Key, T> **iterator**
- typedef QMapConstIterator<Key, T> **const_iterator**
- **QMap** ()
- **QMap** (const QMap<Key, T> & m)
- **QMap** (const std::map<Key, T> & m)
- **~QMap** ()
- QMap<Key, T> & **operator=** (const QMap<Key, T> & m)
- QMap<Key, T> & **operator=** (const std::map<Key, T> & m)
- iterator **begin** ()
- iterator **end** ()
- const_iterator **begin** () const
- const_iterator **end** () const
- iterator **replace** (const Key & k, const T & v)
- size_type **size** () const
- bool **empty** () const
- QPair<iterator, bool> **insert** (const value_type & x)
- void **erase** (iterator it)
- void **erase** (const key_type & k)
- size_type **count** (const key_type & k) const
- T & **operator[]** (const Key & k)
- void **clear** ()
- typedef QMapIterator<Key, T> **Iterator**
- typedef QMapConstIterator<Key, T> **ConstIterator**
- typedef T **ValueType**
- iterator **find** (const Key & k)

- const_iterator **find** (const Key & k) const
- const T & **operator[]** (const Key & k) const
- bool **contains** (const Key & k) const
- size_type **count** () const
- bool **isEmpty** () const
- iterator **insert** (const Key & key, const T & value, bool overwrite = TRUE)
- void **remove** (iterator it)
- void **remove** (const Key & k)

Protected Members

- void **detach** ()

Related Functions

- QDataStream & **operator>>** (QDataStream & s, QMap<Key, T> & m)
- QDataStream & **operator<<** (QDataStream & s, const QMap<Key, T> & m)

Detailed Description

The QMap class is a value-based template class that provides a dictionary.

QMap is a Qt implementation of an STL-like map container. It can be used in your application if the standard `map` is not available. QMap is part of the Qt Template Library.

QMap<Key, Data> defines a template instance to create a dictionary with keys of type Key and values of type Data. QMap does not store pointers to the members of the map; instead, it holds a copy of every member. For that reason, QMap is value-based, whereas QPtrList and QDict are pointer-based.

QMap contains and manages a collection of objects of type Data with associated key values of type Key and provides iterators that allow the contained objects to be addressed. QMap owns the contained items.

Some classes cannot be used within a QMap. For example everything derived from QObject and thus all classes that implement widgets. Only values can be used in a QMap. To qualify as a value, the class must provide

- A copy constructor
- An assignment operator
- A default constructor, i.e. a constructor that does not take any arguments.

Note that C++ defaults to field-by-field assignment operators and copy constructors if no explicit version is supplied. In many cases, this is sufficient.

The class used for the key requires that the `operator<` is implemented to define ordering of the keys.

QMap's function naming is consistent with the other Qt classes (e.g., count(), isEmpty()). QMap also provides extra functions for compatibility with STL algorithms, such as size() and empty(). Programmers already familiar with the STL `map` can use these functions instead.

Example:

```
    #include <qstring.h>
    #include <qmap.h>
    #include <qstring.h>
```

```
class Employee
{
public:
    Employee(): sn(0) {}
    Employee( const QString& forename, const QString& surname, int salary )
        : fn(forename), sn(surname), sal(salary)
    { }

    QString forename() const { return fn; }
    QString surname() const { return sn; }
    int salary() const { return sal; }
    void setSalary( int salary ) { sal = salary; }

private:
    QString fn;
    QString sn;
    int sal;
};

int main(int argc, char **argv)
{
    QApplication app( argc, argv );

    typedef QMap EmployeeMap;
    EmployeeMap map;

    map["JD001"] = Employee("John", "Doe", 50000);
    map["JD002"] = Employee("Jane", "Williams", 80000);
    map["TJ001"] = Employee("Tom", "Jones", 60000);

    Employee sasha( "Sasha", "Hind", 50000 );
    map["SH001"] = sasha;
    sasha.setSalary( 40000 );

    EmployeeMap::Iterator it;
    for ( it = map.begin(); it != map.end(); ++it ) {
        printf( "%s: %s, %s earns %d\n",
                it.key().latin1(),
                it.data().surname().latin1(),
                it.data().forename().latin1(),
                it.data().salary() );
    }
    return 0;
}
```

Program output:

```
JD001: Doe, John earns 50000
JW002: Williams, Jane earns 80000
SH001: Hind, Sasha earns 50000
TJ001: Jones, Tom earns 60000
```

The latest changes to Sasha's salary did not affect the value in the list because the map created a copy of Sasha's entry. In addition, notice that the items are sorted alphabetically (by key) when iterating over the map.

There are several ways to find items in a map. The begin() and end() functions return iterators to the beginning and end of the map. The advantage of using an iterator is that you can move forward or backward by incrementing/decrementing the iterator. The iterator returned by end() points to the element which is one past the last element in the container. The past-the-end iterator is still associated with the map it belongs to, however it is *not* dereferenceable; operator*() will not return a well-defined value. If the map is empty, the iterator returned by begin() will equal the iterator returned by end().

Another way to find an element in the map is by using the find() function. This returns an iterator pointing to the desired item or to the end() iterator if no such element exists.

Another approach uses the operator[]. But be warned: if the map does not contain an entry for the element you are looking for, operator[] inserts a default value. If you do not know that the element you are searching for is really in the list, you should not use operator[]. The following example illustrates this:

```
        QMap map;
        map["Clinton"] = "Bill";
        str << map["Clinton"] << map["Bush"] << endl;
```

The code fragment will print out "Clinton", "". Since the value associated with the "Bush" key did not exist, the map inserted a default value (in this case, an empty string). If you are not sure whether a certain element is in the map, you should use find() and iterators instead.

If you just want to know whether a certain key is contained in the map, use the contains() function. In addition, count() tells you how many keys there are currently in the map.

It is safe to have multiple iterators at the same time. If some member of the map is removed, only iterators pointing to the removed member become invalid; inserting in the map does not invalidate any iterators.

Since QMap is value-based, there is no need to be concerned about deleting items in the map. The map holds its own copies and will free them if the corresponding member or the map itself is deleted.

QMap is implicitly shared. This means you can just make copies of the map in time O(1). If multiple QMap instances share the same data and one is modifying the map's data, this modifying instance makes a copy and modifies its private copy; it thus does not affect other instances. From a developer's point of view you can think that a QMap and a copy of this map have nothing to do with each other. If a QMap is being used in a multi-threaded program, you must protect all access to the map. See QMutex.

There are several ways of inserting new items into the map. One uses the insert() method; the other one uses operator[] like this:

```
    QMap map;
    map["Clinton"] = "Bill";
    map.insert( qMakePair("Bush", "George") );
```

Items can also be removed from the map in several ways. The first is to pass an iterator to remove(). The other is to pass a key value to remove(), which will delete the entry with the requested key. In addition you can clear the entire map using the clear() method.

See also QMapIterator [p. 116], Qt Template Library Classes, Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Type Documentation

QMap::ConstIterator

The map's const iterator type, Qt style.

QMap::Iterator

The map's iterator type, Qt style.

QMap::ValueType

Corresponds to QPair<key_type, mapped_type>, Qt style.

QMap::const_iterator

The map's const iterator type.

QMap::const_pointer

Const pointer to value_type.

QMap::const_reference

Const reference to value_type.

QMap::iterator

The map's iterator type.

QMap::key_type

The map's key type.

QMap::mapped_type

The map's data type.

QMap::pointer

Pointer to value_type.

QMap::reference

Reference to value_type.

QMap::size_type

An unsigned integral type, used to represent various sizes.

QMap::value_type

Corresponds to QPair<key_type, mapped_type>.

Member Function Documentation

QMap::QMap ()

Constructs an empty map.

QMap::QMap (const QMap<Key, T> & m)

Constructs a copy of *m*.

This operation costs O(1) time because QMap is implicitly shared. The first instance of applying modifications to a shared map will create a copy that takes in turn O(n) time. However, returning a QMap from a function is very fast.

QMap::QMap (const std::map<Key, T> & m)

Constructs a copy of *m*.

QMap::~QMap ()

Destroys the map. References to the values in the map and all iterators of this map become invalidated. Since QMap is highly tuned for performance you won't see warnings if you use invalid iterators, because it is not possible for an iterator to check whether it is valid or not.

iterator QMap::begin ()

Returns an iterator pointing to the first element in the map. This iterator equals end() if the map is empty.

The items in the map are traversed in the order defined by operator<(Key, Key).

See also end() [p. 109] and QMapIterator [p. 116].

const_iterator QMap::begin () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

See also end() [p. 109] and QMapConstIterator [p. 113].

void QMap::clear ()

Removes all items from the map.

See also remove() [p. 111].

bool QMap::contains (const Key & k) const

Returns TRUE if the map contains an item with key *k*; otherwise returns FALSE.

size_type QMap::count (const key_type & k) const

Returns the number of items whose key is *k*. Since QMap does not allow duplicate keys, the return value is always 0 or 1.

This function is provided for STL compatibility.

size_type QMap::count () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the number of items in the map.

See also isEmpty() [p. 110].

void QMap::detach () [protected]

If the map does not share its data with another QMap instance, nothing happens; otherwise the function creates a new copy of this map and detaches from the shared one. This function is called whenever the map is modified. The implicit sharing mechanism is implemented this way.

bool QMap::empty () const

Returns TRUE if the map contains zero items; otherwise returns FALSE.

This function is provided for STL compatibility. It is equivalent to isEmpty().

See also size() [p. 111].

iterator QMap::end ()

The iterator returned by end() points to the element which is one past the last element in the container. The past-the-end iterator is still associated with the map it belongs to, however it is *not* dereferenceable; operator*() will not return a well-defined value.

This iterator equals begin() if the map is empty.

See also begin() [p. 108] and QMapIterator [p. 116].

const_iterator QMap::end () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The iterator returned by end() points to the element which is one past the last element in the container. The past-the-end iterator is still associated with the map it belongs to, however it is *not* dereferenceable; operator*() will not return a well-defined value.

This iterator equals begin() if the map is empty.

See also begin() [p. 108] and QMapConstIterator [p. 113].

void QMap::erase (iterator it)

Removes the item associated with the iterator *it* from the map.

This function is provided for STL compatibility. It is equivalent to remove().

See also clear() [p. 108].

void QMap::erase (const key_type & k)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes the item with the key *k* from the map.

iterator QMap::find (const Key & k)

Returns an iterator pointing to the element with key *k* in the map.

Returns end() if no key matched.

See also QMapIterator [p. 116].

const_iterator QMap::find (const Key & k) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an iterator pointing to the element with key *k* in the map.

Returns end() if no key matched.

See also QMapConstIterator [p. 113].

iterator QMap::insert (const Key & key, const T & value, bool overwrite = TRUE)

Inserts the *value* with *key*. If there is already a value associated with *key*, it is replaced, unless *overwrite* is FALSE (it is TRUE by default).

QPair<iterator, bool> QMap::insert (const value_type & x)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts the (key, value) pair *x* into the map. *x* is a QPair whose `first` element is a key to be inserted and whose `second` element is the associated value to be inserted. Returns a pair whose `first` element is an iterator pointing to the inserted item and whose `second` element is a bool indicating TRUE if *x* was inserted and FALSE if it was not inserted because it was already present.

bool QMap::isEmpty () const

Returns TRUE if the map contains zero items; otherwise returns FALSE.

See also count() [p. 109].

QMap<Key, T> & QMap::operator= (const QMap<Key, T> & m)

Assigns *m* to this map and returns a reference to this map.

All iterators of the current map become invalidated by this operation. The cost of such an assignment is O(1), because QMap is implicitly shared.

QMap<Key, T> & QMap::operator= (const std::map<Key, T> & m)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns *m* to this map and returns a reference to this map.

All iterators of the current map become invalidated by this operation.

T & QMap::operator[] (const Key & k)

Returns the value associated with the key *k*. If no such key is present, an empty item is inserted with this key and a reference to the item is returned.

You can use this operator both for reading and writing:

```
QMap map;
map["Clinton"] = "Bill";
stream << map["Clinton"];
```

const T & QMap::operator[] (const Key & k) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Warning: This function differs from the non-const version of the same function. It will *not* insert an empty value if the key *k* does not exist. This may lead to logic errors in your program. You should check if the element exists before calling this function.

Returns the value associated with the key *k*. If no such key is present, a reference to an empty item is returned.

void QMap::remove (iterator it)

Removes the item associated with the iterator *it* from the map.

See also clear() [p. 108].

void QMap::remove (const Key & k)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes the item with the key *k* from the map.

iterator QMap::replace (const Key & k, const T & v)

Replaces the value with key *k* from the map if possible, and inserts the new value *v* with key *k* in the map.

See also insert() [p. 110] and remove() [p. 111].

size_type QMap::size () const

Returns the number of items in the map.

This function is provided for STL compatibility. It is equivalent to count().

See also empty() [p. 109].

Related Functions

QDataStream & operator<< (QDataStream & s, const QMap<Key, T> & m)

Writes the map *m* to the stream *s*. The types *Key* and *T* must implement the streaming operator as well.

QDataStream & operator>> (QDataStream & s, QMap<Key, T> & m)

Reads the map *m* from the stream *s*. The types *Key* and *T* must implement the streaming operator as well.

QMapConstIterator Class Reference

The QMapConstIterator class provides an iterator for QMap.

`#include <qmap.h>`

Public Members

- typedef std::bidirectional_iterator_tag **iterator_category**
- typedef T **value_type**
- typedef const T * **pointer**
- typedef const T & **reference**
- **QMapConstIterator** ()
- **QMapConstIterator** (QMapNode<K, T> * p)
- **QMapConstIterator** (const QMapConstIterator<K, T> & it)
- **QMapConstIterator** (const QMapIterator<K, T> & it)
- bool **operator==** (const QMapConstIterator<K, T> & it) const
- bool **operator!=** (const QMapConstIterator<K, T> & it) const
- const T & **operator*** () const
- const K & **key** () const
- const T & **data** () const
- QMapConstIterator<K, T> & **operator++** ()
- QMapConstIterator<K, T> **operator++** (int)
- QMapConstIterator<K, T> & **operator--** ()
- QMapConstIterator<K, T> **operator--** (int)

Detailed Description

The QMapConstIterator class provides an iterator for QMap.

In contrast to QMapIterator, this class is used to iterate over a const map. It does not allow you to modify the values of the map because this would break the const semantics.

For more information on QMap iterators, see QMapIterator. and the QMap example.

See also QMap [p. 103], QMapIterator [p. 116], Qt Template Library Classes and Non-GUI Classes.

Member Type Documentation

QMapConstIterator::iterator_category

The type of iterator category, `std::bidirectional_iterator_tag`.

QMapConstIterator::pointer

Const pointer to value_type.

QMapConstIterator::reference

Const reference to value_type.

QMapConstIterator::value_type

The type of const value.

Member Function Documentation

QMapConstIterator::QMapConstIterator ()

Constructs an uninitialized iterator.

QMapConstIterator::QMapConstIterator (QMapNode<K, T> * p)

Constructs an iterator starting at node *p*.

QMapConstIterator::QMapConstIterator (const QMapConstIterator<K, T> & it)

Constructs a copy of the iterator, *it*.

QMapConstIterator::QMapConstIterator (const QMapIterator<K, T> & it)

Constructs a copy of the iterator, *it*.

const T & QMapConstIterator::data () const

Returns a const reference to the data of the current item.

const K & QMapConstIterator::key () const

Returns a const reference to the current key.

bool QMapConstIterator::operator!= (const QMapConstIterator<K, T> & it) const

Compares the iterator to the *it* iterator and returns FALSE if they point to the same item; otherwise returns TRUE.

const T & QMapConstIterator::operator* () const

Dereference operator. Returns a const reference to the current item. The same as data().

QMapConstIterator<K, T> & QMapConstIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the map. Incrementing the iterator returned by end() causes undefined results.

QMapConstIterator<K, T> QMapConstIterator::operator++ (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the map. Incrementing the iterator returned by end() causes undefined results.

QMapConstIterator<K, T> & QMapConstIterator::operator-- ()

Prefix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the map. Decrementing the iterator returned by begin() causes undefined results.

QMapConstIterator<K, T> QMapConstIterator::operator-- (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the map. Decrementing the iterator returned by begin() causes undefined results.

bool QMapConstIterator::operator== (const QMapConstIterator<K, T> & it) const

Compares the iterator to the *it* iterator and returns TRUE if they point to the same item; otherwise returns FALSE.

QMapIterator Class Reference

The QMapIterator class provides an iterator for QMap.

```
#include <qmap.h>
```

Public Members

- typedef std::bidirectional_iterator_tag **iterator_category**
- typedef T **value_type**
- typedef T * **pointer**
- typedef T & **reference**
- **QMapIterator** ()
- **QMapIterator** (QMapNode<K, T> * p)
- **QMapIterator** (const QMapIterator<K, T> & it)
- bool **operator==** (const QMapIterator<K, T> & it) const
- bool **operator!=** (const QMapIterator<K, T> & it) const
- T & **operator*** ()
- const T & **operator*** () const
- const K & **key** () const
- T & **data** ()
- const T & **data** () const
- QMapIterator<K, T> & **operator++** ()
- QMapIterator<K, T> **operator++** (int)
- QMapIterator<K, T> & **operator--** ()
- QMapIterator<K, T> **operator--** (int)

Detailed Description

The QMapIterator class provides an iterator for QMap.

You cannot create an iterator by yourself. Instead, you have to ask a map to give you one. An iterator is as big as a pointer; on 32-bit machines that means 4 bytes, on 64-bit ones 8 bytes. That makes copying them very fast. They resemble the semantics of pointers as much as possible, and they are almost as fast as usual pointers. See the QMap example [p. 104].

The only way to traverse a map is to use iterators. QMap is highly optimized for performance and memory usage. On the other hand this means that you have to be a bit more careful with what you are doing. QMap does not know about all its iterators, and the iterators don't even know to which map they belong. That makes things fast but a bit dangerous because it is up to you to make sure that the iterators you are using are still valid. QDictIterator will be able to give warnings, whereas QMapIterator may end up in an undefined state.

For every Iterator there is also a ConstIterator. You have to use the ConstIterator to access a QMap in a const environment or if the reference or pointer to the map is itself const. Its semantics are the same, but it returns only const references to the item it points to.

See also QMap [p. 103], QMapConstIterator [p. 113], Qt Template Library Classes and Non-GUI Classes.

Member Type Documentation

QMapIterator::iterator_category

The type of iterator category, `std::bidirectional_iterator_tag`.

QMapIterator::pointer

Pointer to value_type.

QMapIterator::reference

Reference to value_type.

QMapIterator::value_type

The type of value.

Member Function Documentation

QMapIterator::QMapIterator ()

Creates an uninitialized iterator.

QMapIterator::QMapIterator (QMapNode<K, T> * p)

Constructs an iterator starting at node *p*.

QMapIterator::QMapIterator (const QMapIterator<K, T> & it)

Constructs a copy of the iterator, *it*.

T & QMapIterator::data ()

Returns a reference to the current item.

const T & QMapIterator::data () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const reference to the data of the current item.

const K & QMapIterator::key () const

Returns a const reference to the data of the current key.

bool QMapIterator::operator!= (const QMapIterator<K, T> & it) const

Compares the iterator to the *it* iterator and returns FALSE if they point to the same item; otherwise returns TRUE.

T & QMapIterator::operator* ()

Dereference operator. Returns a reference to the current item. The same as data().

const T & QMapIterator::operator* () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Dereference operator. Returns a const reference to the current item. The same as data().

QMapIterator<K, T> & QMapIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the map. Incrementing the iterator returned by end() causes undefined results.

QMapIterator<K, T> QMapIterator::operator++ (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the map. Incrementing the iterator returned by end() causes undefined results.

QMapIterator<K, T> & QMapIterator::operator-- ()

Prefix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the map. Decrementing the iterator returned by begin() causes undefined results.

QMapIterator<K, T> QMapIterator::operator-- (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the map. Decrementing the iterator returned by begin() causes undefined results.

bool QMapIterator::operator== (const QMapIterator<K, T> & it) const

Compares the iterator to the *it* iterator and returns TRUE if they point to the same item; otherwise returns FALSE.

QMemArray Class Reference

The QMemArray class is a template class that provides arrays of simple types.

`#include <qmemarray.h>`

Inherited by QByteArray [p. 37] and QPointArray [Graphics with Qt].

Public Members

- typedef type * **Iterator**
- typedef const type * **ConstIterator**
- **QMemArray** ()
- **QMemArray** (int size)
- **QMemArray** (const QMemArray<type> & a)
- **~QMemArray** ()
- QMemArray<type> & **operator=** (const QMemArray<type> & a)
- type * **data** () const
- uint **nrefs** () const
- uint **size** () const
- uint **count** () const
- bool **isEmpty** () const
- bool **isNull** () const
- bool **resize** (uint size)
- bool **truncate** (uint pos)
- bool **fill** (const type & v, int size = -1)
- virtual void **detach** ()
- QMemArray<type> **copy** () const
- QMemArray<type> & **assign** (const QMemArray<type> & a)
- QMemArray<type> & **assign** (const type * data, uint size)
- QMemArray<type> & **duplicate** (const QMemArray<type> & a)
- QMemArray<type> & **duplicate** (const type * data, uint size)
- QMemArray<type> & **setRawData** (const type * data, uint size)
- void **resetRawData** (const type * data, uint size)
- int **find** (const type & v, uint index = 0) const
- int **contains** (const type & v) const
- void **sort** ()
- int **bsearch** (const type & v) const
- type & **operator[]** (int index) const
- type & **at** (uint index) const
- **operator const type** * () const
- bool **operator==** (const QMemArray<type> & a) const

- bool **operator!=** (const QMemArray<type> & a) const
- Iterator **begin** ()
- Iterator **end** ()
- ConstIterator **begin** () const
- ConstIterator **end** () const

Protected Members

- **QMemArray** (int, int)

Related Functions

- Q_UINT16 **qChecksum** (const char * data, uint len)
- QDataStream & **operator<<** (QDataStream & s, const QByteArray & a)
- QDataStream & **operator>>** (QDataStream & s, QByteArray & a)

Detailed Description

The QMemArray class is a template class that provides arrays of simple types.

QMemArray is implemented as a template class. Define a template instance QMemArray<X> to create an array that contains X items.

QMemArray stores the array elements directly in the array. It can deal only with simple types (i.e. C++ types, structs, and classes that have no constructors, destructors, or virtual functions). QMemArray uses bitwise operations to copy and compare array elements.

The QPtrVector collection class is also a kind of array. Like most collection classes, it has pointers to the contained items.

QMemArray uses explicit sharing with a reference count. If more than one array share common data and one array is modified, all arrays will be modified.

The benefit of sharing is that a program does not need to duplicate data when it is not required, which results in less memory usage and less copying of data.

Example:

```
#include <qmemarray.h>
#include

QMemArray fib( int num ) // returns fibonacci array
{
    Q_ASSERT( num > 2 );
    QMemArray f( num ); // array of ints

    f[0] = f[1] = 1;
    for ( int i = 2; i < num; i++ )
        f[i] = f[i-1] + f[i-2];

    return f;
}

int main()
```

```
    {
        QMemArray a = fib( 6 ); // get 6 first fibonaccis
        for ( int i = 0; i < a.size(); i++ )
            qDebug( "%d: %d", i, a[i] );

        qDebug( "1 is found %d times", a.contains(1) );
        qDebug( "5 is found at index %d", a.find(5) );

        return 0;
    }
```

Program output:

```
    0: 1
    1: 1
    2: 2
    3: 3
    4: 5
    5: 8
    1 is found 2 times
    5 is found at index 4
```

Note about using QMemArray for manipulating structs or classes: Compilers will often pad the size of structs of odd sizes up to the nearest word boundary. This will then be the size QMemArray will use for its bitwise element comparisons. Because the remaining bytes will typically be uninitialized, this can cause find() etc. to fail to find the element. Example:

```
    // MyStruct may be padded to 4 or 8 bytes
    struct MyStruct
    {
        short i; // 2 bytes
        char c;  // 1 byte
    };

    QMemArray a(1);
    a[0].i = 5;
    a[0].c = 't';

    MyStruct x;
    x.i = '5';
    x.c = 't';
    int i = a.find( x ); // may return -1 if the pad bytes differ
```

To work around this, make sure that you use a struct where sizeof() returns the same as the sum of the sizes of the members either by changing the types of the struct members or by adding dummy members.

QMemArray data can be traversed by iterators (see begin() and end()). The number of items is returned by count(). The array can be resized with resize() and filled using fill().

You can make a shallow copy of the array with assign() (or operator=()) and a deep copy with duplicate().

Search for values in the array with find() and contains(). For sorted arrays (see sort()) you can search using bsearch().

You can set the data directly using setRawData() and resetRawData(), although this requires care.

See also Shared Classes [Programming with Qt] and Non-GUI Classes.

Member Type Documentation

QMemArray::ConstIterator

A const QMemArray iterator.

See also begin() [p. 124] and end() [p. 125].

QMemArray::Iterator

A QMemArray iterator.

See also begin() [p. 124] and end() [p. 125].

Member Function Documentation

QMemArray::QMemArray (int, int) [protected]

Constructs an array *without allocating* array space. The arguments should be (0, 0). Use at your own risk.

QMemArray::QMemArray ()

Constructs a null array.

See also isNull() [p. 126].

QMemArray::QMemArray (int size)

Constructs an array with room for *size* elements. Makes a null array if *size* == 0.

The elements are left uninitialized.

See also resize() [p. 127] and isNull() [p. 126].

QMemArray::QMemArray (const QMemArray<type> & a)

Constructs a shallow copy of *a*.

See also assign() [p. 123].

QMemArray::~QMemArray ()

Dereferences the array data and deletes it if this was the last reference.

QMemArray<type> & QMemArray::assign (const QMemArray<type> & a)

Shallow copy. Dereferences the current array and references the data contained in *a* instead. Returns a reference to this array.

See also operator=() [p. 126].

QMemArray<type> & QMemArray::assign (const type * data, uint size)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Shallow copy. Dereferences the current array and references the array data *data*, which contains *size* elements. Returns a reference to this array.

Do not delete *data* later; QMemArray will take care of it.

type & QMemArray::at (uint index) const

Returns a reference to the element at position *index* in the array.

This can be used to both read and set an element.

See also operator[]() [p. 127].

Iterator QMemArray::begin ()

Returns an iterator pointing at the beginning of this array. This iterator can be used in the same way as the iterators of QValueList and QMap, for example. In fact, not only does it behave like a usual pointer, it is a pointer.

ConstIterator QMemArray::begin () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const iterator pointing at the beginning of this array. This iterator can be used in the same way as the iterators of QValueList and QMap, for example. In fact, not only does it behave like a usual pointer, it is a pointer.

int QMemArray::bsearch (const type & v) const

In a sorted array, finds the first occurrence of *v* by using binary search. For a sorted array this is generally much faster than find(), which does a linear search.

Returns the position of *v*, or -1 if *v* could not be found.

See also sort() [p. 128] and find() [p. 126].

int QMemArray::contains (const type & v) const

Returns the number of times *v* occurs in the array.

See also find() [p. 126].

QMemArray<type> QMemArray::copy () const

Returns a deep copy of this array.

See also detach() [p. 125] and duplicate() [p. 125].

uint QMemArray::count () const

Returns the same as size().

See also size() [p. 128].

Example: scribble/scribble.cpp.

type * QMemArray::data () const

Returns a pointer to the actual array data.

The array is a null array if data() == 0 (null pointer).

See also isNull() [p. 126].

Examples: fileiconview/qfileiconview.cpp and network/networkprotocol/nntp.cpp.

void QMemArray::detach () [virtual]

Detaches this array from shared array data; i.e. it makes a private, deep copy of the data.

Copying will be performed only if the reference count is greater than one.

See also copy() [p. 124].

Reimplemented in QBitArray.

QMemArray<type> & QMemArray::duplicate (const QMemArray<type> & a)

Deep copy. Dereferences the current array and obtains a copy of the data contained in *a* instead. Returns a reference to this array.

See also copy() [p. 124].

QMemArray<type> & QMemArray::duplicate (const type * data, uint size)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Deep copy. Dereferences the current array and obtains a copy of the array data *data* instead. Returns a reference to this array. The size of the array is given by *size*.

See also copy() [p. 124].

Iterator QMemArray::end ()

Returns an iterator pointing behind the last element of this array. This iterator can be used in the same way as the iterators of QValueList and QMap, for example. In fact, not only does it behave like a usual pointer, it is a pointer.

ConstIterator QMemArray::end () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const iterator pointing behind the last element of this array. This iterator can be used in the same way as the iterators of QValueList and QMap, for example. In fact, not only does it behave like a usual pointer, it is a pointer.

bool QMemArray::fill (const type & v, int size = -1)

Fills the array with the value *v*. If *size* is specified as different from -1, then the array will be resized before being filled.

Returns TRUE if successful, or FALSE if the memory cannot be allocated (only when *size* != -1).

See also resize() [p. 127].

int QMemArray::find (const type & v, uint index = 0) const

Finds the first occurrence of *v*, starting at position *index*.

Returns the position of *v*, or -1 if *v* could not be found.

See also contains() [p. 124].

bool QMemArray::isEmpty () const

Returns TRUE if the array is empty; otherwise returns FALSE.

isEmpty() is equivalent to isNull() for QMemArray (unlike QString).

bool QMemArray::isNull () const

Returns TRUE if the array is null; otherwise returns FALSE.

A null array has size() == 0 and data() == 0.

uint QMemArray::nrefs () const

Returns the reference count for the shared array data. This reference count is always greater than zero.

QMemArray::operator const type * () const

Cast operator. Returns a pointer to the array.

See also data() [p. 125].

bool QMemArray::operator!= (const QMemArray<type> & a) const

Returns TRUE if this array is different from *a*; otherwise returns FALSE.

The two arrays are compared bitwise.

See also operator==() [p. 127].

QMemArray<type> & QMemArray::operator= (const QMemArray<type> & a)

Assigns a shallow copy of *a* to this array and returns a reference to this array.

Equivalent to assign(a).

bool QMemArray::operator== (const QMemArray<type> & a) const

Returns TRUE if this array is equal to *a*; otherwise returns FALSE.

The two arrays are compared bitwise.

See also operator!=() [p. 126].

type & QMemArray::operator[] (int index) const

Returns a reference to the element at position *index* in the array.

This can be used to both read and set an element. Equivalent to at().

See also at() [p. 124].

void QMemArray::resetRawData (const type * data, uint size)

Resets raw data that was set using setRawData().

The arguments must be the *data* and length, *size*, that were passed to setRawData(). This is for consistency checking.

See also setRawData() [p. 127].

bool QMemArray::resize (uint size)

Resizes (expands or shrinks) the array to *size* elements. The array becomes a null array if *size* == 0.

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

New elements will not be initialized.

See also size() [p. 128].

Example: fileiconview/qfileiconview.cpp.

QMemArray<type> & QMemArray::setRawData (const type * data, uint size)

Sets raw data and returns a reference to the array.

Dereferences the current array and sets the new array data to *data* and the new array size to *size*. Do not attempt to resize or re-assign the array data when raw data has been set. Call resetRawData(*data*, *size*) to reset the array.

Setting raw data is useful because it sets QMemArray data without allocating memory or copying data.

Example I (intended use):

```
static char bindata[] = { 231, 1, 44, ... };
QByteArray  a;
a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
QDataStream s( a, IO_ReadOnly );            // open on a's data
s >> ;                           // read raw bindata
a.resetRawData( bindata, sizeof(bindata) ); // finished
```

Example II (you don't want to do this):

```
static char bindata[] = { 231, 1, 44, ... };
```

```
    QByteArray  a, b;
    a.setRawData( bindata, sizeof(bindata) );   // a points to bindata
    a.resize( 8 );                              // will crash
    b = a;                                      // will crash
    a[2] = 123;                                 // might crash
    // forget to resetRawData: will crash
```

Warning: If you do not call resetRawData(), QMemArray will attempt to deallocate or reallocate the raw data, which might not be too good. Be careful.

See also resetRawData() [p. 127].

uint QMemArray::size () const

Returns the size of the array (max number of elements).

The array is a null array if size() == 0.

See also isNull() [p. 126] and resize() [p. 127].

void QMemArray::sort ()

Sorts the array elements in ascending order, using bitwise comparison (memcmp()).

See also bsearch() [p. 124].

bool QMemArray::truncate (uint pos)

Truncates the array at position *pos*.

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

Equivalent to resize(*pos*).

See also resize() [p. 127].

Related Functions

QDataStream & operator<< (QDataStream & s, const QByteArray & a)

Writes byte array *a* to the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

QDataStream & operator>> (QDataStream & s, QByteArray & a)

Reads a byte array into *a* from the stream *s* and returns a reference to the stream.

See also Format of the QDataStream operators [Input/Output and Networking with Qt].

Q_UINT16 qChecksum (const char * data, uint len)

Returns the CRC-16 checksum of *len* bytes starting at *data*.

The checksum is independent of the byte order (endianness).

QPair Class Reference

The QPair class is a value-based template class that provides a pair of elements.

```
#include <qpair.h>
```

Public Members

- typedef T1 **first_type**
- typedef T2 **second_type**
- **QPair** ()
- **QPair** (const T1 & t1, const T2 & t2)

Detailed Description

The QPair class is a value-based template class that provides a pair of elements.

QPair is a Qt implementation of an STL-like pair. It can be used in your application if the standard pair<> is not available.

QPair<T1, T2> defines a template instance to create a pair of values that contains two values of type T1 and T2. Please note that QPair does not store pointers to the two elements; it holds a copy of every member. This is why these kinds of classes are called *value based*. If you're interested in *pointer based* classes see, for example, QPtrList and QDict.

QPair holds one copy of type T1 and one copy of type T2, but does not provide iterators to access these elements. Rather, the two elements (`first` and `second`) are public member variables of the pair. QPair owns the contained elements. For more relaxed ownership semantics, see QPtrCollection and friends which are pointer-based containers.

Some classes cannot be used within a QPair: for example, all classes derived from QObject and thus all classes that implement widgets. Only "values" can be used in a QPair. To qualify as a value the class must provide:

- A copy constructor
- An assignment operator
- A constructor that takes no argument

Note that C++ defaults to field-by-field assignment operators and copy constructors if no explicit version is supplied. In many cases this is sufficient.

QPair uses an STL-like syntax to manipulate and address the objects it contains. See the QTL documentation for more information.

Functions that need to return two values can use a QPair. The qMakePair() convenience function makes it easy to create QPair objects.

See also Qt Template Library Classes, Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Type Documentation

QPair::first_type

The type of the first element in the pair.

QPair::second_type

The type of the second element in the pair.

Member Function Documentation

QPair::QPair ()

Constructs an empty pair. The `first` and `second` elements are default constructed.

QPair::QPair (const T1 & t1, const T2 & t2)

Constructs a pair and initializes the `first` element with *t1* and the `second` element with *t2*.

QPtrCollection Class Reference

The QPtrCollection class is the base class of most pointer-based Qt collections.

`#include <qptrcollection.h>`

Inherited by QAsciiDict [p. 20], QCache [p. 38], QDict [p. 77], QIntDict [p. 94], QPtrList [p. 144], QPtrDict [p. 135] and QPtrVector [p. 167].

Public Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)
- virtual uint **count** () const
- virtual void **clear** ()
- typedef void * **Item**

Protected Members

- **QPtrCollection** ()
- **QPtrCollection** (const QPtrCollection & source)
- virtual **~QPtrCollection** ()
- virtual Item **newItem** (Item d)
- virtual void **deleteItem** (Item d)

Detailed Description

The QPtrCollection class is the base class of most pointer-based Qt collections.

The QPtrCollection class is an abstract base class for the Qt collection classes QDict, QPtrList, etc. Qt also includes value based collections, e.g. QValueList, QMap, etc.

A QPtrCollection only knows about the number of objects in the collection and the deletion strategy (see setAutoDelete()).

A collection is implemented using the Item (generic collection item) type, which is a `void*`. The template classes that create the real collections cast the Item to the required type.

See also Collection Classes [p. 9] and Non-GUI Classes.

Member Type Documentation

QPtrCollection::Item

This type is the generic "item" in a QPtrCollection.

Member Function Documentation

QPtrCollection::QPtrCollection () [protected]

Constructs a collection. The constructor is protected because QPtrCollection is an abstract class.

QPtrCollection::QPtrCollection (const QPtrCollection & source) [protected]

Constructs a copy of *source* with autoDelete() set to FALSE. The constructor is protected because QPtrCollection is an abstract class.

Note that if *source* has autoDelete turned on, copying it will risk memory leaks, reading freed memory, or both.

QPtrCollection::~QPtrCollection () [virtual protected]

Destroys the collection. The destructor is protected because QPtrCollection is an abstract class.

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QPtrCollection::clear () [virtual]

Removes all objects from the collection. The objects will be deleted if auto-delete has been enabled.

See also setAutoDelete() [p. 134].

Reimplemented in QAsciiDict, QCache, QDict, QIntDict, QPtrList, QPtrDict and QPtrVector.

uint QPtrCollection::count () const [virtual]

Returns the number of objects in the collection.

Reimplemented in QAsciiDict, QCache, QDict, QIntDict, QPtrList, QPtrDict and QPtrVector.

void QPtrCollection::deleteItem (Item d) [virtual protected]

Reimplement this function if you want to be able to delete items.

Deletes an item that is about to be removed from the collection.

This function has to reimplemented in the collection template classes, and should *only* delete item *d* if auto-delete has been enabled.

Warning: If you reimplement this function you must also reimplement the destructor and call the virtual function clear() from your destructor. This is due to the way virtual functions and destructors work in C++: Virtual functions in derived classes cannot be called from a destructor. If you do not do this, your deleteItem() function will not be called when the container is destroyed.

See also newItem() [p. 134] and setAutoDelete() [p. 134].

Item QPtrCollection::newItem (Item d) [virtual protected]

Virtual function that creates a copy of an object that is about to be inserted into the collection.

The default implementation returns the *d* pointer, i.e. no copy is made.

This function is seldom reimplemented in the collection template classes. It is not common practice to make a copy of something that is being inserted.

See also deleteItem() [p. 133].

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

QPtrDict Class Reference

The QPtrDict class is a template class that provides a dictionary based on void* keys.

```
#include <qptrdict.h>
```

Inherits QPtrCollection [p. 132].

Public Members

- **QPtrDict** (int size = 17)
- **QPtrDict** (const QPtrDict<type> & dict)
- **~QPtrDict** ()
- QPtrDict<type> & **operator=** (const QPtrDict<type> & dict)
- virtual uint **count** () const
- uint **size** () const
- bool **isEmpty** () const
- void **insert** (void * key, const type * item)
- void **replace** (void * key, const type * item)
- bool **remove** (void * key)
- type * **take** (void * key)
- type * **find** (void * key) const
- type * **operator[]** (void * key) const
- virtual void **clear** ()
- void **resize** (uint newsize)
- void **statistics** () const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Protected Members

- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item) const

Detailed Description

The QPtrDict class is a template class that provides a dictionary based on void* keys.

QPtrDict is implemented as a template class. Define a template instance QPtrDict<X> to create a dictionary that operates on pointers to X (X*).

A dictionary is a collection of key-value pairs. The key is a void* used for insertion, removal and lookup. The value is a pointer. Dictionaries provide very fast insertion and lookup.

Example:

```
QPtrDict extra;

QLineEdit *le1 = new QLineEdit( this );
le1->setText( "Simpson" );
QLineEdit *le2 = new QLineEdit( this );
le2->setText( "Homer" );
QLineEdit *le3 = new QLineEdit( this );
le3->setText( "45" );

extra.insert( le1, "Surname" );
extra.insert( le2, "Forename" );
extra.insert( le3, "Age" );

QPtrDictIterator it( extra ); // See QPtrDictIterator
for( ; it.current(); ++it )
    cout << it.current() << endl;
cout << endl;

if ( extra[le1] ) // Prints "Surname: Simpson"
    cout << extra[le1] << ": " <text() << endl;
if ( extra[le2] ) // Prints "Forename: Homer"
    cout << extra[le2] << ": " <text() << endl;

extra.remove( le1 ); // Removes le1 from the dictionary
cout <text() << endl; // Prints "Simpson"
```

In this example we use a dictionary to add an extra property (a char*) to the line edits we're using.

See QDict for full details, including the choice of dictionary size, and how deletions are handled.

See also QPtrDictIterator [p. 141], QDict [p. 77], QAsciiDict [p. 20], QIntDict [p. 94], Collection Classes [p. 9], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QPtrDict::QPtrDict (int size = 17)

Constructs a dictionary using an internal hash array with the size *size*.

Setting *size* to a suitably large prime number (equal to or greater than the expected number of entries) makes the hash distribution better and hence the lookup faster.

QPtrDict::QPtrDict (const QPtrDict<type> & dict)

Constructs a copy of *dict*.

Each item in *dict* is inserted into this dictionary. Only the pointers are copied (shallow copy).

QPtrDict::~QPtrDict ()

Removes all items from the dictionary and destroys it.

All iterators that access this dictionary will be reset.

See also setAutoDelete() [p. 134].

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QPtrDict::clear () [virtual]

Removes all items from the dictionary.

The removed items are deleted if auto-deletion is enabled.

All dictionary iterators that access this dictionary will be reset.

See also remove() [p. 138], take() [p. 139] and setAutoDelete() [p. 134].

Reimplemented from QPtrCollection [p. 133].

uint QPtrDict::count () const [virtual]

Returns the number of items in the dictionary.

See also isEmpty() [p. 138].

Reimplemented from QPtrCollection [p. 133].

type * QPtrDict::find (void * key) const

Returns the item associated with *key*, or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last item that was inserted will be found.

Equivalent to the [] operator.

See also operator[]() [p. 138].

void QPtrDict::insert (void * key, const type * item)

Inserts the *key* with the *item* into the dictionary.

The key does not have to be a unique dictionary key. If multiple items are inserted with the same key, only the last item will be visible.

Null items are not allowed.

See also replace() [p. 138].

bool QPtrDict::isEmpty () const

Returns TRUE if the dictionary is empty; otherwise returns FALSE.

See also count() [p. 137].

QPtrDict<type> & QPtrDict::operator= (const QPtrDict<type> & dict)

Assigns *dict* to this dictionary and returns a reference to this dictionary.

This dictionary is first cleared and then each item in *dict* is inserted into the dictionary. Only the pointers are copied (shallow copy), unless newItem() has been reimplemented.

type * QPtrDict::operator[] (void * key) const

Returns the item associated with *key*, or null if the key does not exist in the dictionary.

This function uses an internal hashing algorithm to optimize lookup.

If there are two or more items with equal keys, then the last item that was inserted will be found.

Equivalent to the find() function.

See also find() [p. 137].

QDataStream & QPtrDict::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a dictionary item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 140].

bool QPtrDict::remove (void * key)

Removes the item associated with *key* from the dictionary. Returns TRUE if successful, or FALSE if the key does not exist in the dictionary.

If there are two or more items with equal keys, then the last item that was inserted of will be removed.

The removed item is deleted if auto-deletion is enabled.

All dictionary iterators that refer to the removed item will be set to point to the next item in the dictionary traversal order.

See also take() [p. 139], clear() [p. 137] and setAutoDelete() [p. 134].

void QPtrDict::replace (void * key, const type * item)

If the dictionary has key *key*, this key's item is replaced with *item*. If the dictionary doesn't contain key *key*, *item* is inserted into the dictionary using key *key*.

Null items are not allowed.

Equivalent to

```
    QPtrDict dict;
```

```
        ...
    if ( dict.find( key ) )
        dict.remove( key );
    dict.insert( key, item );
```

If there are two or more items with equal keys, then the last inserted of these will be replaced.

See also insert() [p. 137].

void QPtrDict::resize (uint newsize)

Changes the size of the hash table to *newsize*. The contents of the dictionary are preserved, but all iterators on the dictionary become invalid.

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

uint QPtrDict::size () const

Returns the size of the internal hash table (as specified in the constructor).

See also count() [p. 137].

void QPtrDict::statistics () const

Debugging-only function that prints out the dictionary distribution using qDebug().

type * QPtrDict::take (void * key)

Takes the item associated with *key* out of the dictionary without deleting it (even if auto-deletion is enabled).

If there are two or more items with equal keys, then the last item that was inserted of will be removed.

Returns a pointer to the item taken out, or null if the key does not exist in the dictionary.

All dictionary iterators that refer to the taken item will be set to point to the next item in the dictionary traversal order.

See also remove() [p. 138], clear() [p. 137] and setAutoDelete() [p. 134].

QDataStream & QPtrDict::write (QDataStream & s, QPtrCollection::Item) const [virtual protected]

Writes a dictionary item to the stream *s* and returns a reference to the stream.

See also read() [p. 138].

QPtrDictIterator Class Reference

The QPtrDictIterator class provides an iterator for QPtrDict collections.

`#include <qptrdict.h>`

Public Members

- **QPtrDictIterator** (const QPtrDict<type> & dict)
- **~QPtrDictIterator** ()
- uint **count** () const
- bool **isEmpty** () const
- type * **toFirst** ()
- **operator type** * () const
- type * **current** () const
- void * **currentKey** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)

Detailed Description

The QPtrDictIterator class provides an iterator for QPtrDict collections.

QPtrDictIterator is implemented as a template class. Define a template instance QPtrDictIterator<X> to create a dictionary iterator that operates on QPtrDict<X> (dictionary of X*).

Example:

```
    QPtrDict extra;

    QLineEdit *le1 = new QLineEdit( this );
    le1->setText( "Simpson" );
    QLineEdit *le2 = new QLineEdit( this );
    le2->setText( "Homer" );
    QLineEdit *le3 = new QLineEdit( this );
    le3->setText( "45" );

    extra.insert( le1, "Surname" );
    extra.insert( le2, "Forename" );
    extra.insert( le3, "Age" );

    QPtrDictIterator it( extra );
    for( ; it.current(); ++it ) {
        QLineEdit *le = (QLineEdit)it.currentKey();
        cout << it.current() << ": " <text() << endl;
    }
    cout << endl;

    // Output (random order):
    //  Forename: Homer
    //  Age: 45
    //  Surname: Simpson
```

In the example we insert some line edits into a dictionary, then iterate over the dictionary printing the strings associated with those line edits.

Multiple iterators may independently traverse the same dictionary. A QPtrDict knows about all iterators that are operating on the dictionary. When an item is removed from the dictionary, QPtrDict updates all iterators that refer the removed item to point to the next item in the traversing order.

See also QPtrDict [p. 135], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QPtrDictIterator::QPtrDictIterator (const QPtrDict<type> & dict)

Constructs an iterator for *dict*. The current iterator item is set to point on the first item in the *dict*.

QPtrDictIterator::~QPtrDictIterator ()

Destroys the iterator.

uint QPtrDictIterator::count () const

Returns the number of items in the dictionary this iterator operates on.

See also isEmpty() [p. 142].

type * QPtrDictIterator::current () const

Returns a pointer to the current iterator item.

void * QPtrDictIterator::currentKey () const

Returns the key for the current iterator item.

bool QPtrDictIterator::isEmpty () const

Returns TRUE if the dictionary is empty; otherwise returns FALSE.

See also count() [p. 142].

QPtrDictIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QPtrDictIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QPtrDictIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the dictionary or if it was null, null is returned.

type * QPtrDictIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null.

type * QPtrDictIterator::toFirst ()

Sets the current iterator item to point to the first item in the dictionary and returns a pointer to the item. If the dictionary is empty, it sets the current item to null and returns null.

QPtrList Class Reference

The QPtrList class is a template class that provides doubly-linked lists.

`#include <qptrlist.h>`

Inherits QPtrCollection [p. 132].

Inherited by QSortedList and QStrList [p. 211].

Public Members

- **QPtrList** ()
- **QPtrList** (const QPtrList<type> & list)
- **~QPtrList** ()
- QPtrList<type> & **operator=** (const QPtrList<type> & list)
- bool **operator==** (const QPtrList<type> & list) const
- virtual uint **count** () const
- bool **isEmpty** () const
- bool **insert** (uint index, const type * item)
- void **inSort** (const type * item)
- void **prepend** (const type * item)
- void **append** (const type * item)
- bool **remove** (uint index)
- bool **remove** ()
- bool **remove** (const type * item)
- bool **removeRef** (const type * item)
- void **removeNode** (QLNode * node)
- bool **removeFirst** ()
- bool **removeLast** ()
- type * **take** (uint index)
- type * **take** ()
- type * **takeNode** (QLNode * node)
- virtual void **clear** ()
- void **sort** ()
- int **find** (const type * item)
- int **findNext** (const type * item)
- int **findRef** (const type * item)
- int **findNextRef** (const type * item)
- uint **contains** (const type * item) const
- uint **containsRef** (const type * item) const
- type * **at** (uint index)

- int **at** () const
- type * **current** () const
- QLNode * **currentNode** () const
- type * **getFirst** () const
- type * **getLast** () const
- type * **first** ()
- type * **last** ()
- type * **next** ()
- type * **prev** ()
- void **toVector** (QGVector * vec) const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Protected Members

- virtual int **compareItems** (QPtrCollection::Item item1, QPtrCollection::Item item2)
- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item item) const

Detailed Description

The QPtrList class is a template class that provides doubly-linked lists.

Define a template instance QPtrList<X> to create a list that operates on pointers to X (X*).

The list class is indexable and has a current index and a current item. The first item corresponds to index 0. The current index is -1 if the current item is null.

Items are inserted with prepend(), insert() or append(). Items are removed with remove(), removeRef(), removeFirst() and removeLast(). You can search for an item using find(), findNext(), findRef() or findNextRef(). The list can be sorted with sort(). You can count the number of occurrences of an item with contains() or containsRef(). You can get a pointer to the current item with current(), to an item at a particular index position in the list with at() or to the first or last item with getFirst() and getLast(). You can also iterate over the list with first(), last(), next() and prev() (which all update current()). The list's deletion property is set with setAutoDelete().

Example:

```
class Employee
{
public:
    Employee() : sn( 0 ) { }
    Employee( const QString& forename, const QString& surname, int salary )
        : fn( forename ), sn( surname ), sal( salary )
    { }

    void setSalary( int salary ) { sal = salary; }

    QString forename() const { return fn; }
    QString surname() const { return sn; }
```

```
        int salary() const { return sal; }

    private:
        QString fn;
        QString sn;
        int sal;
    };

    QPtrList list;
    list.setAutoDelete( TRUE ); // the list owns the objects

    list.append( new Employee("John", "Doe", 50000) );
    list.append( new Employee("Jane", "Williams", 80000) );
    list.append( new Employee("Tom", "Jones", 60000) );

    Employee *employee;
    for ( employee = list.first(); employee; employee = list.next() )
        cout <surname().latin1() << ", " <forename().latin1() << " earns " <salary() << endl;
    cout << endl;

    // very inefficient for big lists
    for ( uint i = 0; i < list.count(); ++i )
        if ( list.at(i) )
            cout << list.at( i )->surname().latin1() << endl;
```

The output is

```
    Doe, John earns 50000
    Williams, Jane earns 80000
    Jones, Tom earns 60000

    Doe
    Williams
    Jones
```

QPtrList has several member functions for traversing the list, but using a QPtrListIterator can be more practical. Multiple list iterators may traverse the same list, independently of each other and of the current list item.

In the example above we make the call setAutoDelete(TRUE). Enabling auto-deletion tells the list to delete items that are removed from the list. The default is to not delete items when they are removed but that would cause a memory leak in our example because we have no other references to the list items.

List items are stored as `void*` in an internal QLNode, which also holds pointers to the next and previous list items. The functions currentNode(), removeNode(), and takeNode() operate directly on the QLNode, but they should be used with care. The data component of the node is available through QLNode::getData().

When inserting an item into a list only the pointer is copied, not the item itself, i.e. we make a shallow copy. It is possible to make the list copy all of the item's data (deep copy) when an item is inserted. insert(), inSort() and append() call the virtual function QPtrCollection::newItem() for the item to be inserted. Inherit a list and reimplement it if you want deep copies.

When removing an item from a list, the virtual function QPtrCollection::deleteItem() is called. QPtrList's default implementation is to delete the item if auto-deletion is enabled.

The virtual function compareItems() can be reimplemented to compare two list items. This function is called from all list functions that need to compare list items, for instance remove(const type*). If you only want to deal with pointers, there are functions that compare pointers instead, for instance removeRef(const type*). These functions are somewhat faster than those that call compareItems().

The QStrList class defined in qstrlist.h is a list of `char*`. It reimplements newItem(), deleteItem() and compareItems().

See also QPtrListIterator [p. 156], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QPtrList::QPtrList ()

Constructs an empty list.

QPtrList::QPtrList (const QPtrList<type> & list)

Constructs a copy of *list*.

Each item in *list* is appended to this list. Only the pointers are copied (shallow copy).

QPtrList::~QPtrList ()

Removes all items from the list and destroys the list.

All list iterators that access this list will be reset.

See also setAutoDelete() [p. 134].

void QPtrList::append (const type * item)

Inserts the *item* at the end of the list.

The inserted item becomes the current list item. This is equivalent to `insert( count(), item )`.

The *item* must not be a null pointer.

See also insert() [p. 151], current() [p. 149] and prepend() [p. 152].

Examples: customlayout/border.cpp, customlayout/card.cpp, customlayout/flow.cpp, grapher/grapher.cpp, listviews/listviews.cpp, listviews/listviews.h and qwerty/qwerty.cpp.

type * QPtrList::at (uint index)

Returns a pointer to the item at position *index* in the list, or null if the index is out of range.

Sets the current list item to this item if *index* is valid. The valid range is `0..(count() - 1)` inclusive.

This function is very efficient. It starts scanning from the first item, last item, or current item, whichever is closest to *index*.

See also current() [p. 149].

Examples: customlayout/border.cpp, customlayout/card.cpp, customlayout/flow.cpp, dirview/dirview.cpp, fileiconview/qfileiconview.cpp, mdi/application.cpp and qwerty/qwerty.cpp.

int QPtrList::at () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the index of the current list item. The returned value is -1 if the current item is null.

See also current() [p. 149].

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

void QPtrList::clear () [virtual]

Removes all items from the list.

The removed items are deleted if auto-deletion is enabled.

All list iterators that access this list will be reset.

See also remove() [p. 152], take() [p. 154] and setAutoDelete() [p. 134].

Reimplemented from QPtrCollection [p. 133].

int QPtrList::compareItems (QPtrCollection::Item item1, QPtrCollection::Item item2) [virtual protected]

This virtual function compares two list items.

Returns:

- zero if *item1* == *item2*
- nonzero if *item1* != *item2*

This function returns *int* rather than *bool* so that reimplementations can return three values and use it to sort by:

- 0 if *item1* == *item2*
- > 0 (positive integer) if *item1* > *item2*
- < 0 (negative integer) if *item1* < *item2*

inSort() requires that compareItems() is implemented as described here.

This function should not modify the list because some const functions call compareItems().

The default implementation compares the pointers.

uint QPtrList::contains (const type * item) const

Counts and returns the number of occurrences of *item* in the list.

The compareItems() function is called when looking for the *item* in the list. If compareItems() is not reimplemented, it is more efficient to call containsRef().

This function does not affect the current list item.

See also containsRef() [p. 148] and compareItems() [p. 148].

uint QPtrList::containsRef (const type * item) const

Counts and returns the number of occurrences of *item* in the list.

Calling this function is much faster than contains() because contains() compares *item* with each list item using compareItems(). This function only compares the pointers.

This function does not affect the current list item.

See also contains() [p. 148].

uint QPtrList::count () const [virtual]

Returns the number of items in the list.

See also isEmpty() [p. 151].

Examples: customlayout/border.cpp, customlayout/card.cpp, customlayout/flow.cpp, fileiconview/qfileiconview.cpp, grapher/grapher.cpp, mdi/application.cpp and qwerty/qwerty.cpp.

Reimplemented from QPtrCollection [p. 133].

type * QPtrList::current () const

Returns a pointer to the current list item. The current item may be null (implies that the current index is -1).

See also at() [p. 147].

QLNode * QPtrList::currentNode () const

Returns a pointer to the current list node.

The node can be kept and removed later using removeNode(). The advantage is that the item can be removed directly without searching the list.

Warning: Do not call this function unless you are an expert.

See also removeNode() [p. 153], takeNode() [p. 155] and current() [p. 149].

int QPtrList::find (const type * item)

Finds the first occurrence of *item* in the list.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1, and returns -1.

The compareItems() function is called when searching for the item in the list. If compareItems() is not reimplemented, it is more efficient to call findRef().

See also findNext() [p. 149], findRef() [p. 150], compareItems() [p. 148] and current() [p. 149].

int QPtrList::findNext (const type * item)

Finds the next occurrence of *item* in the list, starting from the current list item.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1, and returns -1.

The compareItems() function is called when searching for the item in the list. If compareItems() is not reimplemented, it is more efficient to call findNextRef().

See also find() [p. 149], findNextRef() [p. 150], compareItems() [p. 148] and current() [p. 149].

int QPtrList::findNextRef (const type * item)

Finds the next occurrence of *item* in the list, starting from the current list item.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1, and returns -1.

Calling this function is much faster than findNext() because findNext() compares *item* with each list item using compareItems(). This function only compares the pointers.

See also findRef() [p. 150], findNext() [p. 149] and current() [p. 149].

int QPtrList::findRef (const type * item)

Finds the first occurrence of *item* in the list.

If the item is found, the list sets the current item to point to the found item and returns the index of this item. If the item is not found, the list sets the current item to null, the current index to -1, and returns -1.

Calling this function is much faster than find() because find() compares *item* with each list item using compareItems(). This function only compares the pointers.

See also findNextRef() [p. 150], find() [p. 149] and current() [p. 149].

type * QPtrList::first ()

Returns a pointer to the first item in the list and makes this the current list item, or null if the list is empty.

See also getFirst() [p. 150], last() [p. 151], next() [p. 151], prev() [p. 152] and current() [p. 149].

Examples: grapher/grapher.cpp, listviews/listviews.h and showimg/showimg.cpp.

type * QPtrList::getFirst () const

Returns a pointer to the first item in the list, or null if the list is empty.

This function does not affect the current list item.

See also first() [p. 150] and getLast() [p. 150].

type * QPtrList::getLast () const

Returns a pointer to the last item in the list, or null if the list is empty.

This function does not affect the current list item.

See also last() [p. 151] and getFirst() [p. 150].

void QPtrList::inSort (const type * item)

Inserts the *item* at its sorted position in the list.

The sort order depends on the virtual compareItems() function. All items must be inserted with inSort() to maintain the sorting order.

The inserted item becomes the current list item.

The *item* must not be a null pointer.

Please note that inSort() is slow. If you want to insert lots of items in a list and sort after inserting, you should use sort(). inSort() takes up to O(n) compares. That means inserting n items in your list will need O(n^2) compares whereas sort() only needs O(n*log n) for the same task. So use inSort() only if you already have a presorted list and want to insert just a few additional items.

See also insert() [p. 151], compareItems() [p. 148], current() [p. 149] and sort() [p. 154].

bool QPtrList::insert (uint index, const type * item)

Inserts the *item* at the position *index* in the list.

Returns TRUE if successful or FALSE if *index* is out of range. The valid range is 0 to count() (inclusively). The item is appended if *index* == count().

The inserted item becomes the current list item.

The *item* must not be a null pointer.

See also append() [p. 147] and current() [p. 149].

bool QPtrList::isEmpty () const

Returns TRUE if the list is empty; otherwise returns FALSE.

See also count() [p. 149].

type * QPtrList::last ()

Returns a pointer to the last item in the list and makes this the current list item, or null if the list is empty.

See also getLast() [p. 150], first() [p. 150], next() [p. 151], prev() [p. 152] and current() [p. 149].

type * QPtrList::next ()

Returns a pointer to the item succeeding the current item. Returns null if the current item is null or equal to the last item.

Makes the succeeding item current. If the current item before this function call was the last item, the current item will be set to null. If the current item was null, this function does nothing.

See also first() [p. 150], last() [p. 151], prev() [p. 152] and current() [p. 149].

Examples: grapher/grapher.cpp, listviews/listviews.h and showimg/showimg.cpp.

QPtrList<type> & QPtrList::operator= (const QPtrList<type> & list)

Assigns *list* to this list and returns a reference to this list.

This list is first cleared and then each item in *list* is appended to this list. Only the pointers are copied (shallow copy) unless newItem() has been reimplemented().

bool QPtrList::operator== (const QPtrList<type> & list) const

Compares this list with *list*. Returns TRUE if the lists contain the same data; otherwise returns FALSE.

void QPtrList::prepend (const type * item)

Inserts the *item* at the start of the list.

The inserted item becomes the current list item. This is equivalent to `insert( 0, item )`.

The *item* must not be a null pointer.

See also append() [p. 147], insert() [p. 151] and current() [p. 149].

type * QPtrList::prev ()

Returns a pointer to the item preceding the current item. Returns null if the current item is null or equal to the first item.

Makes the preceding item current. If the current item before this function call was the first item, the current item will be set to null. If the current item was null, this function does nothing.

See also first() [p. 150], last() [p. 151], next() [p. 151] and current() [p. 149].

QDataStream & QPtrList::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a list item from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 155].

bool QPtrList::remove (uint index)

Removes the item at position *index* in the list.

Returns TRUE if successful, or FALSE if *index* is out of range. The valid range is `0..(count() - 1)` inclusive.

The removed item is deleted if auto-deletion is enabled.

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item.

All list iterators that refer to the removed item will be set to point to the new current item.

See also take() [p. 154], clear() [p. 148], setAutoDelete() [p. 134], current() [p. 149] and removeRef() [p. 154].

bool QPtrList::remove ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes the current list item.

Returns TRUE if successful, or FALSE if the current item is null.

The removed item is deleted if auto-deletion is enabled.

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also take() [p. 154], clear() [p. 148], setAutoDelete() [p. 134], current() [p. 149] and removeRef() [p. 154].

bool QPtrList::remove (const type * item)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes the first occurrence of *item* from the list.

Returns TRUE if successful, or FALSE if the item could not be found in the list.

The removed item is deleted if auto-deletion is enabled.

The compareItems() function is called when searching for the item in the list. If compareItems() is not reimplemented, it is more efficient to call removeRef().

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also removeRef() [p. 154], take() [p. 154], clear() [p. 148], setAutoDelete() [p. 134], compareItems() [p. 148] and current() [p. 149].

bool QPtrList::removeFirst ()

Removes the first item from the list. Returns TRUE if successful, or FALSE if the list is empty.

The removed item is deleted if auto-deletion is enabled.

The first item in the list becomes the new current list item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also removeLast() [p. 153], setAutoDelete() [p. 134], current() [p. 149] and remove() [p. 152].

bool QPtrList::removeLast ()

Removes the last item from the list. Returns TRUE if successful, or FALSE if the list is empty.

The removed item is deleted if auto-deletion is enabled.

The last item in the list becomes the new current list item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also removeFirst() [p. 153], setAutoDelete() [p. 134] and current() [p. 149].

void QPtrList::removeNode (QLNode * node)

Removes the *node* from the list.

This node must exist in the list, otherwise the program may crash.

The removed item is deleted if auto-deletion is enabled.

The first item in the list will become the new current list item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the item succeeding this item or to the preceding item if the removed item was the last item.

Warning: Do not call this function unless you are an expert.

See also takeNode() [p. 155], currentNode() [p. 149], remove() [p. 152] and removeRef() [p. 154].

bool QPtrList::removeRef (const type * item)

Removes the first occurrence of *item* from the list.

Returns TRUE if successful, or FALSE if the item cannot be found in the list.

The removed item is deleted if auto-deletion is enabled.

The list is scanned until the pointer *item* is found. It is removed if it is found.

Equivalent to:

```
    if ( list.findRef( item ) != -1 )
        list.remove();
```

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the removed item will be set to point to the new current item.

See also remove() [p. 152], clear() [p. 148], setAutoDelete() [p. 134] and current() [p. 149].

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

void QPtrList::sort ()

Sorts the list by the result of the virtual compareItems() function.

The Heap-Sort algorithm is used for sorting. It sorts n items with O(n*log n) comparisons. This is the asymptotic optimal solution of the sorting problem.

If the items in your list support operator< and operator==, you might be better off with QSortedList because it implements the compareItems() function for you using these two operators.

See also inSort() [p. 150].

type * QPtrList::take (uint index)

Takes the item at position *index* out of the list without deleting it (even if auto-deletion is enabled).

Returns a pointer to the item taken out of the list, or null if the index is out of range. The valid range is `0..(count() - 1)` inclusive.

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the taken item will be set to point to the new current item.

See also remove() [p. 152], clear() [p. 148] and current() [p. 149].

Examples: customlayout/border.cpp, customlayout/card.cpp and customlayout/flow.cpp.

type * QPtrList::take ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Takes the current item out of the list without deleting it (even if auto-deletion is enabled). Returns a pointer to the item taken out of the list, or null if the current item is null.

The item after the removed item becomes the new current list item if the removed item is not the last item in the list. If the last item is removed, the new last item becomes the current item. The current item is set to null if the list becomes empty.

All list iterators that refer to the taken item will be set to point to the new current item.

See also remove() [p. 152], clear() [p. 148] and current() [p. 149].

type * QPtrList::takeNode (QLNode * node)

Takes the *node* out of the list without deleting its item (even if auto-deletion is enabled). Returns a pointer to the item taken out of the list.

This node must exist in the list, otherwise the program may crash.

The first item in the list becomes the new current list item.

All list iterators that refer to the taken item will be set to point to the item succeeding this item or to the preceding item if the taken item was the last item.

Warning: Do not call this function unless you are an expert.

See also removeNode() [p. 153] and currentNode() [p. 149].

void QPtrList::toVector (QGVector * vec) const

Stores all list items in the vector *vec*.

The vector must be have the same item type, otherwise the result will be undefined.

QDataStream & QPtrList::write (QDataStream & s, QPtrCollection::Item item) const [virtual protected]

Writes a list item, *item* to the stream *s* and returns a reference to the stream.

The default implementation does nothing.

See also read() [p. 152].

QPtrListIterator Class Reference

The QPtrListIterator class provides an iterator for QPtrList collections.

`#include <qptrlist.h>`

Inherited by QStrListIterator [p. 213].

Public Members

- **QPtrListIterator** (const QPtrList<type> & list)
- **~QPtrListIterator** ()
- uint **count** () const
- bool **isEmpty** () const
- bool **atFirst** () const
- bool **atLast** () const
- type * **toFirst** ()
- type * **toLast** ()
- **operator type** * () const
- type * **operator*** ()
- type * **current** () const
- type * **operator()** ()
- type * **operator++** ()
- type * **operator+=** (uint jump)
- type * **operator--** ()
- type * **operator-=** (uint jump)
- QPtrListIterator<type> & **operator=** (const QPtrListIterator<type> & it)

Detailed Description

The QPtrListIterator class provides an iterator for QPtrList collections.

Define a template instance QPtrListIterator<X> to create a list iterator that operates on QPtrList<X> (list of X*).

The following example is similar to the example in the QPtrList class documentation, but it uses QPtrListIterator. The class Employee is defined there.

```
    QPtrList list;

    list.append( new Employee("John", "Doe", 50000) );
    list.append( new Employee("Jane", "Williams", 80000) );
    list.append( new Employee("Tom", "Jones", 60000) );
    QPtrListIterator it( list );
    Employee *employee;
    while ( (employee = it.current()) != 0 ) {
        ++it;
        cout <surname().latin1() << ", " <forename().latin1() << " earns " <salary() << endl;
    }
```

The output is

```
    Doe, John earns 50000
    Williams, Jane earns 80000
    Jones, Tom earns 60000
```

Using a list iterator is a more robust way of traversing the list than using the QPtrList member functions first(), next(), current(), etc., as many iterators can traverse the same list independently.

An iterator has its own current list item and can get the next and previous list items. It doesn't modifies the list in any way.

When an item is removed from the list, all iterator that points to that item are updated to point to QPtrList::current() instead to avoid dangling references.

See also QPtrList [p. 144], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QPtrListIterator::QPtrListIterator (const QPtrList<type> & list)

Constructs an iterator for *list*. The current iterator item is set to point on the first item in the *list*.

QPtrListIterator::~QPtrListIterator ()

Destroys the iterator.

bool QPtrListIterator::atFirst () const

Returns TRUE if the current iterator item is the first list item; otherwise returns FALSE.

See also toFirst() [p. 159] and atLast() [p. 157].

bool QPtrListIterator::atLast () const

Returns TRUE if the current iterator item is the last list item; otherwise returns FALSE.

See also toLast() [p. 159] and atFirst() [p. 157].

uint QPtrListIterator::count () const

Returns the number of items in the list this iterator operates on.

See also isEmpty() [p. 158].

Example: customlayout/card.cpp.

type * QPtrListIterator::current () const

Returns a pointer to the current iterator item.

Examples: customlayout/card.cpp and customlayout/flow.cpp.

bool QPtrListIterator::isEmpty () const

Returns TRUE if the list is empty; otherwise returns FALSE.

See also count() [p. 157].

QPtrListIterator::operator type * () const

Cast operator. Returns a pointer to the current iterator item. Same as current().

type * QPtrListIterator::operator() ()

Makes the succeeding item current and returns the original current item.

If the current iterator item was the last item in the list or if it was null, null is returned.

type * QPtrListIterator::operator* ()

Asterix operator. Returns a pointer to the current iterator item. Same as current().

type * QPtrListIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns the new current item.

If the current iterator item was the last item in the list or if it was null, null is returned.

type * QPtrListIterator::operator+= (uint jump)

Sets the current item to the item *jump* positions after the current item and returns a pointer to that item.

If that item is beyond the last item or if the dictionary is empty, it sets the current item to null and returns null

type * QPtrListIterator::operator-- ()

Prefix - makes the preceding item current and returns the new current item.

If the current iterator item was the first item in the list or if it was null, null is returned.

type * QPtrListIterator::operator-= (uint jump)

Returns the item *jump* positions before the current item or null if it is beyond the first item. Makes this the current item.

QPtrListIterator<type> & QPtrListIterator::operator= (const QPtrListIterator<type> & it)

Assignment. Makes a copy of the iterator *it* and returns a reference to this iterator.

type * QPtrListIterator::toFirst ()

Sets the current iterator item to point to the first list item and returns a pointer to the item. Sets the current item to null and returns null if the list is empty.

See also toLast() [p. 159] and atFirst() [p. 157].

type * QPtrListIterator::toLast ()

Sets the current iterator item to point to the last list item and returns a pointer to the item. Sets the current item to null and returns null if the list is empty.

See also toFirst() [p. 159] and atLast() [p. 157].

QPtrQueue Class Reference

The QPtrQueue class is a template class that provides a queue.

```
#include <qptrqueue.h>
```

Public Members

- **QPtrQueue** ()
- **QPtrQueue** (const QPtrQueue<type> & queue)
- **~QPtrQueue** ()
- QPtrQueue<type> & **operator=** (const QPtrQueue<type> & queue)
- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)
- uint **count** () const
- bool **isEmpty** () const
- void **enqueue** (const type * d)
- type * **dequeue** ()
- bool **remove** ()
- void **clear** ()
- type * **head** () const
- **operator type** * () const
- type * **current** () const

Protected Members

- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item item) const

Detailed Description

The QPtrQueue class is a template class that provides a queue.

A template instance QPtrQueue<X> is a queue that operates on pointers to X (X*).

A queue is a first in, first out structure. Items are added to the tail of the queue with enqueue() and retrieved from the head with dequeue(). You can peek at the head item without dequeing it using head().

You can control the queue's deletion policy with setAutoDelete().

For compatibility with the QPtrCollection classes, current() and remove() are provided; both operate on the head().

See also QPtrList [p. 144], QPtrStack [p. 164], Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QPtrQueue::QPtrQueue ()

Creates an empty queue with autoDelete() set to FALSE.

QPtrQueue::QPtrQueue (const QPtrQueue<type> & queue)

Creates a queue from *queue*.

Only the pointers are copied; the items are not. The autoDelete() flag is set to FALSE.

QPtrQueue::~QPtrQueue ()

Destroys the queue. Items in the queue are deleted if autoDelete() is TRUE.

bool QPtrQueue::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 162].

void QPtrQueue::clear ()

Removes all items from the queue, and deletes them if autoDelete() is TRUE.

See also remove() [p. 162].

uint QPtrQueue::count () const

Returns the number of items in the queue.

See also isEmpty() [p. 162].

type * QPtrQueue::current () const

Returns a reference to the head item in the queue. The queue is not changed.

See also dequeue() [p. 161] and isEmpty() [p. 162].

type * QPtrQueue::dequeue ()

Takes the head item from the queue and returns a pointer to it.

See also enqueue() [p. 161] and count() [p. 161].

void QPtrQueue::enqueue (const type * d)

Adds item *d* to the tail of the queue.

See also count() [p. 161] and dequeue() [p. 161].

type * QPtrQueue::head () const

Returns a reference to the head item in the queue. The queue is not changed.

See also dequeue() [p. 161] and isEmpty() [p. 162].

bool QPtrQueue::isEmpty () const

Returns TRUE if the queue is empty; otherwise returns FALSE.

See also count() [p. 161], dequeue() [p. 161] and head() [p. 162].

QPtrQueue::operator type * () const

Returns a reference to the head item in the queue. The queue is not changed.

See also dequeue() [p. 161] and isEmpty() [p. 162].

QPtrQueue<type> & QPtrQueue::operator= (const QPtrQueue<type> & queue)

Assigns *queue* to this queue and returns a reference to this queue.

This queue is first cleared and then each item in *queue* is enqueued to this queue. Only the pointers are copied.

Note that the autoDelete() flag is not modified. If it it TRUE for both *queue* and this queue, deleting the two lists will cause double-deletion of the items.

QDataStream & QPtrQueue::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a queue item, *item*, from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 163].

bool QPtrQueue::remove ()

Removes the head item from the queue, and returns TRUE if there was an item or FALSE if the queue was empty.

The item is deleted if autoDelete() is TRUE.

See also head() [p. 162], isEmpty() [p. 162] and dequeue() [p. 161].

void QPtrQueue::setAutoDelete (bool enable)

Sets the queue to auto-delete its contents if *enable* is TRUE and not to delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a queue are deleted when the queue itself is deleted. This can be quite convenient if the queue has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the queue: you might find yourself with two queues deleting the same items.

See also autoDelete() [p. 161].

QDataStream & QPtrQueue::write (QDataStream & s, QPtrCollection::Item item) const [virtual protected]

Writes a queue item, *item*, to the stream *s* and returns a reference to the stream.

The default implementation does nothing.

See also read() [p. 162].

QPtrStack Class Reference

The QPtrStack class is a template class that provides a stack.

`#include <qptrstack.h>`

Public Members

- **QPtrStack** ()
- **QPtrStack** (const QPtrStack<type> & s)
- **~QPtrStack** ()
- QPtrStack<type> & **operator=** (const QPtrStack<type> & s)
- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)
- uint **count** () const
- bool **isEmpty** () const
- void **push** (const type * d)
- type * **pop** ()
- bool **remove** ()
- void **clear** ()
- type * **top** () const
- **operator type** * () const
- type * **current** () const

Protected Members

- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item item) const

Detailed Description

The QPtrStack class is a template class that provides a stack.

Define a template instance QPtrStack<X> to create a stack that operates on pointers to X, (X*).

A stack is a last in, first out (LIFO) structure. Items are added to the top of the stack with push() and retrieved from the top with pop(). Use top() to get a reference to the top element without changing the stack.

You can control the stack's deletion policy with setAutoDelete().

For compatibility with the QPtrCollection classes current() and remove() are provided; they both operate on the top().

See also QPtrList [p. 144], QPtrQueue [p. 160] and Non-GUI Classes.

Member Function Documentation

QPtrStack::QPtrStack ()

Creates an empty stack.

QPtrStack::QPtrStack (const QPtrStack<type> & s)

Creates a stack by making a shallow copy of another stack *s*.

QPtrStack::~QPtrStack ()

Destroys the stack. All items will be deleted if autoDelete() is TRUE.

bool QPtrStack::autoDelete () const

The same as QPtrCollection::autoDelete().

See also setAutoDelete() [p. 166].

void QPtrStack::clear ()

Removes all items from the stack, deleting them if autoDelete() is TRUE.

See also remove() [p. 166].

uint QPtrStack::count () const

Returns the number of items in the stack.

See also isEmpty() [p. 165].

type * QPtrStack::current () const

Returns a reference to the top item on the stack (most recently pushed). The stack is not changed.

bool QPtrStack::isEmpty () const

Returns TRUE is the stack contains no elements to be popped; otherwise returns FALSE.

QPtrStack::operator type * () const

Returns a reference to the top item on the stack (most recently pushed). The stack is not changed.

QPtrStack<type> & QPtrStack::operator= (const QPtrStack<type> & s)

Sets the contents of this stack by making a shallow copy of another stack *s*. Elements currently in this stack will be deleted if autoDelete() is TRUE.

type * QPtrStack::pop ()

Removes the top item from the stack and returns it.

void QPtrStack::push (const type * d)

Adds an element *d* to the top of the stack. Last in, first out.

QDataStream & QPtrStack::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a stack item, *item*, from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 166].

bool QPtrStack::remove ()

Removes the top item from the stack and deletes it if autoDelete() is TRUE. Returns TRUE if there was an item to pop; otherwise returns FALSE.

See also clear() [p. 165].

void QPtrStack::setAutoDelete (bool enable)

Defines whether this stack auto-deletes its contents. The same as QPtrCollection::setAutoDelete().

If *enable* is TRUE the stack auto-deletes its contents; if *enable* is FALSE the stack does not delete its contents.

See also autoDelete() [p. 165].

type * QPtrStack::top () const

Returns a reference to the top item on the stack (most recently pushed). The stack is not changed.

QDataStream & QPtrStack::write (QDataStream & s, QPtrCollection::Item item) const [virtual protected]

Writes a stack item, *item*, to the stream *s* and returns a reference to the stream.

The default implementation does nothing.

See also read() [p. 166].

QPtrVector Class Reference

The QPtrVector class is a template collection class that provides a vector (array).

`#include <qptrvector.h>`

Inherits QPtrCollection [p. 132].

Public Members

- **QPtrVector** ()
- **QPtrVector** (uint size)
- **QPtrVector** (const QPtrVector<type> & v)
- **~QPtrVector** ()
- QPtrVector<type> & **operator=** (const QPtrVector<type> & v)
- bool **operator==** (const QPtrVector<type> & v) const
- type ** **data** () const
- uint **size** () const
- virtual uint **count** () const
- bool **isEmpty** () const
- bool **isNull** () const
- bool **resize** (uint size)
- bool **insert** (uint i, const type * d)
- bool **remove** (uint i)
- type * **take** (uint i)
- virtual void **clear** ()
- bool **fill** (const type * d, int size = -1)
- void **sort** ()
- int **bsearch** (const type * d) const
- int **findRef** (const type * d, uint i = 0) const
- int **find** (const type * d, uint i = 0) const
- uint **containsRef** (const type * d) const
- uint **contains** (const type * d) const
- type * **operator[]** (int i) const
- type * **at** (uint i) const
- void **toList** (QGList * list) const

Important Inherited Members

- bool **autoDelete** () const
- void **setAutoDelete** (bool enable)

Protected Members

- virtual int **compareItems** (QPtrCollection::Item d1, QPtrCollection::Item d2)
- virtual QDataStream & **read** (QDataStream & s, QPtrCollection::Item & item)
- virtual QDataStream & **write** (QDataStream & s, QPtrCollection::Item item) const

Detailed Description

The QPtrVector class is a template collection class that provides a vector (array).

QPtrVector is implemented as a template class. Defines a template instance QPtrVector<X> to create a vector that contains pointers to X (X*).

A vector is the same as an array. The main difference between QPtrVector and QMemArray is that QPtrVector stores pointers to the elements, whereas QMemArray stores the elements themselves (i.e. QMemArray is value-based and QPtrVector is pointer-based).

Items are added to the vector using insert() or fill(). Items are removed with remove(). You can get a pointer to an item at a particular index position using at().

Unless otherwise stated, all functions that remove items from the vector will also delete the element pointed to if auto-deletion is enabled. By default, auto-deletion is disabled; see setAutoDelete(). This behaviour can be changed in a subclass by reimplementing the virtual function deleteItem().

Functions that compare items (find() and sort() for example) will do so using the virtual function compareItems(). The default implementation of this function only compares the pointer values. Reimplement compareItems() in a subclass to get searching and sorting based on the item contents. You can perform a linear search for a pointer in the vector using findRef(), or a binary search (of a sorted vector) using bsearch(). You can count the number of times an item appears in the vector with contains() or containsRef().

See also QMemArray [p. 120] and Non-GUI Classes.

Member Function Documentation

QPtrVector::QPtrVector ()

Constructs a null vector.

See also isNull() [p. 171].

QPtrVector::QPtrVector (uint size)

Constructs an vector with room for *size* items. Makes a null vector if *size* == 0.

All *size* positions in the vector are initialized to 0.

See also size() [p. 172], resize() [p. 172] and isNull() [p. 171].

QPtrVector::QPtrVector (const QPtrVector<type> & v)

Constructs a copy of *v*. Only the pointers are copied (i.e. shallow copy).

QPtrVector::~QPtrVector ()

Removes all items from the vector, and destroys the vector itself.

See also clear() [p. 169].

type * QPtrVector::at (uint i) const

Returns the item at position *i*, or 0 if there is no item at that position. *i* must be less than size().

bool QPtrCollection::autoDelete () const

Returns the setting of the auto-delete option. The default is FALSE.

See also setAutoDelete() [p. 134].

int QPtrVector::bsearch (const type * d) const

In a sorted array, finds the first occurrence of *d* using a binary search. For a sorted array, this is generally much faster than find(), which does a linear search.

Returns the position of *d*, or -1 if *d* could not be found. *d* may not be 0.

Compares items using the virtual function compareItems().

See also sort() [p. 172] and find() [p. 170].

void QPtrVector::clear () [virtual]

Removes all items from the vector, and destroys the vector itself.

The vector becomes a null vector.

See also isNull() [p. 171].

Reimplemented from QPtrCollection [p. 133].

int QPtrVector::compareItems (QPtrCollection::Item d1, QPtrCollection::Item d2) [virtual protected]

This virtual function compares two list items.

Returns:

- zero if *d1* == *d2*
- nonzero if *d1* != *d2*

This function returns *int* rather than *bool* so that reimplementations can return one of three values and use it to sort by:

- 0 if *d1* == *d2*
- > 0 (positive integer) if *d1* > *d2*
- < 0 (negative integer) if *d1* < *d2*

The sort() and bsearch() functions require that compareItems() is implemented as described here.

This function should not modify the vector because some const functions call compareItems().

uint QPtrVector::contains (const type * d) const

Returns the number of occurrences of item *d* in the vector.

Compares items using the virtual function compareItems().

See also containsRef() [p. 170].

uint QPtrVector::containsRef (const type * d) const

Returns the number of occurrences of the item pointer *d* in the vector.

This function does *not* use compareItems() to compare items.

See also findRef() [p. 171].

uint QPtrVector::count () const [virtual]

Returns the number of items in the vector. The vector is empty if count() == 0.

See also isEmpty() [p. 171] and size() [p. 172].

Reimplemented from QPtrCollection [p. 133].

type ** QPtrVector::data () const

Returns a pointer to the actual vector data, which is an array of type*.

The vector is a null vector if data() == 0 (null pointer).

See also isNull() [p. 171].

bool QPtrVector::fill (const type * d, int size = -1)

Inserts item *d* in all positions in the vector. Any existing items are removed. If *d* is 0, the vector becomes empty.

If *size* >= 0, the vector is first resized to *size*. By default, *size* is -1.

Returns TRUE if successful, or FALSE if the memory cannot be allocated (only if a resize has been requested).

See also resize() [p. 172], insert() [p. 171] and isEmpty() [p. 171].

int QPtrVector::find (const type * d, uint i = 0) const

Finds the first occurrence of item *d* in the vector using a linear search. The search starts at position *i*, which must be less than size(). *i* is by default 0; i.e. the search starts at the start of the vector.

Returns the position of *d*, or -1 if *d* could not be found.

Compares items using the virtual function compareItems().

Use the much faster bsearch() to search a sorted vector.

See also findRef() [p. 171] and bsearch() [p. 169].

int QPtrVector::findRef (const type * d, uint i = 0) const

Finds the first occurrence of the item pointer *d* in the vector using a linear search. The search starts at position *i*, which must be less than size(). *i* is by default 0; i.e. the search starts at the start of the vector.

Returns the position of *d*, or -1 if *d* could not be found.

This function does *not* use compareItems() to compare items.

Use the much faster bsearch() to search a sorted vector.

See also find() [p. 170] and bsearch() [p. 169].

bool QPtrVector::insert (uint i, const type * d)

Sets position *i* in the vector to contain the item *d*. *i* must be less than size(). Any previous element in position *i* is removed.

See also at() [p. 169].

bool QPtrVector::isEmpty () const

Returns TRUE if the vector is empty; otherwise returns FALSE.

See also count() [p. 170].

bool QPtrVector::isNull () const

Returns TRUE if the vector is null; otherwise returns FALSE.

A null vector has size() == 0 and data() == 0.

See also size() [p. 172].

QPtrVector<type> & QPtrVector::operator= (const QPtrVector<type> & v)

Assigns *v* to this vector and returns a reference to this vector.

This vector is first cleared and then all the items from *v* are copied into the vector. Only the pointers are copied (i.e. shallow copy).

See also clear() [p. 169].

bool QPtrVector::operator== (const QPtrVector<type> & v) const

Returns TRUE if this vector and *v* are equal; otherwise returns FALSE.

type * QPtrVector::operator[] (int i) const

Returns the item at position *i*, or 0 if there is no item at that position. *i* must be less than size().

Equivalent to at(*i*).

See also at() [p. 169].

QDataStream & QPtrVector::read (QDataStream & s, QPtrCollection::Item & item) [virtual protected]

Reads a vector item, *item*, from the stream *s* and returns a reference to the stream.

The default implementation sets *item* to 0.

See also write() [p. 173].

bool QPtrVector::remove (uint i)

Removes the item at position *i* in the vector, if there is one. *i* must be less than size().

Returns TRUE unless *i* is out of range.

See also take() [p. 173] and at() [p. 169].

bool QPtrVector::resize (uint size)

Resizes (expands or shrinks) the vector to *size* elements. The array becomes a null array if *size* == 0.

Any items at position *size* or beyond in the vector are removed. New positions are initialized 0.

Returns TRUE if successful, or FALSE if the memory cannot be allocated.

See also size() [p. 172] and isNull() [p. 171].

void QPtrCollection::setAutoDelete (bool enable)

Sets the collection to auto-delete its contents if *enable* is TRUE and to never delete them if *enable* is FALSE.

If auto-deleting is turned on, all the items in a collection are deleted when the collection itself is deleted. This is convenient if the collection has the only pointer to the items.

The default setting is FALSE, for safety. If you turn it on, be careful about copying the collection - you might find yourself with two collections deleting the same items.

Note that the auto-delete setting may also affect other functions in subclasses. For example, a subclass that has a remove() function will remove the item from its data structure, and if auto-delete is enabled, will also delete the item.

See also autoDelete() [p. 133].

Examples: grapher/grapher.cpp, scribble/scribble.cpp and table/bigtable/main.cpp.

uint QPtrVector::size () const

Returns the size of the vector, i.e. the number of vector positions. This is also the maximum number of items the vector can hold.

The vector is a null vector if size() == 0.

See also isNull() [p. 171], resize() [p. 172] and count() [p. 170].

void QPtrVector::sort ()

Sorts the items in ascending order. Any empty positions will be put last.

Compares items using the virtual function compareItems().

See also bsearch() [p. 169].

type * QPtrVector::take (uint i)

Returns the item at position *i* in the vector, and removes that item from the vector. *i* must be less than size(). If there is no item at position *i*, 0 is returned.

In contrast to remove(), this function does *not* call deleteItem() for the removed item.

See also remove() [p. 172] and at() [p. 169].

void QPtrVector::toList (QGList * list) const

Copies all items in this vector to the list *list*. *list* is first cleared and then all items are appended to *list*.

See also QPtrList [p. 144], QPtrStack [p. 164] and QPtrQueue [p. 160].

QDataStream & QPtrVector::write (QDataStream & s, QPtrCollection::Item item) const [virtual protected]

Writes a vector item, *item*, to the stream *s* and returns a reference to the stream.

The default implementation does nothing.

See also read() [p. 172].

QStrIList Class Reference

The QStrIList class provides a doubly-linked list of char* with case-insensitive comparison.

`#include <qstrlist.h>`

Inherits QStrList [p. 211].

Public Members

- **QStrIList** (bool deepCopies = TRUE)
- **~QStrIList** ()

Detailed Description

The QStrIList class provides a doubly-linked list of char* with case-insensitive comparison.

This class is a QPtrList<char> instance (a list of char*).

QStrIList is identical to QStrList except that the virtual compareItems() function is reimplemented to compare strings case-insensitively. The inSort() function inserts strings in a sorted order. In general it is fastest to insert the strings as they come and sort() at the end; inSort() is useful when you just have to add a few extra strings to an already sorted list.

The QStrListIterator class works for QStrIList.

See also Collection Classes [p. 9] and Non-GUI Classes.

Member Function Documentation

QStrIList::QStrIList (bool deepCopies = TRUE)

Constructs a list of strings. Will make deep copies of all inserted strings if *deepCopies* is TRUE, or use shallow copies if *deepCopies* is FALSE.

QStrIList::~QStrIList ()

Destroys the list. All strings are removed.

QString Class Reference

The QString class provides an abstraction of Unicode text and the classic C null-terminated char array.

`#include <qstring.h>`

Public Members

- **QString** ()
- **QString** (QChar ch)
- **QString** (const QString & s)
- **QString** (const QByteArray & ba)
- **QString** (const QChar * unicode, uint length)
- **QString** (const char * str)
- **~QString** ()
- QString & **operator=** (const QString & s)
- QString & **operator=** (const char * str)
- QString & **operator=** (const QCString & cs)
- QString & **operator=** (QChar c)
- QString & **operator=** (char c)
- bool **isNull** () const
- bool **isEmpty** () const
- uint **length** () const
- void **truncate** (uint newLen)
- QString & **fill** (QChar c, int len = -1)
- QString copy () const *(obsolete)*
- QString **arg** (long a, int fieldwidth = 0, int base = 10) const
- QString **arg** (ulong a, int fieldwidth = 0, int base = 10) const
- QString **arg** (int a, int fieldwidth = 0, int base = 10) const
- QString **arg** (uint a, int fieldwidth = 0, int base = 10) const
- QString **arg** (short a, int fieldwidth = 0, int base = 10) const
- QString **arg** (ushort a, int fieldwidth = 0, int base = 10) const
- QString **arg** (char a, int fieldwidth = 0) const
- QString **arg** (QChar a, int fieldwidth = 0) const
- QString **arg** (const QString & a, int fieldwidth = 0) const
- QString **arg** (double a, int fieldwidth = 0, char fmt = 'g', int prec = -1) const
- QString & **sprintf** (const char * cformat, ...)
- int **find** (QChar c, int index = 0, bool cs = TRUE) const
- int **find** (char c, int index = 0, bool cs = TRUE) const
- int **find** (const QString & str, int index = 0, bool cs = TRUE) const
- int **find** (const QRegExp & rx, int index = 0) const

- int **find** (const char * str, int index = 0) const
- int **findRev** (QChar c, int index = -1, bool cs = TRUE) const
- int **findRev** (char c, int index = -1, bool cs = TRUE) const
- int **findRev** (const QString & str, int index = -1, bool cs = TRUE) const
- int **findRev** (const QRegExp & rx, int index = -1) const
- int **findRev** (const char * str, int index = -1) const
- int **contains** (QChar c, bool cs = TRUE) const
- int **contains** (char c, bool cs = TRUE) const
- int **contains** (const char * str, bool cs = TRUE) const
- int **contains** (const QString & str, bool cs = TRUE) const
- int **contains** (const QRegExp & rx) const
- enum **SectionFlags** { SectionDefault = 0x00, SectionSkipEmpty = 0x01, SectionIncludeLeadingSep = 0x02, SectionIncludeTrailingSep = 0x04, SectionCaseInsensitiveSeps = 0x08 }
- QString **section** (QChar sep, int start, int end = 0xffffffff, int flags = SectionDefault) const
- QString **section** (char sep, int start, int end = 0xffffffff, int flags = SectionDefault) const
- QString **section** (const char * sep, int start, int end = 0xffffffff, int flags = SectionDefault) const
- QString **section** (const QString & sep, int start, int end = 0xffffffff, int flags = SectionDefault) const
- QString **section** (const QRegExp & reg, int start, int end = 0xffffffff, int flags = SectionDefault) const
- QString **left** (uint len) const
- QString **right** (uint len) const
- QString **mid** (uint index, uint len = 0xffffffff) const
- QString **leftJustify** (uint width, QChar fill = ' ', bool truncate = FALSE) const
- QString **rightJustify** (uint width, QChar fill = ' ', bool truncate = FALSE) const
- QString **lower** () const
- QString **upper** () const
- QString **stripWhiteSpace** () const
- QString **simplifyWhiteSpace** () const
- QString & **insert** (uint index, const QString & s)
- QString & **insert** (uint index, const QChar * s, uint len)
- QString & **insert** (uint index, QChar c)
- QString & **insert** (uint index, char c)
- QString & **append** (char ch)
- QString & **append** (QChar ch)
- QString & **append** (const QString & str)
- QString & **prepend** (char ch)
- QString & **prepend** (QChar ch)
- QString & **prepend** (const QString & s)
- QString & **remove** (uint index, uint len)
- QString & **replace** (uint index, uint len, const QString & s)
- QString & **replace** (uint index, uint len, const QChar * s, uint slen)
- QString & **replace** (const QRegExp & rx, const QString & str)
- short **toShort** (bool * ok = 0, int base = 10) const
- ushort **toUShort** (bool * ok = 0, int base = 10) const
- int **toInt** (bool * ok = 0, int base = 10) const
- uint **toUInt** (bool * ok = 0, int base = 10) const
- long **toLong** (bool * ok = 0, int base = 10) const
- ulong **toULong** (bool * ok = 0, int base = 10) const
- float **toFloat** (bool * ok = 0) const
- double **toDouble** (bool * ok = 0) const
- QString & **setNum** (short n, int base = 10)

- QString & **setNum** (ushort n, int base = 10)
- QString & **setNum** (int n, int base = 10)
- QString & **setNum** (uint n, int base = 10)
- QString & **setNum** (long n, int base = 10)
- QString & **setNum** (ulong n, int base = 10)
- QString & **setNum** (float n, char f = 'g', int prec = 6)
- QString & **setNum** (double n, char f = 'g', int prec = 6)
- void setExpand (uint index, QChar c) *(obsolete)*
- QString & **operator+=** (const QString & str)
- QString & **operator+=** (QChar c)
- QString & **operator+=** (char c)
- QChar **at** (uint i) const
- QChar **operator[]** (int i) const
- QCharRef **at** (uint i)
- QCharRef **operator[]** (int i)
- QChar **constref** (uint i) const
- QChar & **ref** (uint i)
- const QChar * **unicode** () const
- const char * ascii () const *(obsolete)*
- const char * **latin1** () const
- QCString **utf8** () const
- QCString **local8Bit** () const
- bool **operator!** () const
- **operator const char *** () const
- QString & **setUnicode** (const QChar * unicode, uint len)
- QString & **setUnicodeCodes** (const ushort * unicode_as_ushorts, uint len)
- QString & **setLatin1** (const char * str, int len = -1)
- int **compare** (const QString & s) const
- int **localeAwareCompare** (const QString & s) const
- void **compose** ()
- const char * data () const *(obsolete)*
- bool **startsWith** (const QString & s) const
- bool **endsWith** (const QString & s) const
- void **setLength** (uint newLen)

Static Public Members

- QString **number** (long n, int base = 10)
- QString **number** (ulong n, int base = 10)
- QString **number** (int n, int base = 10)
- QString **number** (uint n, int base = 10)
- QString **number** (double n, char f = 'g', int prec = 6)
- QString **fromLatin1** (const char * chars, int len = -1)
- QString **fromUtf8** (const char * utf8, int len = -1)
- QString **fromLocal8Bit** (const char * local8Bit, int len = -1)
- int **compare** (const QString & s1, const QString & s2)
- int **localeAwareCompare** (const QString & s1, const QString & s2)

Related Functions

- bool **operator==** (const QString & s1, const QString & s2)
- bool **operator==** (const QString & s1, const char * s2)
- bool **operator==** (const char * s1, const QString & s2)
- bool **operator!=** (const QString & s1, const QString & s2)
- bool **operator!=** (const QString & s1, const char * s2)
- bool **operator!=** (const char * s1, const QString & s2)
- bool **operator<** (const QString & s1, const char * s2)
- bool **operator<** (const char * s1, const QString & s2)
- bool **operator<=** (const QString & s1, const char * s2)
- bool **operator<=** (const char * s1, const QString & s2)
- bool **operator>** (const QString & s1, const char * s2)
- bool **operator>** (const char * s1, const QString & s2)
- bool **operator>=** (const QString & s1, const char * s2)
- bool **operator>=** (const char * s1, const QString & s2)
- const QString **operator+** (const QString & s1, const QString & s2)
- const QString **operator+** (const QString & s1, const char * s2)
- const QString **operator+** (const char * s1, const QString & s2)
- const QString **operator+** (const QString & s, char c)
- const QString **operator+** (char c, const QString & s)
- QDataStream & **operator<<** (QDataStream & s, const QString & str)
- QDataStream & **operator>>** (QDataStream & s, QString & str)

Detailed Description

The QString class provides an abstraction of Unicode text and the classic C null-terminated char array.

QString uses implicit sharing, which makes it very efficient and easy to use.

In all of the QString methods that take `const char *` parameters, the `const char *` is interpreted as a classic C-style 0-terminated ASCII string. It is legal for the `const char *` parameter to be 0. If the `const char *` is not 0-terminated, the results are undefined. Functions that copy classic C strings into a QString will not copy the terminating 0 character. The QChar array of the QString (as returned by unicode()) is generally not terminated by a null.

A QString that has not been assigned to anything is *null*, i.e., both the length and data pointer is 0. A QString that references the empty string ("", a single '\0' char) is *empty*. Both null and empty QStrings are legal parameters to the methods. Assigning `(const char *) 0` to QString gives a null QString. For convenience, QString::null is a null QString.

Note that if you find that you are mixing usage of QCString, QString, and QByteArray, this causes lots of unnecessary copying and might indicate that the true nature of the data you are dealing with is uncertain. If the data is 0-terminated 8-bit data, use QCString; if it is unterminated (i.e. contains 0s) 8-bit data, use QByteArray; if it is text, use QString.

Lists of strings are handled by the QStringList class. You can split a string into a list of strings using QStringList::split(), and join a list of strings into a single string with an optional separator using QStringList::join(). You can obtain a list of strings from a string list that contain a particular substring or that match a particular regex using QStringList::grep().

Note for C programmers

Due to C++'s type system and the fact that QString is implicitly shared, QStrings may be treated like ints or other simple base types. For example:

```
QString boolToString( bool b )
{
    QString result;
    if ( b )
        result = "True";
    else
        result = "False";
    return result;
}
```

The variable, result, is an auto variable allocated on the stack. When return is called, because we're returning by value, The copy constructor is called and a copy of the string is returned. (No actual copying takes place thanks to the implicit sharing, see below.)

Throughout Qt's source code you will encounter QString usages like this:

```
QString func( const QString& input )
{
    QString output = input;
    // process output
    return output;
}
```

The 'copying' of input to output is almost as fast as copying a pointer because behind the scenes copying is achieved by incrementing a reference count. QString operates on a copy-on-write basis, only copying if an instance is actually changed.

See also QChar [p. 45], QCString [p. 58], QByteArray [p. 37], QConstString [p. 56], Implicitly and Explicitly Shared Classes, Text Related Classes and Non-GUI Classes.

Member Type Documentation

QString::SectionFlags

- `QString::SectionDefault` - Empty fields are counted, leading and trailing separators are not included, and the separator is compared case sensitively.
- `QString::SectionSkipEmpty` - Treat empty fields as if they don't exist, i.e. they are not considered as far as *start* and *end* are oncerned.
- `QString::SectionIncludeLeadingSep` - Include the leading separator (if any) in the result string.
- `QString::SectionIncludeTrailingSep` - Include the trailing separator (if any) in the result string.
- `QString::SectionCaseInsensitiveSeps` - Compare the separator case-insensitively.

Any of the last four values can be OR-ed together to form a flag.

See also section() [p. 196].

Member Function Documentation

QString::QString ()

Constructs a null string. This is a string that has not been assigned to anything, i.e. both the length and data pointer is 0.

See also isNull() [p. 189].

QString::QString (QChar ch)

Constructs a string giving it a length of one character, assigning it the character *ch*.

QString::QString (const QString & s)

Constructs an implicitly shared copy of *s*. This is instantaneous, since reference counting is used.

QString::QString (const QByteArray & ba)

Constructs a string that is a deep copy of *ba* interpreted as a classic C string.

QString::QString (const QChar * unicode, uint length)

Constructs a string that is a deep copy of the first *length* characters in the QChar array.

If *unicode* and *length* are 0, then a null string is created.

If only *unicode* is 0, the string is empty but has *length* characters of space preallocated - QString expands automatically anyway, but this may speed up some cases a little. We recommend using the plain constructor and setLength() for this purpose since it will result in more readable code.

See also isNull() [p. 189] and setLength() [p. 198].

QString::QString (const char * str)

Constructs a string that is a deep copy of *str*, interpreted as a classic C string.

If *str* is 0, then a null string is created.

This is a cast constructor, but it is perfectly safe: converting a Latin1 const char* to QString preserves all the information. You can disable this constructor by defining QT_NO_CAST_ASCII when you compile your applications. You can also make QString objects by using setLatin1(), fromLatin1(), fromLocal8Bit(), and fromUtf8(). Or whatever encoding is appropriate for the 8-bit data you have.

See also isNull() [p. 189].

QString::~QString ()

Destroys the string and frees the "real" string if this is the last copy of that string.

QString & QString::append (const QString & str)

Appends *str* to the string and returns a reference to the result.

```
    string = "Test";
    string.append( "ing" );        // string == "Testing"
```

Equivalent to operator+=().

Example: dirview/dirview.cpp.

QString & QString::append (char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends character *ch* to the string and returns a reference to the result.

Equivalent to operator+=().

QString & QString::append (QChar ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends character *ch* to the string and returns a reference to the result.

Equivalent to operator+=().

QString QString::arg (const QString & a, int fieldwidth = 0) const

This function will return a string that replaces the lowest occurrence of `%i` (`i` being '1' or '2' or ... or '9') with *a*.

The *fieldwidth* value specifies the minimum amount of space that *a* is padded to. A positive value will produce right-aligned text, whereas a negative value will produce left-aligned text.

```
QString firstName( "Joe" );
QString lastName( "Bloggs" );
QString fullName;
fullName = QString( "First name is '%1', last name is '%2'" )
           .arg( firstName )
           .arg( lastName );

// fullName == First name is 'Joe', last name is 'Bloggs'
```

Warning: If you use arg() to construct "real" sentences like the one shown in the examples above, then this may cause problems with translation (when you use the tr() function).

If there is no `%i` pattern, a warning message (qWarning()) is outputted and the text is appended at the end of the string. This is error recovery done by the function and should not occur in correct code.

See also QObject::tr() [Additional Functionality with Qt].

QString QString::arg (long a, int fieldwidth = 0, int base = 10) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The *fieldwidth* value specifies the minimum amount of space that *a* is padded to. A positive value will produce a right-aligned number, whereas a negative value will produce a left-aligned number.

a is expressed in base *base*, which is 10 by default and must be between 2 and 36.

```
QString str;
str = QString( "Decimal 63 is %1 in hexadecimal" )
      .arg( 63, 0, 16 );
// str == "Decimal 63 is 3f in hexadecimal"
```

QString QString::arg (ulong a, int fieldwidth = 0, int base = 10) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a is expressed in base *base*, which is 10 by default and must be between 2 and 36.

QString QString::arg (int a, int fieldwidth = 0, int base = 10) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a is expressed in base *base*, which is 10 by default and must be between 2 and 36.

QString QString::arg (uint a, int fieldwidth = 0, int base = 10) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a is expressed in base *base*, which is 10 by default and must be between 2 and 36.

QString QString::arg (short a, int fieldwidth = 0, int base = 10) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a is expressed in base *base*, which is 10 by default and must be between 2 and 36.

QString QString::arg (ushort a, int fieldwidth = 0, int base = 10) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a is expressed in base *base*, which is 10 by default and must be between 2 and 36.

QString QString::arg (char a, int fieldwidth = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

a is assumed to be in the Latin1 character set.

QString QString::arg (QChar a, int fieldwidth = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QString QString::arg (double a, int fieldwidth = 0, char fmt = ’g’, int prec = -1) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Argument *a* is formatted according to the *fmt* format specified, which is `g` by default and can be any of the following:

- `e` - format as [-]9.9e[+|-]999
- `E` - format as [-]9.9E[+|-]999
- `f` - format as [-]9.9
- `g` - use `e` or `f` format, whichever is the most concise
- `G` - use `E` or `f` format, whichever is the most concise

In all cases the number of digits after the decimal point is equal to the precision specified in *prec*.

```
double d = 12.34;
QString ds = QString( "'E' format, precision 3, gives %1" )
             .arg( d, 0, 'E', 3 );
// ds == "1.234E+001"
```

const char * QString::ascii () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

This function simply calls latin1() and returns the result.

Example: network/networkprotocol/nntp.cpp.

QChar QString::at (uint i) const

Returns the character at index *i*, or 0 if *i* is beyond the length of the string.

```
const QString string( "abcdefgh" );
QChar ch = string.at( 4 );
// ch == 'e'
```

If the QString is not const (i.e. const QString) or const& (i.e. const QString &), then the non-const overload of at() will be used instead.

QCharRef QString::at (uint i)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The function returns a reference to the character at index *i*. The resulting reference can then be assigned to, or used immediately, but it will become invalid once further modifications are made to the original string.

If *i* is beyond the length of the string then the string is expanded with QChar::null.

int QString::compare (const QString & s1, const QString & s2) [static]

Lexically compares *s1* with *s2* and returns an integer less than, equal to, or greater than zero if *s1* is less than, equal to, or greater than *s2*.

The comparison is based exclusively on the numeric Unicode values of the characters and is very fast, but is not what a human would expect. Consider sorting user-interface strings with QString::localeAwareCompare().

```
int a = QString::compare( "def", "abc" );   // a > 0
int b = QString::compare( "abc", "def" );   // b < 0
int c = QString::compare(" abc", "abc" );   // c == 0
```

int QString::compare (const QString & s) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Lexically compares this string with *s* and returns an integer less than, equal to, or greater than zero if it is less than, equal to, or greater than *s*.

void QString::compose ()

Note that this function is not supported in Qt 3.0 and is merely for experimental and illustrative purposes. It is mainly of interest to those experimenting with Arabic and other composition-rich texts.

Applies possible ligatures to a QString. Useful when composition-rich text requires rendering with glyph-poor fonts, but it also makes compositions such as QChar(0x0041) ('A') and QChar(0x0308) (Unicode accent diaresis), giving QChar(0x00c4) (German A Umlaut).

QChar QString::constref (uint i) const

Returns the QChar at index *i* by value.

Equivalent to at(*i*).

See also ref() [p. 195].

int QString::contains (QChar c, bool cs = TRUE) const

Returns the number of times the character *c* occurs in the string.

If *cs* is TRUE then the match is case sensitive. If *cs* is FALSE, then the match is case insensitive.

```
    QString string( "Trolltech and Qt" );
    int i = string.contains( 't', FALSE );  // i == 3
```

Examples: fileiconview/qfileiconview.cpp and mdi/application.cpp.

int QString::contains (char c, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

int QString::contains (const char * str, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the number of times the string *str* occurs in the string.

If *cs* is TRUE then the match is case sensitive. If *cs* is FALSE, then the match is case insensitive.

int QString::contains (const QString & str, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the number of times *str* occurs in the string.

The match is case sensitive if *cs* is TRUE or case insensitive if *cs* if FALSE.

This function counts overlapping strings, so in the example below, there are two instances of "ana" in "bananas".

```
    QString str( "bananas" );
    int i = str.contains( "ana" );  // i == 2
```

See also findRev() [p. 186].

int QString::contains (const QRegExp & rx) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns the number of times the regexp, *rx*, occurs in the string.

This function counts overlapping occurrences, so in the example below, there are four instances of "ana" or "ama".

```
    QString str = "banana and panama";
    QRegExp rxp = QRegExp( "a[nm]a", TRUE, FALSE );
    int i = str.contains( rxp );    // i == 4
```

See also find() [p. 185] and findRev() [p. 186].

QString QString::copy () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

In Qt 2.0 and later, all calls to this function are needless. Just remove them.

const char * QString::data () const

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Returns a pointer to a 0-terminated classic C string.

In Qt 1.x, this returned a char* allowing direct manipulation of the string as a sequence of bytes. In Qt 2.x where QString is a Unicode string, char* conversion constructs a temporary string, and hence direct character operations are meaningless.

bool QString::endsWith (const QString & s) const

Returns TRUE if the string ends with *s*; otherwise it returns FALSE.

See also startsWith() [p. 201].

QString & QString::fill (QChar c, int len = -1)

Fills the string with *len* characters of value *c*, and returns a reference to the string.

If *len* is negative (the default), the current string length is used.

```
    QString str;
    str.fill( 'g', 5 );      // string == "gggggg"
```

int QString::find (const QRegExp & rx, int index = 0) const

Finds the first occurrence of the constant regular expression *rx*, starting at position *index*. If *index* is -1, the search starts at the last character; if -2, at the next to last character and so on. (See findRev() for searching backwards.)

Returns the position of the first occurrence of *rx* or -1 if *rx* was not found.

```
    QString string( "bananas" );
    int i = string.find( QRegExp("an"), 0 );    // i == 1
```

See also findRev() [p. 186], replace() [p. 195] and contains() [p. 184].

Example: network/mail/smtp.cpp.

int QString::find (QChar c, int index = 0, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the character *c*, starting at position *index*. If *index* is -1, the search starts at the last character; if -2, at the next to last character and so on. (See findRev() for searching backwards.)

If *cs* is TRUE, then the search is case sensitive. If *cs* is FALSE, then the search is case insensitive.

Returns the position of *c* or -1 if *c* could not be found.

int QString::find (char c, int index = 0, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Find character *c* starting from position *index*. If *cs* is TRUE then the match is case sensitive. If *cs* is FALSE, then the match is case insensitive.

int QString::find (const QString & str, int index = 0, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the string *str*, starting at position *index*. If *index* is -1, the search starts at the last character, if it is -2, at the next to last character and so on. (See findRev() for searching backwards.)

The search is case sensitive if *cs* is TRUE or case insensitive if *cs* is FALSE.

Returns the position of *str* or -1 if *str* could not be found.

int QString::find (const char * str, int index = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Equivalent to find(QString(*str*), *index*).

int QString::findRev (const char * str, int index = -1) const

Equivalent to findRev(QString(*str*), *index*).

int QString::findRev (QChar c, int index = -1, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the character *c*, starting at position *index* and searching backwards. If the index is -1, the search starts at the last character, if it is -2, at the next to last character and so on.

Returns the position of *c* or -1 if *c* could not be found.

If *cs* is TRUE then the search is case sensitive. If *cs* is FALSE then the search is case insensitive.

```
    QString string( "bananas" );
    int i = string.findRev( 'a' );      // i == 5
```

int QString::findRev (char c, int index = -1, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Find character *c* starting from position *index* and working backwards. If *cs* is TRUE then the match is case sensitive. If *cs* is FALSE, then the match is case insensitive.

int QString::findRev (const QString & str, int index = -1, bool cs = TRUE) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the string *str*, starting at position *index* and searching backwards. If the index is -1, the search starts at the last character, if it is -2, at the next to last character and so on.

Returns the position of *str* or -1 if *str* could not be found.

If *cs* is TRUE then the search is case sensitive. If *cs* is FALSE then the search is case insensitive.

```
    QString string("bananas");
    int i = string.findRev( "ana" );      // i == 3
```

int QString::findRev (const QRegExp & rx, int index = -1) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of the regexp *rx*, starting at position *index* and searching backwards. If the index is -1, the search starts at the last character, if it is -2, at the next to last character and so on. (See findRev() for searching backwards.)

Returns the position of *rx* or -1 if *rx* could not be found.

```
    QString string( "bananas" );
    int i = string.findRev( QRegExp("an") );      // i == 3
```

See also find() [p. 185].

QString QString::fromLatin1 (const char * chars, int len = -1) [static]

Returns the unicode string decoded from the first *len* characters of *chars*, ignoring the rest of *chars*. If *len* is -1 then the length of *chars* is used. If *len* is bigger than the length of *chars* then it will use the length of *chars*.

This is the same as the QString(const char*) constructor, but you can make that constructor invisible if you compile with the define QT_NO_CAST_ASCII, in which case you can explicitly create a QString from Latin-1 text using this function.

```
    QString str = QString::fromLatin1( "123456789", 5 );
    // str == "12345"
```

Examples: listbox/listbox.cpp and network/mail/smtp.cpp.

QString QString::fromLocal8Bit (const char * local8Bit, int len = -1) [static]

Returns the unicode string decoded from the first *len* characters of *local8Bit*, ignoring the rest of *local8Bit*. If *len* is -1 then the length of *local8Bit* is used. If *len* is bigger than the length of *local8Bit* then it will use the length of *local8Bit*.

```
    QString str = QString::fromLocal8Bit( "123456789", 5 );
    // str == "12345"
```

local8Bit is assumed to be encoded in a locale-specific format.

See QTextCodec for more diverse coding/decoding of Unicode strings.

QString QString::fromUtf8 (const char * utf8, int len = -1) [static]

Returns the unicode string decoded from the first *len* characters of *utf8*, ignoring the rest of *utf8*. If *len* is -1 then the length of *utf8* is used. If *len* is bigger than the length of *utf8* then it will use the length of *utf8*.

```
    QString str = QString::fromUtf8( "123456789", 5 );
    // str == "12345"
```

See QTextCodec for more diverse coding/decoding of Unicode strings.

Example: fonts/simple-qfont-demo/viewer.cpp.

QString & QString::insert (uint index, const QString & s)

Inserts *s* into the string before position *index*.

If *index* is beyond the end of the string, the string is extended with spaces to length *index* and *s* is then appended and returns a reference to the string.

```
    QString string( "I like fish" );
    str = string.insert( 2, "don’t " );
    // str == "I don’t like fish"
```

See also remove() [p. 195] and replace() [p. 195].

Example: xform/xform.cpp.

QString & QString::insert (uint index, const QChar * s, uint len)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts the character in *s* into the string before the position *index len* number of times and returns a reference to the string.

QString & QString::insert (uint index, QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Insert *c* into the string at (before) position *index* and returns a reference to the string.

If *index* is beyond the end of the string, the string is extended with spaces (ASCII 32) to length *index* and *c* is then appended.

QString & QString::insert (uint index, char c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Insert character *c* at position *index*.

bool QString::isEmpty () const

Returns TRUE if the string is empty, i.e., if length() == 0. Thus, null strings are empty strings.

```
QString a( "" );
a.isEmpty();        // TRUE
a.isNull();         // FALSE

QString b;
b.isEmpty();        // TRUE
b.isNull();         // TRUE
```

See also isNull() [p. 189] and length() [p. 190].

Examples: addressbook/mainwindow.cpp, hello/main.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp, network/networkprotocol/nntp.cpp, qmag/qmag.cpp and qwerty/qwerty.cpp.

bool QString::isNull () const

Returns TRUE if the string is null. A null string is always empty.

```
QString a;          // a.unicode() == 0, a.length() == 0
a.isNull();         // TRUE, because a.unicode() == 0
a.isEmpty();        // TRUE
```

See also isEmpty() [p. 189] and length() [p. 190].

Examples: i18n/main.cpp and qdir/qdir.cpp.

const char * QString::latin1 () const

Returns a Latin-1 representation of the string. Note that the returned value is undefined if the string contains non-Latin-1 characters. If you want to convert strings into formats other than Unicode, see the QTextCodec classes.

This function is mainly useful for boot-strapping legacy code to use Unicode.

The result remains valid so long as one unmodified copy of the source string exists.

See also utf8() [p. 203] and local8Bit() [p. 190].

Examples: fileiconview/qfileiconview.cpp and network/networkprotocol/nntp.cpp.

QString QString::left (uint len) const

Returns a substring that contains the *len* leftmost characters of the string.

The whole string is returned if *len* exceeds the length of the string.

```
QString s = "Pineapple";
QString t = s.left( 4 );    // t == "Pine"
```

See also right() [p. 196], mid() [p. 191] and isEmpty() [p. 189].

QString QString::leftJustify (uint width, QChar fill = ' ', bool truncate = FALSE) const

Returns a string of length *width* that contains this string padded by the *fill* character.

If *truncate* is FALSE and the length of the string is more than *width*, then the returned string is a copy of the string.

If *truncate* is TRUE and the length of the string is more than *width*, then any characters in a copy of the string after length *width* are removed, and the copy is returned.

```
    QString s( "apple" );
    QString t = s.leftJustify( 8, '.' );        // t == "apple..."
```

See also rightJustify() [p. 196].

uint QString::length () const

Returns the length of the string.

Null strings and empty strings have zero length.

See also isNull() [p. 189] and isEmpty() [p. 189].

Examples: fileiconview/qfileiconview.cpp, network/networkprotocol/nntp.cpp and rot13/rot13.cpp.

QCString QString::local8Bit () const

Returns the string encoded in a locale-specific format. On X11, this is the QTextCodec::codecForLocale(). On Windows, it is a system-defined encoding.

See QTextCodec for more diverse coding/decoding of Unicode strings.

See also QString::fromLocal8Bit() [p. 188], latin1() [p. 189] and utf8() [p. 203].

int QString::localeAwareCompare (const QString & s1, const QString & s2) [static]

Compares *s1* with *s2* and returns an integer less than, equal to, or greater than zero if *s1* is less than, equal to, or greater than *s2*.

The comparison is performed in a locale- and also platform-dependent manner. Use this function to present sorted lists of strings to the user.

Bugs and limitations:

- This function is only implemented on Windows. Elsewhere, it is a synonym for QString::compare().

See also QString::compare() [p. 183] and QTextCodec::locale() [Accessibility and Internationalization with Qt].

int QString::localeAwareCompare (const QString & s) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Compares this string with *s*.

QString QString::lower () const

Returns a string that is the string converted to lowercase.

```
    QString string( "TROlltECH" );
    str = string.lower();   // str == "trolltech"
```

See also upper() [p. 203].

Example: scribble/scribble.cpp.

QString QString::mid (uint index, uint len = 0xffffffff) const

Returns a string that contains the *len* characters of this string, starting at position *index*.

Returns a null string if the string is empty or *index* is out of range. Returns the whole string from *index* if *index+len* exceeds the length of the string.

```
    QString s( "Five pineapples" );
    QString t = s.mid( 5, 4 );                  // t == "pine"
```

See also left() [p. 189] and right() [p. 196].

Examples: network/mail/smtp.cpp and qmag/qmag.cpp.

QString QString::number (long n, int base = 10) [static]

A convenience function that returns a string equivilant of the number *n* to base *base*, which is 10 by default and must be between 2 and 36.

```
    long a = 63;
    QString str = QString::number( a, 16 );             // str == "3f"
    QString str = QString::number( a, 16 ).upper();     // str == "3F"
```

See also setNum() [p. 199].

Examples: action/application.cpp, application/application.cpp, fonts/simple-qfont-demo/viewer.cpp, helpviewer/helpwindow.cpp, mdi/application.cpp and sql/overview/extract/main.cpp.

QString QString::number (ulong n, int base = 10) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

See also setNum() [p. 199].

QString QString::number (int n, int base = 10) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

See also setNum() [p. 199].

QString QString::number (uint n, int base = 10) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

A convenience factory function that returns a string representation of the number *n* to the base *base*, which is 10 by default and must be between 2 and 36.

See also setNum() [p. 199].

QString QString::number (double n, char f = ’g’, int prec = 6) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Argument *n* is formatted according to the *f* format specified, which is `g` by default, and can be any of the following:

- `e` - format as [-]9.9e[+|-]999
- `E` - format as [-]9.9E[+|-]999
- `f` - format as [-]9.9
- `g` - use `e` or `f` format, whichever is the most concise
- `G` - use `E` or `f` format, whichever is the most concise

In all cases the number of digits after the decimal point is equal to the precision specified in *prec*.

```
    double d = 12.34;
    QString ds = QString( "'E' format, precision 3, gives %1" )
                    .arg( d, 0, 'E', 3 );
    // ds == "1.234E+001"
```

See also setNum() [p. 199].

QString::operator const char * () const

Returns latin1(). Be sure to see the warnings documented there. Note that for new code which you wish to be strictly Unicode-clean, you can define the macro QT_NO_ASCII_CAST when compiling your code to hide this function so that automatic casts are not done. This has the added advantage that you catch the programming error described under operator!().

bool QString::operator! () const

Returns TRUE if it is a null string; otherwise FALSE.

```
    QString name = getName();
    if ( !name )
        name = "Rodney";
```

Note that if you say

```
    QString name = getName();
    if ( name )
        doSomethingWith(name);
```

It will call "operator const char*()", which is inefficent; you may wish to define the macro QT_NO_ASCII_CAST when writing code which you wish to remain Unicode-clean.

When you want the above semantics, use:

```
    QString name = getName();
    if ( !name.isNull() )
        doSomethingWith(name);
```

QString & QString::operator+= (const QString & str)

Appends *str* to the string and returns a reference to the string.

QString & QString::operator+= (QChar c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends *c* to the string and returns a reference to the string.

QString & QString::operator+= (char c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends *c* to the string and returns a reference to the string.

QString & QString::operator= (QChar c)

Sets the string to contain just the single character *c*.

QString & QString::operator= (const QString & s)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns a shallow copy of *s* to this string and returns a reference to this string. This is very fast because the string isn't actually copied.

QString & QString::operator= (const char * str)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns a deep copy of *str*, interpreted as a classic C string to this string and returns a reference to this string.

If *str* is 0, then a null string is created.

See also isNull() [p. 189].

QString & QString::operator= (const QCString & cs)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns a deep copy of *cs*, interpreted as a classic C string, to this string and returns a reference to this string.

QString & QString::operator= (char c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to contain just the single character *c*.

QChar QString::operator[] (int i) const

Returns the character at index *i*, or QChar::null if *i* is beyond the length of the string.

If the QString is not const (i.e., const QString) or const& (i.e., const QString&), then the non-const overload of operator[] will be used instead.

QCharRef QString::operator[] (int i)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

The function returns a reference to the character at index *i*. The resulting reference can then be assigned to, or used immediately, but it will become invalid once further modifications are made to the original string.

If *i* is beyond the length of the string then the string is expanded with QChar::nulls, so that the QCharRef references a valid (null) character in the string.

The QCharRef internal class can be used much like a constant QChar, but if you assign to it, you change the original string (which will detach itself because of QString's copy-on-write semantics). You will get compilation errors if you try to use the result as anything but a QChar.

QString & QString::prepend (const QString & s)

Inserts *s* at the beginning of the string and returns a reference to the string.

Equivalent to insert(0, *s*).

```
    QString string = "42";
    string.prepend( "The answer is " );
    // string == "The answer is 42"
```

See also insert() [p. 188].

QString & QString::prepend (char ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts *ch* at the beginning of the string and returns a reference to the string.

Equivalent to insert(0, *ch*).

See also insert() [p. 188].

QString & QString::prepend (QChar ch)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts *ch* at the beginning of the string and returns a reference to the string.

Equivalent to insert(0, *ch*).

See also insert() [p. 188].

QChar & QString::ref (uint i)

Returns the QChar at index *i* by reference, expanding the string with QChar::null if necessary. The resulting reference can be assigned to, or otherwise used immediately, but becomes invalid once furher modifications are made to the string.

```
    QString string("ABCDEF");
    QChar ch = string.ref( 3 );         // ch == 'D'
```

See also constref() [p. 184].

QString & QString::remove (uint index, uint len)

Removes *len* characters starting at position *index* from the string and returns a reference to the string.

If *index* is beyond the length of the string, nothing happens. If *index* is within the string, but *index* plus *len* is beyond the end of the string, the string is truncated at position *index*.

```
    QString string( "Montreal" );
    string.remove( 1, 4 );      // string == "Meal"
```

See also insert() [p. 188] and replace() [p. 195].

QString & QString::replace (uint index, uint len, const QString & s)

Replaces *len* characters starting at position *index* from the string with *s*, and returns a reference to the string.

If *index* is beyond the length of the string, nothing is deleted and *s* is appended at the end of the string. If *index* is valid, but *index* plus *len* is beyond the end of the string, the string is truncated at position *index*, then *s* is appended at the end.

```
    QString string( "Say yes!" );
    string = string.replace( 4, 3, "NO" );
    // string == "Say NO!"
```

See also insert() [p. 188] and remove() [p. 195].

Examples: listviews/listviews.cpp, network/networkprotocol/nntp.cpp and qmag/qmag.cpp.

QString & QString::replace (uint index, uint len, const QChar * s, uint slen)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Replaces *len* characters starting at position *index* by *slen* characters of QChar data from *s*, and returns a reference to the string.

See also insert() [p. 188] and remove() [p. 195].

QString & QString::replace (const QRegExp & rx, const QString & str)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Replaces every occurrence of the regexp *rx* in the string with *str*. Returns a reference to the string.

```
QString string = "banana";
string = string.replace( QRegExp("an"), "" ); // string == "ba"
```

See also find() [p. 185] and findRev() [p. 186].

QString QString::right (uint len) const

Returns a string that contains the *len* rightmost characters of the string.

If *len* is greater than the length of the string then the whole string is returned.

```
QString string( "Pineapple" );
QString t = string.right( 5 );   // t == "apple"
```

See also left() [p. 189], mid() [p. 191] and isEmpty() [p. 189].

Example: fileiconview/qfileiconview.cpp.

QString QString::rightJustify (uint width, QChar fill = ' ', bool truncate = FALSE) const

Returns a string of length *width* that contains the *fill* character followed by the string.

If *truncate* is FALSE and the length of the string is more than *width*, then the returned string is a copy of the string.

If *truncate* is TRUE and the length of the string is more than *width*, then the resulting string is truncated at position *width*.

```
QString string( "apple" );
QString t = string.rightJustify( 8, '.' );  // t == "...apple"
```

See also leftJustify() [p. 190].

QString QString::section (QChar sep, int start, int end = 0xffffffff, int flags = SectionDefault) const

This function returns a section of the string.

This string is treated as a sequence of fields separated by the character, *sep*. The returned string consists of the fields from position *start* to position *end* inclusive. If *end* is not specified, all fields from position *start* to the end of the string are included.

The *flags* argument can be used to affect some aspects of the function's behaviour, e.g. whether to be case sensitive, whether to skip empty fields and how to deal with leading and trailing separators; see SectionFlags.

```
QString csv( "forename,middlename,surname,phone" );
QString s = csv.section( ',', 2, 2 );   // s == "surname"

QString path( "/usr/local/bin/myapp" ); // First field is empty
QString s = path.section( '/', 3, 4 );  // s == "bin/myapp"
QString s = path.section( '/', 3, 3, SectionSkipEmpty ); // s == "myapp"
```

If *start* or *end* is negative, we count fields from the right of the string, the right-most field being -1, the one from right-most field being -2, and so on.

```
QString csv( "forename,middlename,surname,phone" );
QString s = csv.section( ',', -3, -2 );  // s == "middlename,surname"

QString path( "/usr/local/bin/myapp" ); // First field is empty
QString s = path.section( '/', -1 ); // s == "myapp"
```

See also QStringList::split() [p. 210].

QString QString::section (char sep, int start, int end = 0xffffffff, int flags = SectionDefault) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QString QString::section (const char * sep, int start, int end = 0xffffffff, int flags = SectionDefault) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

QString QString::section (const QString & sep, int start, int end = 0xffffffff, int flags = SectionDefault) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function returns a section of the string.

This string is treated as a sequence of fields separated by the string, *sep*. The returned string consists of the fields from position *start* to position *end* inclusive. If *end* is not specified, all fields from position *start* to the end of the string are included.

The *flags* argument can be used to affect some aspects of the function's behaviour, e.g. whether to be case sensitive, whether to skip empty fields and how to deal with leading and trailing separators; see SectionFlags.

```
QString data( "forename**middlename**surname**phone" );
QString s = data.section( "**", 2, 2 ); // s == "surname"
```

If *start* or *end* is negative, we count fields from the right of the string, the right-most field being -1, the one from right-most field being -2, and so on.

```
QString data( "forename**middlename**surname**phone" );
QString s = data.section( "**", -3, -2 ); // s == "middlename**surname"
```

See also QStringList::split() [p. 210].

QString QString::section (const QRegExp & reg, int start, int end = 0xffffffff, int flags = SectionDefault) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This function returns a section of the string.

This string is treated as a sequence of fields separated by the regular expression, *reg*. The returned string consists of the fields from position *start* to position *end* inclusive. If *end* is not specified, all fields from position *start* to the end of the string are included.

The *flags* argument can be used to affect some aspects of the function's behaviour, e.g. whether to be case sensitive, whether to skip empty fields and how to deal with leading and trailing separators; see SectionFlags.

```
QString line( "forename\tmiddlename  surname \t \t phone" );
QRegExp sep( "\s+" );
QString s = line.section( sep, 2, 2 ); // s == "surname"
```

If *start* or *end* is negative, we count fields from the right of the string, the right-most field being -1, the one from right-most field being -2, and so on.

```
QString line( "forename\tmiddlename  surname \t \t phone" );
QRegExp sep( "\\s+" );
QString s = line.section( sep, -3, -2 ); // s == "middlename  surname"
```

Warning: Section on QRegExp is much more expensive than the overloaded string and character versions.

See also QStringList::split() [p. 210] and simplifyWhiteSpace() [p. 200].

void QString::setExpand (uint index, QChar c)

This function is obsolete. It is provided to keep old source working. We strongly advise against using it in new code.

Sets the character at position *index* to *c* and expands the string if necessary, filling with spaces.

This method is redundant in Qt 3.x, because operator[] will expand the string as necessary.

QString & QString::setLatin1 (const char * str, int len = -1)

Sets this string to *str*, interpreted as a classic Latin1 C string. If *len* is -1 (the default), then it is set to strlen(str).

If *str* is 0 a null string is created. If *str* is "", an empty string is created.

See also isNull() [p. 189] and isEmpty() [p. 189].

void QString::setLength (uint newLen)

Ensures that at least *newLen* characters are allocated to the string, and sets the length of the string to *newLen*. Any new space allocated contains arbitrary data.

If *newLen* is 0, then the string becomes empty, unless the string is null, in which case it remains null.

This function always detaches the string from other references to the same data.

This function is useful for code that needs to build up a long string and wants to avoid repeated reallocation. In this example, we want to add to the string until some condition is true, and we're fairly sure that size is big enough:

```
QString result;
int resultLength = 0;
result.setLength( newLen ) // allocate some space
while ( ... ) {
    result[resultLength++] = ... // fill (part of) the space with data
}
result.truncate[resultLength]; // and get rid of the undefined junk
```

If *newLen* is an underestimate, the worst that will happen is that the loop will slow down.

See also truncate() [p. 203], isNull() [p. 189], isEmpty() [p. 189] and length() [p. 190].

QString & QString::setNum (long n, int base = 10)

Sets the string to the printed value of *n* in base *base* and returns a reference to the string.

The base is 10 by default and must be between 2 and 36.

```
QString string;
string = string.setNum( 1234 );     // string == "1234"
```

QString & QString::setNum (short n, int base = 10)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n* in base *base* and returns a reference to the string.

The base is 10 by default and must be between 2 and 36.

QString & QString::setNum (ushort n, int base = 10)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n* in base *base* and returns a reference to the string.

The base is 10 by default and must be between 2 and 36.

QString & QString::setNum (int n, int base = 10)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n* in base *base* and returns a reference to the string.

The base is 10 by default and must be between 2 and 36.

QString & QString::setNum (uint n, int base = 10)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n* in base *base* and returns a reference to the string.

The base is 10 by default and must be between 2 and 36.

QString & QString::setNum (ulong n, int base = 10)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n* in base *base* and returns a reference to the string.

The base is 10 by default and must be between 2 and 36.

QString & QString::setNum (float n, char f = ’g’, int prec = 6)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n*, formatted in format *f* with precision *prec*, and returns a reference to the string.

The format *f* can be ’f’, ’F’, ’e’, ’E’, ’g’ or ’G’. See arg() for an explanation of the formats.

QString & QString::setNum (double n, char f = ’g’, int prec = 6)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Sets the string to the printed value of *n*, formatted in format *f* with precision *prec*, and returns a reference to the string.

The format *f* can be ’f’, ’F’, ’e’, ’E’, ’g’ or ’G’. See arg() for an explanation of the formats.

QString & QString::setUnicode (const QChar * unicode, uint len)

Resizes the string to *len* characters and copies *unicode* into the string. If *unicode* is null, nothing is copied, but the string is still resized to *len*. If *len* is zero, then the string becomes a null string.

See also setLatin1() [p. 198] and isNull() [p. 189].

QString & QString::setUnicodeCodes (const ushort * unicode_as_ushorts, uint len)

Resizes the string to *len* characters and copies *unicode_as_ushorts* into the string (on some X11 client platforms this will involve a byte-swapping pass).

If *unicode_as_ushorts* is null, nothing is copied, but the string is still resized to *len*. If *len* is zero, the string becomes a null string.

See also setLatin1() [p. 198] and isNull() [p. 189].

QString QString::simplifyWhiteSpace () const

Returns a string that has whitespace removed from the start and the end of the string, and any sequence of internal whitespace is replaced with a single space.

Whitespace means any character for which QChar::isSpace() returns TRUE. This includes UNICODE characters with decimal values 9 (TAB), 10 (LF), 11 (VT), 12 (FF), 13 (CR), and 32 (Space).

```
    QString string = "  lots\t of\nwhite    space ";
    QString t = string.simplifyWhiteSpace();
    // t == "lots of white space"
```

See also stripWhiteSpace() [p. 201].

QString & QString::sprintf (const char * cformat, ...)

Safely builds a formatted string from the format string *cformat* and an arbitrary list of arguments. The format string supports all the escape sequences of printf() in the standard C library.

The %s escape sequence expects a utf8() encoded string. The format string *cformat* is expected to be in latin1. If you need a unicode format string, use arg() instead. For typesafe string building, with full Unicode support, you can use QTextOStream like this:

```
    QString str;
    QString s = ...;
    int x = ...;
    QTextOStream( &str ) << s << " : " << x;
```

For translations, especially if the strings contains more than one escape sequence, you should consider using the arg() function instead. This allows the order of the replacements to be controlled by the translator, and has Unicode support.

See also arg() [p. 181].

Examples: dclock/dclock.cpp, forever/forever.cpp, layout/layout.cpp, qmag/qmag.cpp, scrollview/scrollview.cpp, tooltip/tooltip.cpp and xform/xform.cpp.

bool QString::startsWith (const QString & s) const

Returns TRUE if the string starts with *s*; otherwise it returns FALSE.

```
    QString string("Bananas");
    bool a = string.startsWith("Ban");      //  a == TRUE
```

See also endsWith() [p. 185].

QString QString::stripWhiteSpace () const

Returns a string that has whitespace removed from the start and the end.

Whitespace means any character for which QChar::isSpace() returns TRUE. This includes UNICODE characters with decimal values 9 (TAB), 10 (LF), 11 (VT), 12 (FF), 13 (CR) and 32 (Space), and may also include other Unicode characters.

```
    QString string = "   white space   ";
    QString s = string.stripWhiteSpace();       // s == "white space"
```

See also simplifyWhiteSpace() [p. 200].

double QString::toDouble (bool * ok = 0) const

Returns the string converted to a `double` value.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

```
    QString string( "1234.56" );
    double a = string.toDouble();   // a == 1234.56
```

See also number() [p. 191].

float QString::toFloat (bool * ok = 0) const

Returns the string converted to a `float` value.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

See also number() [p. 191].

int QString::toInt (bool * ok = 0, int base = 10) const

Returns the string converted to an `int` value to the base *base*, which is 10 by default and must be between 2 and 36.

If **ok* is nonnull, and is TRUE then there have been no errors in the conversion. If **ok* is nonnull, and is FALSE, then the string is not a number at all or it has invalid characters at the end.

```
    QString str( "FF" );
    bool ok;
    int hex = str.toInt( &ok, 16 );     // hex == 255, ok == TRUE
    int dec = str.toInt( &ok, 10 );     // dec == 0, ok == FALSE
```

See also number() [p. 191].

long QString::toLong (bool * ok = 0, int base = 10) const

Returns the string converted to a `long` value to the base *base*, which is 10 by default and must be between 2 and 36.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

See also number() [p. 191].

short QString::toShort (bool * ok = 0, int base = 10) const

Returns the string converted to a `short` value to the base *base*, which is 10 by default and must be between 2 and 36.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

uint QString::toUInt (bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned int` value to the base *base*, which is 10 by default and must be between 2 and 36.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

See also number() [p. 191].

ulong QString::toULong (bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned long` value to the base *base*, which is 10 by default and must be between 2 and 36.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

See also number() [p. 191].

ushort QString::toUShort (bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned short` value to the base *base*, which is 10 by default and must be between 2 and 36.

If a conversion error occurs, **ok* is set to FALSE (unless *ok* is null, the default) and 0 is returned. Otherwise, **ok* is set to true.

void QString::truncate (uint newLen)

If *newLen* is less than the length of the string, then the string is truncated at position *newLen*. Otherwise nothing will happen.

In Qt 1.x, it was possible to "truncate" a string to a longer length. This is no longer possible; use setLength() if you need to extend the length of a string.

```
    QString s = "truncate this string";
    s.truncate( 5 );                        // s == "trunc"
```

See also setLength() [p. 198].

Example: network/mail/smtp.cpp.

const QChar * QString::unicode () const

Returns the Unicode representation of the string. The result remains valid until the string is modified.

QString QString::upper () const

Returns a string that is the string converted to uppercase.

```
    QString string( "TeXt" );
    str = string.upper();     // t == "TEXT"
```

See also lower() [p. 191].

Examples: scribble/scribble.cpp and sql/overview/custom1/main.cpp.

QCString QString::utf8 () const

Returns the string encoded in UTF8 format.

See QTextCodec for more diverse coding/decoding of Unicode strings.

See also QString::fromUtf8() [p. 188], local8Bit() [p. 190] and latin1() [p. 189].

Related Functions

bool operator!= (const QString & s1, const QString & s2)

Returns TRUE if *s1* is not equal to *s2* or FALSE if they are equal. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) != 0.

bool operator!= (const QString & s1, const char * s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is not equal to *s2* or FALSE if they are equal. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) != 0.

bool operator!= (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is not equal to *s2* or FALSE if they are equal. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) != 0.

const QString operator+ (const QString & s1, const QString & s2)

Returns a string which is the result of concatenating the string *s1* and the string *s2*.

Equivalent to *s1*.append(*s2*).

const QString operator+ (const QString & s1, const char * s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which is the result of concatenating the string *s1* and character *s2*.

Equivalent to *s1*.append(*s2*).

const QString operator+ (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which is the result of concatenating the character *s1* and string *s2*.

const QString operator+ (const QString & s, char c)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which is the result of concatenating the string *s* and character *c*.

Equivalent to *s*.append(*c*).

const QString operator+ (char c, const QString & s)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a string which is the result of concatenating the character *c* and string *s*.

Equivalent to *s*.prepend(*c*).

bool operator< (const QString & s1, const char * s2)

Returns TRUE if *s1* is lexically less than *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) < 0.

bool operator< (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is lexically less than *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) < 0.

QDataStream & operator<< (QDataStream & s, const QString & str)

Writes the string *str* to the stream *s*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt]

bool operator<= (const QString & s1, const char * s2)

Returns TRUE if *s1* is lexically less than or equal to *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*,*s2*) <= 0.

bool operator<= (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is lexically less than or equal to *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) <= 0.

bool operator== (const QString & s1, const QString & s2)

Returns TRUE if *s1* is equal to *s2* or FALSE if they are different. Note that a null string is not equal to a nonnull empty string.

Equivalent to compare(*s1*, *s2*) != 0.

bool operator== (const QString & s1, const char * s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is equal to *s2* or FALSE if they are different. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) == 0.

bool operator== (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is equal to *s2* or FALSE if they are different. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) == 0.

bool operator> (const QString & s1, const char * s2)

Returns TRUE if *s1* is lexically greater than *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) > 0.

bool operator> (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is lexically greater than *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) > 0.

bool operator>= (const QString & s1, const char * s2)

Returns TRUE if *s1* is lexically greater than or equal to *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) >= 0.

bool operator>= (const char * s1, const QString & s2)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if *s1* is lexically greater than or equal to *s2* or FALSE if it is not. The comparison is case sensitive. Note that a null string is not equal to an empty string which is nonnull.

Equivalent to compare(*s1*, *s2*) >= 0.

QDataStream & operator>> (QDataStream & s, QString & str)

Reads a string from the stream *s* into string *str*.

See also Format of the QDataStream operators [Input/Output and Networking with Qt]

QStringList Class Reference

The QStringList class provides a list of strings.

`#include <qstringlist.h>`

Inherits QValueList [p. 214]<QString>.

Public Members

- **QStringList** ()
- **QStringList** (const QStringList & l)
- **QStringList** (const QValueList<QString> & l)
- **QStringList** (const QString & i)
- **QStringList** (const char * i)
- void **sort** ()
- QString **join** (const QString & sep) const
- QStringList **grep** (const QString & str, bool cs = TRUE) const
- QStringList **grep** (const QRegExp & expr) const

Static Public Members

- QStringList **fromStrList** (const QStrList & ascii)
- QStringList **split** (const QString & sep, const QString & str, bool allowEmptyEntries = FALSE)
- QStringList **split** (const QChar & sep, const QString & str, bool allowEmptyEntries = FALSE)
- QStringList **split** (const QRegExp & sep, const QString & str, bool allowEmptyEntries = FALSE)

Detailed Description

The QStringList class provides a list of strings.

It is used to store and manipulate strings that logically belong together. Basically QStringList is a QValueList of QString objects. As opposed to QStrList, which stores pointers to characters, QStringList deals with real QString objects. It is the class of choice whenever you work with Unicode strings. QStringList is part of the Qt Template Library.

Like QString itself, QStringList objects are implicitly shared. Passing them around as value-parameters is both fast and safe.

Strings can be added to a list using append(), operator+=() or operator<<(), e.g.

```
    QStringList fonts;
    fonts.append( "Times" );
    fonts += "Courier";
    fonts += "Courier New";
    fonts << "Helvetica [Cronyx]" << "Helvetica [Adobe]";
```

String lists have an iterator, QStringList::Iterator(), e.g.

```
    for ( QStringList::Iterator it = fonts.begin(); it != fonts.end(); ++it ) {
        cout << *it << ":";
    }
    cout << endl;
    // Output:
    //  Times:Courier:Courier New:Helvetica [Cronyx]:Helvetica [Adobe]:
```

You can concatenate all the strings in a string list into a single string (with an optional separator) using join(), e.g.

```
    QString allFonts = fonts.join( ", " );
    cout << allFonts << endl;
    // Output:
    //  Times, Courier, Courier New, Helvetica [Cronyx], Helvetica [Adobe]
```

You can sort the list with sort(), and extract a new list which contains only those strings which contain a particular substring (or match a particular regular expression) using the grep() functions, e.g.

```
    fonts.sort();
    cout << fonts.join( ", " ) << endl;
    // Output:
    //  Courier, Courier New, Helvetica [Adobe], Helvetica [Cronyx], Times

    QStringList helveticas = fonts.grep( "Helvetica" );
    cout << helveticas.join( ", " ) << endl;
    // Output:
    //  Helvetica [Adobe], Helvetica [Cronyx]
```

Existing strings can be split into string lists with character, string or regular expression separators, e.g.

```
    QString s = "Red\tGreen\tBlue";
    QStringList colors = QStringList::split( "\t", s );
    cout << colors.join( ", " ) << endl;
    // Output:
    //  Red, Green, Blue
```

See also Implicitly and Explicitly Shared Classes, Text Related Classes and Non-GUI Classes.

Member Function Documentation

QStringList::QStringList ()

Creates an empty string list.

QStringList::QStringList (const QStringList & l)

Creates a copy of the list *l*. This function is very fast because QStringList is implicitly shared. However, for the programmer this is the same as a deep copy. If this list or the original one or some other list referencing the same shared data is modified, the modifying list first makes a copy, i.e. copy-on-write.

QStringList::QStringList (const QValueList<QString> & l)

Constructs a new string list that is a copy of *l*.

QStringList::QStringList (const QString & i)

Constructs a string list consisting of the single string *i*. Longer lists are easily created as follows:

```
    QStringList items;
    items << "Buy" << "Sell" << "Update" << "Value";
```

QStringList::QStringList (const char * i)

Constructs a string list consisting of the single latin-1 string *i*.

QStringList QStringList::fromStrList (const QStrList & ascii) [static]

Converts from an ASCII-QStrList *ascii* to a QStringList (Unicode).

QStringList QStringList::grep (const QString & str, bool cs = TRUE) const

Returns a list of all strings containing the substring *str*.

If *cs* is TRUE, the grep is done case-sensitively; otherwise case is ignored.

QStringList QStringList::grep (const QRegExp & expr) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a list of all the strings that contain a substring that matches the regular expression *expr*.

QString QStringList::join (const QString & sep) const

Joins the string list into a single string with each element separated by the string *sep*.

See also split() [p. 210].

void QStringList::sort ()

Sorts the list of strings in ascending case-sensitive order.

Sorting is very fast. It uses the Qt Template Library's efficient HeapSort implementation that has a time complexity of O(n*log n).

If you want to sort your strings in an arbitrary order consider using a QMap. For example you could use a QMap<QString,QString> to create a case-insensitive ordering (e.g. mapping the lowercase text to the text), or a QMap<int,QString> to sort the strings by some integer index, etc.

QStringList QStringList::split (const QRegExp & sep, const QString & str, bool allowEmptyEntries = FALSE) [static]

Splits the string *str* into strings wherever the regular expression *sep* occurs, and returns the list of those strings.

If *allowEmptyEntries* is TRUE, an empty string is inserted in the list wherever the separator matches twice without intervening text.

For example, if you split the string "a,,b,c" on commas, split() returns the three-item list "a", "b", "c" if *allowEmptyEntries* is FALSE (the default), and the four-item list "a", "", "b", "c" if *allowEmptyEntries* is TRUE.

If *sep* does not match anywhere in *str*, split() returns a list consisting of the single string *str*.

See also join() [p. 209] and QString::section() [p. 196].

Examples: dirview/dirview.cpp and network/httpd/httpd.cpp.

QStringList QStringList::split (const QString & sep, const QString & str, bool allowEmptyEntries = FALSE) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This version of the function uses a QString as separator, rather than a regular expression.

If *sep* is an empty string, the return value is a list of one-character strings: split(QString(""), "mfc") returns the three-item list, "m", "f", "c".

If *allowEmptyEntries* is TRUE, an empty string is inserted in the list wherever the separator matches twice without intervening text.

See also join() [p. 209] and QString::section() [p. 196].

QStringList QStringList::split (const QChar & sep, const QString & str, bool allowEmptyEntries = FALSE) [static]

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

This version of the function uses a QChar as separator, rather than a regular expression.

See also join() [p. 209] and QString::section() [p. 196].

QStrList Class Reference

The QStrList class provides a doubly-linked list of char*.

`#include <qstrlist.h>`

Inherits QPtrList [p. 144]<char>.

Inherited by QStrIList [p. 174].

Public Members

- **QStrList** (bool deepCopies = TRUE)
- **QStrList** (const QStrList & list)
- **~QStrList** ()
- QStrList & **operator=** (const QStrList & list)

Detailed Description

The QStrList class provides a doubly-linked list of char*.

If you want a string list of QStrings use QStringList.

This class is a QPtrList<char> instance (a list of char*).

QStrList can make deep or shallow copies of the strings that are inserted.

A deep copy means that memory is allocated for the string and then the string data is copied into this memory. A shallow copy is just a copy of the pointer value and not of the string data itself.

The disadvantage of shallow copies is that because a pointer can be deleted only once, the program must put all strings in a central place and know when it is safe to delete them (i.e. when the strings are no longer referenced by other parts of the program). This can make the program more complex. The advantage of shallow copies is that shallow copies consume far less memory than deep copies. It is also much faster to copy a pointer (typically 4 or 8 bytes) than to copy string data.

A QStrList that operates on deep copies will, by default, turn on auto-deletion (see setAutoDelete()). Thus, by default QStrList will deallocate any string copies it allocates.

The virtual compareItems() function is reimplemented and does a case-sensitive string comparison. The inSort() function will insert strings in a sorted order. In general it is fastest to insert the strings as they come and sort() at the end; inSort() is useful when you just have to add a few extra strings to an already sorted list.

The QStrListIterator class is an iterator for QStrList.

See also Collection Classes [p. 9], Text Related Classes and Non-GUI Classes.

Member Function Documentation

QStrList::QStrList (bool deepCopies = TRUE)

Constructs an empty list of strings. Will make deep copies of all inserted strings if *deepCopies* is TRUE, or use shallow copies if *deepCopies* is FALSE.

QStrList::QStrList (const QStrList & list)

Constructs a copy of *list*.

If *list* has deep copies, this list will also get deep copies. Only the pointers are copied (shallow copy) if the other list does not use deep copies.

QStrList::~QStrList ()

Destroys the list. All strings are removed.

QStrList & QStrList::operator= (const QStrList & list)

Assigns *list* to this list and returns a reference to this list.

If *list* has deep copies, this list will also get deep copies. Only the pointers are copied (shallow copy) if the other list does not use deep copies.

QStrListIterator Class Reference

The QStrListIterator class is an iterator for the QStrList and QStrIList classes.

`#include <qstrlist.h>`

Inherits QPtrListIterator [p. 156]<char>.

Detailed Description

The QStrListIterator class is an iterator for the QStrList and QStrIList classes.

This class is a QPtrListIterator<char> instance. It can traverse the strings in the QStrList and QStrIList classes.

See also Non-GUI Classes.

QValueList Class Reference

The QValueList class is a value-based template class that provides doubly linked lists.

`#include <qvaluelist.h>`

Inherited by QCanvasItemList [Graphics with Qt], QStringList [p. 207] and QValueStack [p. 233].

Public Members

- typedef QValueListIterator<T> **iterator**
- typedef QValueListConstIterator<T> **const_iterator**
- typedef T **value_type**
- typedef value_type * **pointer**
- typedef const value_type * **const_pointer**
- typedef value_type & **reference**
- typedef const value_type & **const_reference**
- typedef size_t **size_type**
- **QValueList** ()
- **QValueList** (const QValueList<T> & l)
- **QValueList** (const std::list<T> & l)
- **~QValueList** ()
- QValueList<T> & **operator=** (const QValueList<T> & l)
- QValueList<T> & **operator=** (const std::list<T> & l)
- bool **operator==** (const std::list<T> & l) const
- bool **operator==** (const QValueList<T> & l) const
- bool **operator!=** (const QValueList<T> & l) const
- iterator **begin** ()
- const_iterator **begin** () const
- iterator **end** ()
- const_iterator **end** () const
- iterator **insert** (iterator it, const T & x)
- uint **remove** (const T & x)
- void **clear** ()
- QValueList<T> & **operator<<** (const T & x)
- size_type **size** () const
- bool **empty** () const
- void **push_front** (const T & x)
- void **push_back** (const T & x)
- iterator **erase** (iterator it)
- iterator **erase** (iterator first, iterator last)
- reference **front** ()

- const_reference **front** () const
- reference **back** ()
- const_reference **back** () const
- void **pop_front** ()
- void **pop_back** ()
- void **insert** (iterator pos, size_type n, const T & x)
- QValueList<T> **operator+** (const QValueList<T> & l) const
- QValueList<T> & **operator+=** (const QValueList<T> & l)
- iterator **fromLast** ()
- const_iterator **fromLast** () const
- bool **isEmpty** () const
- iterator **append** (const T & x)
- iterator **prepend** (const T & x)
- iterator **remove** (iterator it)
- T & **first** ()
- const T & **first** () const
- T & **last** ()
- const T & **last** () const
- T & **operator[]** (size_type i)
- const T & **operator[]** (size_type i) const
- iterator **at** (size_type i)
- const_iterator **at** (size_type i) const
- iterator **find** (const T & x)
- const_iterator **find** (const T & x) const
- iterator **find** (iterator it, const T & x)
- const_iterator **find** (const_iterator it, const T & x) const
- int **findIndex** (const T & x) const
- size_type **contains** (const T & x) const
- size_type **count** () const
- QValueList<T> & **operator+=** (const T & x)
- typedef QValueListIterator<T> **Iterator**
- typedef QValueListConstIterator<T> **ConstIterator**

Related Functions

- QDataStream & **operator>>** (QDataStream & s, QValueList<T> & l)
- QDataStream & **operator<<** (QDataStream & s, const QValueList<T> & l)

Detailed Description

The QValueList class is a value-based template class that provides doubly linked lists.

QValueList is a Qt implementation of an STL-like list container. It can be used in your application if the standard `list` is not available. QValueList is part of the Qt Template Library.

QValueList<T> defines a template instance to create a list of values that all have the class T. Note that QValueList does not store pointers to the members of the list; it holds a copy of every member. This is why these kinds of classes are called "value based"; QPtrList and QDict are "pointer based".

QValueList contains and manages a collection of objects of type T and provides iterators that allow the contained objects to be addressed. QValueList owns the contained items. For more relaxed ownership semantics, see QPtrCollection and friends which are pointer-based containers.

Some classes cannot be used within a QValueList, for example, all classes derived from QObject and thus all classes that implement widgets. Only values can be used in a QValueList. To qualify as a value the class must provide:

- A copy constructor
- An assignment operator
- A default constructor, i.e. a constructor that does not take any arguments.

Note that C++ defaults to field-by-field assignment operators and copy constructors if no explicit version is supplied. In many cases this is sufficient.

QValueList's function naming is consistent with the other Qt classes (e.g., count(), isEmpty()). QMap also provides extra functions for compatibility with STL algorithms, such as size() and empty(). Programmers already familiar with the STL `list` can use these functions instead.

Example:

```
class Employee
{
public:
    Employee(): sn(0) {}
    Employee( const QString& forename, const QString& surname, int salary )
        : fn(forename), sn(surname), sal(salary)
    {}

    QString forename() const { return fn; }
    QString surname() const { return sn; }
    int salary() const { return sal; }
    void setSalary( int salary ) { sal = salary; }
private:
    QString fn;
    QString sn;
    int sal;
};

    typedef QValueList EmployeeList;
    EmployeeList list;

    list.append( Employee("John", "Doe", 50000) );
    list.append( Employee("Jane", "Williams", 80000) );
    list.append( Employee("Tom", "Jones", 60000) );

    Employee mary( "Mary", "Hawthorne", 90000 );
    list.append( mary );
    mary.setSalary( 100000 );

    EmployeeList::iterator it;
    for ( it = list.begin(); it != list.end(); ++it )
        cout << (*it).surname().latin1() << ", " <<
                (*it).forename().latin1() << " earns " <<
                (*it).salary() << endl;

    // Output:
    // Doe, John earns 50000
    // Williams, Jane earns 80000
    // Hawthorne, Mary earns 90000
    // Jones, Tom earns 60000
```

Notice that the latest changes to Mary's salary did not affect the value in the list because the list created a copy of Mary's entry.

There are several ways to find items in the list. The begin() and end() functions return iterators to the beginning and end of the list. The advantage of getting an iterator is that you can move forward or backward from this position by incrementing/decrementing the iterator. The iterator returned by end() points to the item which is one past the last item in the container. The past-the-end iterator is still associated with the list it belongs to, however it is *not* dereferenceable; operator*() will not return a well-defined value. If the list is empty(), the iterator returned by begin() will equal the iterator returned by end().

Another way to find an item in the list is by using the qFind() algorithm. For example:

```
    QValueList list;
    ...
    QValueList::iterator it = qFind( list.begin(), list.end(), 3 );
    if ( it != list.end() )
        // it points to the found item
```

It is safe to have multiple iterators on the list at the same time. If some member of the list is removed, only iterators pointing to the removed member become invalid. Inserting into the list does not invalidate any iterator. For convenience, the function last() returns a reference to the last item in the list, and first() returns a reference to the the first item. If the list is empty(), both last() and first() have undefined behavior (your application will crash or do unpredictable things). Use last() and first() with caution, for example:

```
    QValueList list;
    list.append( 1 );
    list.append( 2 );
    list.append( 3 );
    ...
    if ( !list.empty() ) {
        // OK, modify the first item
        int& i = list.first();
        i = 18;
    }
    ...
    QValueList dlist;
    double d = dlist.last(); // undefined
```

Because QValueList is value-based there is no need to be careful about deleting items in the list. The list holds its own copies and will free them if the corresponding member or the list itself is deleted. You can force the list to free all of its items with clear().

QValueList is shared implicitly, which means it can be copied in constant time. If multiple QValueList instances share the same data and one needs to modify its contents, this modifying instance makes a copy and modifies its private copy; therefore it not affect the other instances. This is often called "copy on write". If a QValueList is being used in a multi-threaded program, you must protect all access to the list. See QMutex.

There are several ways to insert items into the list. The prepend() and append() functions insert items at the beginning and the end of the list respectively. The insert() function comes in several flavors and can be used to add one or more items at specific positions within the list.

Items can be also be removed from the list in several ways. There are several variants of the remove() function, which removes a specific item from the list. The remove() function will find and remove items according to a specific item value.

Lists can be also sorted with the sort() function, or can be sorted using the Qt Template Library. For example with qHeapSort():

Example:

```
    QValueList l;
    l.append( 5 );
    l.append( 8 );
    l.append( 3 );
    l.append( 4 );
    qHeapSort( l );
```

See also QValueListIterator [p. 230], Qt Template Library Classes, Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Type Documentation

QValueList::ConstIterator

This iterator is an instantiation of QValueListConstIterator for the same type as this QValueList. In other words, if you instantiate QValueList, ConstIterator is a QValueListConstIterator. Several member function use it, such as QValueList::begin(), which returns an iterator pointing to the first item in the list.

Functionally, this is almost the same as Iterator. The only difference is you cannot use ConstIterator for non-const operations, and that the compiler often can generate better code if you use ConstIterator.

See also QValueListIterator [p. 230] and Iterator [p. 218].

QValueList::Iterator

This iterator is an instantiation of QValueListIterator for the same type as this QValueList. In other words, if you instantiate QValueList, Iterator is a QValueListIterator. Several member function use it, such as QValueList::begin(), which returns an iterator pointing to the first item in the list.

Functionally, this is almost the same as ConstIterator. The only difference is you cannot use ConstIterator for non-const operations, and that the compiler often can generate better code if you use ConstIterator.

See also QValueListIterator [p. 230] and ConstIterator [p. 218].

QValueList::const_iterator

The list's const iterator type, QValueListConstIterator.

QValueList::const_pointer

The const pointer to T type.

QValueList::const_reference

The const reference to T type.

QValueList::iterator

The list's iterator type, QValueListIterator.

QValueList::pointer

The pointer to T type.

QValueList::reference

The reference to T type.

QValueList::size_type

An unsigned integral type, used to represent various sizes.

QValueList::value_type

The type of the object stored in the list, T.

Member Function Documentation

QValueList::QValueList ()

Constructs an empty list.

QValueList::QValueList (const QValueList<T> & l)

Constructs a copy of *l*.

This operation takes O(1) time because QValueList is shared implicitly.

The first modification to a list will take O(n) time.

QValueList::QValueList (const std::list<T> & l)

Contructs a copy of *l*.

This constructor is provided for compatibility with STL containers.

QValueList::~QValueList ()

Destroys the list. References to the values in the list and all iterators of this list become invalidated. Note that it is impossible for an iterator to check whether or not it is valid - QValueList is highly tuned for performance, not for error checking.

iterator QValueList::append (const T & x)

Inserts *x* at the end of the list.

See also insert() [p. 223] and prepend() [p. 225].

Examples: checklists/checklists.cpp and fonts/simple-qfont-demo/viewer.cpp.

const_iterator QValueList::at (size_type i) const

Returns an iterator pointing to the item at position *i* in the list, or end() if the index is out of range.

Warning: This function uses a linear search and can be extremely slow for large lists. QValueList is not optimized for random item access. If you need random access use a different container, such as QValueVector.

iterator QValueList::at (size_type i)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an iterator pointing to the item at position *i* in the list, or end() if the index is out of range.

reference QValueList::back ()

Returns a reference to the last item. If the list contains no last item (i.e. empty() returns TRUE), the return value is undefined.

This function is provided for STL compatibility. It is equivalent to last().

See also front() [p. 223].

const_reference QValueList::back () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

const_iterator QValueList::begin () const

Returns an iterator pointing to the first item in the list. This iterator equals end() if the list is empty.

See also first() [p. 222] and end() [p. 221].

Examples: checklists/checklists.cpp, dirview/dirview.cpp, fonts/simple-qfont-demo/viewer.cpp, network/ftpclient/ftpmainwindow.cpp, network/ftpclient/ftpview.cpp and sql/overview/insert/main.cpp.

iterator QValueList::begin ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an iterator pointing to the first item in the list. This iterator equals end() if the list is empty.

See also first() [p. 222] and end() [p. 221].

void QValueList::clear ()

Removes all items from the list.

See also remove() [p. 225].

size_type QValueList::contains (const T & x) const

Returns the number of occurrences of the value *x* in the list.

size_type QValueList::count () const

Returns the number of items in the list.

See also isEmpty() [p. 223].

bool QValueList::empty () const

Returns TRUE if the list contains no items; otherwise returns FALSE.

See also size() [p. 226].

iterator QValueList::end ()

Returns an iterator pointing behind the last item in the list. This iterator equals begin() if the list is empty.

See also last() [p. 223] and begin() [p. 220].

Examples: checklists/checklists.cpp, dirview/dirview.cpp, fonts/simple-qfont-demo/viewer.cpp, network/ftpclient/ftpmainwindow.cpp, network/ftpclient/ftpview.cpp and sql/overview/insert/main.cpp.

const_iterator QValueList::end () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an iterator pointing behind the last item in the list. This iterator equals begin() if the list is empty.

See also last() [p. 223] and begin() [p. 220].

iterator QValueList::erase (iterator it)

Removes the item pointed to by *it* from the list. No iterators other than *it* or other iterators pointing at the same item as *it* are invalidated. Returns an iterator to the next item after *it*, or end() if there is no such item.

This function is provided for STL compatibility. It is equivalent to remove().

iterator QValueList::erase (iterator first, iterator last)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Deletes all items from *first* to *last* (not including *last*). No iterators are invalidated, except those pointing to the removed items themselves. Returns *last*.

iterator QValueList::find (const T & x)

Returns an iterator pointing to the first occurrence of *x* in the list.

Returns end() is no item matched.

const_iterator QValueList::find (const T & x) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an iterator pointing to the first occurrence of *x* in the list.

Returns end() is no item matched.

iterator QValueList::find (iterator it, const T & x)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of *x* in the list starting at the position given by *it*.

Returns end() if no item matched.

const_iterator QValueList::find (const_iterator it, const T & x) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Finds the first occurrence of *x* in the list starting at the position given by *it*. Returns end() if no item matched.

int QValueList::findIndex (const T & x) const

Returns the index of the first occurrence of the value *x*. Returns -1 if no item matched.

T & QValueList::first ()

Returns a reference to the first item. If the list contains no first item (i.e. isEmpty() returns TRUE), the return value is undefined.

See also last() [p. 223].

Example: network/mail/smtp.cpp.

const T & QValueList::first () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

const_iterator QValueList::fromLast () const

Returns an iterator to the last item in the list, or end() if there is no last item.

Use the end() function instead. For example:

```
QValueList l;
...
QValueList::iterator it = l.end();
--it;
if ( it != end() )
    // ...
```

iterator QValueList::fromLast ()

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns an iterator to the last item in the list, or end() if there is no last item.

Use the end() function instead. For example:

```
QValueList l;
...
QValueList::iterator it = l.end();
--it;
if ( it != end() )
    // ...
```

reference QValueList::front ()

Returns a reference to the first item. If the list contains no first item (i.e. empty() returns TRUE), the return value is undefined.

This function is provided for STL compatibility. It is equivalent to first().

See also back() [p. 220].

const_reference QValueList::front () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

iterator QValueList::insert (iterator it, const T & x)

Inserts the value *x* in front of the iterator *it*.

Returns an iterator pointing at the inserted item.

See also append() [p. 219] and prepend() [p. 225].

void QValueList::insert (iterator pos, size_type n, const T & x)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts *n* copies of *x* before position *pos*.

bool QValueList::isEmpty () const

Returns TRUE if the list contains no items; otherwise returns FALSE.

See also count() [p. 221].

Examples: fonts/simple-qfont-demo/viewer.cpp, network/ftpclient/ftpmainwindow.cpp and network/mail/smtp.cpp.

T & QValueList::last ()

Returns a reference to the last item. If the list contains no last item (i.e. empty() returns TRUE), the return value is undefined.

const T & QValueList::last () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

bool QValueList::operator!= (const QValueList<T> & l) const

Compares both lists.

Returns TRUE if this list and *l* are unequal; otherwise returns FALSE.

QValueList<T> QValueList::operator+ (const QValueList<T> & l) const

Creates a new list and fills it with the items of this list. Then the items of *l* are appended. Returns the new list.

QValueList<T> & QValueList::operator+= (const QValueList<T> & l)

Appends the items of *l* to this list. Returns a reference to this list.

QValueList<T> & QValueList::operator+= (const T & x)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Appends the value *x* to the list. Returns a reference to the list.

QValueList<T> & QValueList::operator<< (const T & x)

Adds the value *x* to the end of the list.

Returns a reference to the list.

QValueList<T> & QValueList::operator= (const QValueList<T> & l)

Assigns *l* to this list and returns a reference to this list.

All iterators of the current list become invalidated by this operation. The cost of such an assignment is O(1) since QValueList is implicitly shared.

QValueList<T> & QValueList::operator= (const std::list<T> & l)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns the contents of *l* to the list.

All iterators of the current list become invalidated by this operation.

bool QValueList::operator== (const QValueList<T> & l) const

Compares both lists.

Returns TRUE if this list and *l* are equal; otherwise returns FALSE.

bool QValueList::operator== (const std::list<T> & l) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns TRUE if this list and *l* are equal; otherwise returns FALSE.

This operator is provided for compatibility with STL containers.

const T & QValueList::operator[] (size_type i) const

Returns a const reference to the item with index *i* in the list. It is up to you to check whether this item really exists. You can do that easily with the count() function. However this operator does not check whether *i* is in range and will deliver undefined results if it does not exist.

Warning: This function uses a linear search and can be extremely slow for large lists. QValueList is not optimized for random item access. If you need random access use a different container, such as QValueVector.

T & QValueList::operator[] (size_type i)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a non-const reference to the item with index *i*.

void QValueList::pop_back ()

Removes the last item. If there is no last item, this operation is undefined.

This function is provided for STL compatibility.

void QValueList::pop_front ()

Removes the first item. If there is no first item, this operation is undefined.

This function is provided for STL compatibility.

iterator QValueList::prepend (const T & x)

Inserts *x* at the beginning of the list.

See also insert() [p. 223] and append() [p. 219].

void QValueList::push_back (const T & x)

Inserts *x* at the end of the list.

This function is provided for STL compatibility. It is equivalent to append().

void QValueList::push_front (const T & x)

Inserts *x* at the beginning of the list.

This function is provided for STL compatibility. It is equivalent to prepend().

iterator QValueList::remove (iterator it)

Removes the item pointed to by *it* from the list. No iterators other than *it* or other iterators pointing at the same item as *it* are invalidated. Returns an iterator to the next item after *it*, or end() if there is no such item.

See also clear() [p. 220].

uint QValueList::remove (const T & x)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes all items that have value *x* and returns the number of removed items.

size_type QValueList::size () const

Returns the number of items in the list.

This function is provided for STL compatibility. It is equivalent to count().

See also empty() [p. 221].

Example: network/ftpclient/ftpview.cpp.

Related Functions

QDataStream & operator<< (QDataStream & s, const QValueList<T> & l)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Writes a list, *l*, to the stream *s*. The type T stored in the list must implement the streaming operator, too.

QDataStream & operator>> (QDataStream & s, QValueList<T> & l)

Reads a list, *l*, from the stream *s*. The type T stored in the list must implement the streaming operator, too.

QValueListConstIterator Class Reference

The QValueListConstIterator class provides a const iterator for QValueList.

`#include <qvaluelist.h>`

Public Members

- typedef T **value_type**
- typedef const T * **pointer**
- typedef const T & **reference**
- **QValueListConstIterator** ()
- **QValueListConstIterator** (const QValueListConstIterator<T> & it)
- **QValueListConstIterator** (const QValueListIterator<T> & it)
- bool **operator==** (const QValueListConstIterator<T> & it) const
- bool **operator!=** (const QValueListConstIterator<T> & it) const
- const T & **operator*** () const
- QValueListConstIterator<T> & **operator++** ()
- QValueListConstIterator<T> **operator++** (int)
- QValueListConstIterator<T> & **operator--** ()
- QValueListConstIterator<T> **operator--** (int)

Detailed Description

The QValueListConstIterator class provides a const iterator for QValueList.

In contrast to QValueListIterator, this class is used to iterate over a const list. It does not allow modification of the values of the list since that would break const semantics.

You can create the appropriate const iterator type by using the `const_iterator` typedef provided by QValueList.

For more information on QValueList iterators, see QValueListIterator.

See also QValueListIterator [p. 230], QValueList [p. 214], Qt Template Library Classes and Non-GUI Classes.

Member Type Documentation

QValueListConstIterator::pointer

Pointer to value_type.

QValueListConstIterator::reference

Reference to value_type.

QValueListConstIterator::value_type

The type of value.

Member Function Documentation

QValueListConstIterator::QValueListConstIterator ()

Creates un uninitialized iterator.

QValueListConstIterator::QValueListConstIterator (const QValueListConstIterator<T> & it)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Constructs a copy of the iterator *it*.

QValueListConstIterator::QValueListConstIterator (const QValueListIterator<T> & it)

Constructs a copy of the iterator *it*.

bool QValueListConstIterator::operator!= (const QValueListConstIterator<T> & it) const

Compares this iterator with *it* and returns TRUE if they point to different items; otherwise returns FALSE.

const T & QValueListConstIterator::operator* () const

Asterisk operator. Returns a reference to the current iterator item.

QValueListConstIterator<T> & QValueListConstIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

QValueListConstIterator<T> QValueListConstIterator::operator++ (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

QValueListConstIterator<T> & QValueListConstIterator::operator-- ()

Prefix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

QValueListConstIterator<T> QValueListConstIterator::operator-- (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

bool QValueListConstIterator::operator== (const QValueListConstIterator<T> & it) const

Compares this iterator with *it* and returns TRUE if they point to the same item; otherwise returns FALSE.

QValueListIterator Class Reference

The QValueListIterator class provides an iterator for QValueList.

`#include <qvaluelist.h>`

Public Members

- typedef T **value_type**
- typedef T * **pointer**
- typedef T & **reference**
- **QValueListIterator** ()
- **QValueListIterator** (const QValueListIterator<T> & it)
- bool **operator==** (const QValueListIterator<T> & it) const
- bool **operator!=** (const QValueListIterator<T> & it) const
- const T & **operator*** () const
- T & **operator*** ()
- QValueListIterator<T> & **operator++** ()
- QValueListIterator<T> **operator++** (int)
- QValueListIterator<T> & **operator--** ()
- QValueListIterator<T> **operator--** (int)

Detailed Description

The QValueListIterator class provides an iterator for QValueList.

An iterator is a class for accessing the items of a container classes - a generalization of the index in an array. A pointer into a "const char *" and an index into an "int[]" are both iterators, and the general idea is to provide that functionality for any data structure.

The QValueListIterator class is an iterator for QValueList instantiations. You can create the appropriate iterator type by using the `iterator` typedef provided by QValueList.

The only way to access the items in a QValueList is to use an iterator.

Example (see QValueList for the complete code):

```
EmployeeList::iterator it;
for ( it = list.begin(); it != list.end(); ++it )
    cout << (*it).surname().latin1() << ", " <<
            (*it).forename().latin1() << " earns " <<
            (*it).salary() << endl;

// Output:
// Doe, John earns 50000
// Williams, Jane earns 80000
// Hawthorne, Mary earns 90000
// Jones, Tom earns 60000
```

QValueList is highly optimized for performance and memory usage. This means that you must be careful: QValueList does not know about all its iterators and the iterators don't know to which list they belong. This makes things very fast, but if you're not careful, you can get spectacular bugs. Always make sure iterators are valid before dereferencing them or using them as parameters to generic algorithms in the STL or the QTL.

Using an invalid iterator is undefined (your application will probably crash).

For every Iterator there is a ConstIterator. When accessing a QValueList in a const environment or if the reference or pointer to the list is itself const, then you must use the ConstIterator. Its semantics are the same as the Iterator, but it returns only const references.

See also QValueList [p. 214], QValueListConstIterator [p. 227], Qt Template Library Classes and Non-GUI Classes.

Member Type Documentation

QValueListIterator::pointer

Pointer to value_type.

QValueListIterator::reference

Reference to value_type.

QValueListIterator::value_type

The type of value, T.

Member Function Documentation

QValueListIterator::QValueListIterator ()

Creates un uninitialized iterator.

QValueListIterator::QValueListIterator (const QValueListIterator<T> & it)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Constructs a copy of the iterator *it*.

bool QValueListIterator::operator!= (const QValueListIterator<T> & it) const

Compares this iterator and *it* and returns TRUE if they point to different items; otherwise returns FALSE.

T & QValueListIterator::operator* ()

Asterisk operator. Returns a reference to the current iterator item.

const T & QValueListIterator::operator* () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Asterisk operator. Returns a reference to the current iterator item.

QValueListIterator<T> & QValueListIterator::operator++ ()

Prefix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

QValueListIterator<T> QValueListIterator::operator++ (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix ++ makes the succeeding item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the end of the list. Incrementing the iterator as returned by end() causes undefined results.

QValueListIterator<T> & QValueListIterator::operator-- ()

Prefix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

QValueListIterator<T> QValueListIterator::operator-- (int)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Postfix — makes the previous item current and returns an iterator pointing to the new current item. The iterator cannot check whether it reached the beginning of the list. Decrementing the iterator as returned by begin() causes undefined results.

bool QValueListIterator::operator== (const QValueListIterator<T> & it) const

Compares this iterator and *it* and returns TRUE if they point to the same item; otherwise returns FALSE.

QValueStack Class Reference

The QValueStack class is a value-based template class that provides a stack.

`#include <qvaluestack.h>`

Inherits QValueList [p. 214]<T>.

Public Members

- **QValueStack** ()
- **~QValueStack** ()
- void **push** (const T & d)
- T **pop** ()
- T & **top** ()
- const T & **top** () const

Detailed Description

The QValueStack class is a value-based template class that provides a stack.

Define a template instance QValueStack<X> to create a stack of values that all have the class X. QValueStack is part of the Qt Template Library.

Note that QValueStack does not store pointers to the members of the stack; it holds a copy of every member. That is why these kinds of classes are called "value based"; QPtrStack, QPtrList, and QDict are "reference based".

A stack is a last in, first ut (LIFO) structure. Items are added to the top of the stack with push() and retrieved from the top with pop(). Furthermore, top() provides access to the topmost item without removing it.

Example:

```
QValueStack stack;
stack.push( 1 );
stack.push( 2 );
stack.push( 3 );
while ( ! stack.isEmpty() )
    cout << "Item: " << stack.pop() << endl;

// Output:
//  Item: 3
//  Item: 2
//  Item: 1
```

QValueStack is a specialized QValueList provided for convenience. All of QValueList's functionality also applies to QPtrStack, for example the facility to iterate over all elements using QValueStack::Iterator. See QValueListIterator for further details.

Some classes cannot be used within a QValueStack, for example everything derived from QObject and thus all classes that implement widgets. Only values can be used in a QValueStack. To qualify as a value, the class must provide

- A copy constructor
- An assignment operator
- A default constructor, i.e. a constructor that does not take any arguments.

Note that C++ defaults to field-by-field assignment operators and copy constructors if no explicit version is supplied. In many cases this is sufficient.

See also Qt Template Library Classes, Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Function Documentation

QValueStack::QValueStack ()

Constructs an empty stack.

QValueStack::~QValueStack ()

Destroys the stack. References to the values in the stack and all iterators of this stack become invalidated. Because QValueStack is highly tuned for performance, you won't see warnings if you use invalid iterators because it is impossible for an iterator to check whether or not it is valid.

T QValueStack::pop ()

Removes the top item from the stack and returns it.

See also top() [p. 234] and push() [p. 234].

void QValueStack::push (const T & d)

Adds element, *d*, to the top of the stack. Last in, first out.

This function is equivalent to append().

See also pop() [p. 234] and top() [p. 234].

T & QValueStack::top ()

Returns a reference to the top item of the stack or the item referenced by end() if no such item exists. Note that you must not change the value the end() iterator points to.

This function is equivalent to last().

See also pop() [p. 234], push() [p. 234] and QValueList::fromLast() [p. 222].

const T & QValueStack::top () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a reference to the top item of the stack or the item referenced by end() if no such item exists.

This function is equivalent to last().

See also pop() [p. 234], push() [p. 234] and QValueList::fromLast() [p. 222].

QValueVector Class Reference

The QValueVector class is a value-based template class that provides a dynamic array.

`#include <qvaluevector.h>`

Public Members

- typedef T **value_type**
- typedef value_type * **pointer**
- typedef const value_type * **const_pointer**
- typedef value_type * **iterator**
- typedef const value_type * **const_iterator**
- typedef value_type & **reference**
- typedef const value_type & **const_reference**
- typedef size_t **size_type**
- typedef ptrdiff_t **difference_type**
- **QValueVector** ()
- **QValueVector** (const QValueVector<T> & v)
- **QValueVector** (size_type n, const T & val = T ( ))
- **QValueVector** (std::vector<T> & v)
- **~QValueVector** ()
- QValueVector<T> & **operator=** (const QValueVector<T> & v)
- QValueVector<T> & **operator=** (const std::vector<T> & v)
- size_type **size** () const
- bool **empty** () const
- size_type **capacity** () const
- iterator **begin** ()
- const_iterator **begin** () const
- iterator **end** ()
- const_iterator **end** () const
- reference **at** (size_type i, bool * ok = 0)
- const_reference **at** (size_type i, bool * ok = 0) const
- reference **operator[]** (size_type i)
- const_reference **operator[]** (size_type i) const
- reference **front** ()
- const_reference **front** () const
- reference **back** ()
- const_reference **back** () const
- void **push_back** (const T & x)
- void **pop_back** ()

- iterator **insert** (iterator pos, const T & x)
- iterator **insert** (iterator pos, size_type n, const T & x)
- void **reserve** (size_type n)
- void **resize** (size_type n, const T & val = T ( ))
- void **clear** ()
- iterator **erase** (iterator pos)
- iterator **erase** (iterator first, iterator last)
- bool **operator==** (const QValueVector<T> & x)

Protected Members

- void **detach** ()

Detailed Description

The QValueVector class is a value-based template class that provides a dynamic array.

QValueVector is a Qt implementation of an STL-like vector container. It can be used in your application if the standard `vector` is not available. QValueVector is part of the Qt Template Library.

QValueVector<T> defines a template instance to create a vector of values that all have the class T. Please note that QValueVector does not store pointers to the members of the vector; it holds a copy of every member. QValueVector is said to be value based; in contrast, QPtrList and QDict are pointer based.

QValueVector contains and manages a collection of objects of type T and provides random access iterators that allow the contained objects to be addressed. QValueVector owns the contained elements. For more relaxed ownership semantics, see QPtrCollection and friends which are pointer-based containers.

QValueVector provides good performance if you append or remove elements from the end of the vector. If you insert or remove elements from anywhere but the end, performance is very bad. The reason for this is that elements will need to be copied into new positions.

Some classes cannot be used within a QValueVector - for example, all classes derived from QObject and thus all classes that implement widgets. Only values can be used in a QValueVector. To qualify as a value the class must provide:

- A copy constructor
- An assignment operator
- A default constructor, i.e., a constructor that does not take any arguments.

Note that C++ defaults to field-by-field assignment operators and copy constructors if no explicit version is supplied. In many cases this is sufficient.

QValueVector uses an STL-like syntax to manipulate and address the objects it contains. See this document for more information.

Example:

```
#include <qvaluevector.h>
#include <qstring.h>
#include

class Employee
{
public:
    Employee(): s(0) {}
    Employee( const QString& name, int salary )
        : n(name), s(salary)
    {}

    QString     name()   const              { return n; }
    int         salary() const              { return s; }
    void        setSalary( int salary )     { s = salary; }
private:
    QString     n;
    int         s;
};

int main()
{
    typedef QValueVector EmployeeVector;
    EmployeeVector vec( 4 );        // vector of 4 Employees

    vec[0] = Employee("Bill", 50000);
    vec[1] = Employee("Steve",80000);
    vec[2] = Employee("Ron",  60000);

    Employee joe( "Joe", 50000 );
    vec.push_back( joe );
    joe.setSalary( 4000 );

    EmployeeVector::iterator it;
    for( it = vec.begin(); it != vec.end(); ++it )
        printf( "%s earns %d\n", (*it).name().latin1(), (*it).salary() );

    return 0;
}
```

Program output:

```
        Bill earns 50000
        Steve earns 80000
        Ron earns 60000
        Joe earns 50000
```

As you can see, the latest changes to Joe's salary did not affect the value in the vector because the vector created a copy of Joe's entry.

There are several ways to find items in the vector. The begin() and end() functions return iterators to the beginning and end of the vector. The advantage of getting an iterator is that you can now move forward or backward from this position by incrementing/decrementing the iterator. The iterator returned by end() points to the element which is one past the last element in the container. The past-the-end iterator is still associated with the vector it belongs to, however it is *not* dereferenceable; operator*() will not return a well-defined value. If the vector is empty(), the iterator returned by begin() will equal the iterator returned by end().

The fastest way to access an element of a vector is by using operator[]. This function provides random access and will return a reference to the element located at the specified index. Thus, you can access every element directly, in constant time, providing you know the location of the element. It is undefined to access an element that does not exist (your application will probably crash). For example:

```
  QValueVector vec1;  // an empty vector
  vec1[10] = 4;  // WARNING: undefined, probably a crash
```

```
QValueVector vec2(25); // initialize with 25 elements
vec2[10] = "Dave";  // OK
```

Whenever inserting, removing or referencing elements in a vector, always make sure you are referring to valid positions. For example:

```
void func( QValueVector& vec )
{
    if ( vec.size() > 10 ) {
        vec[9] = 99; // OK
    }
};
```

The iterators provided by vector are random access iterators, therefore you can use them with many generic algorithms, for example, algorithms provided by the STL or the QTL.

Another way to find an element in the vector is by using the std::find() or qFind() algorithms. For example:

```
QValueVector vec;
...
QValueVector::const_iterator it = qFind( vec.begin(), vec.end(), 3 );
if ( it != vector.end() )
    // 'it' points to the found element
```

It is safe to have multiple iterators on the vector at the same time. Since QValueVector manages memory dynamically, all iterators can become invalid if a memory reallocation occurs. For example, if some member of the vector is removed, iterators that point to the removed element and to all following elements become invalidated. Inserting into the middle of the vector will invalidate all iterators. For convenience, the function back() returns a reference to the last element in the vector, and front() one for the first. If the vector is empty(), both back() and front() have undefined behavior (your application will crash or do unpredictable things). Use back() and front() with caution, for example:

```
QValueVector vec( 3 );
vec.push_back( 1 );
vec.push_back( 2 );
vec.push_back( 3 );
...
if ( !vec.empty() ) {
    // OK: modify the first element
    int& i = vec.front();
    i = 18;
}
...
QValueVector dvec;
double d = dvec.back(); // undefined behavior
```

Because QValueVector manages memory dynamically, it is recommended to contruct a vector with an initial size. Inserting and removing elements happens fastest when:

- Inserting or removing elements happens at the end() of the vector
- The vector does not need to allocate additional memory

By creating a QValueVector with a sufficiently large initial size, there will be less memory allocations. Do not use an initial size that is too big, since it will still take time to construct all the empty entries, and the extra space may be wasted if it is never used.

Because QValueVector is value-based there is no need to be careful about deleting elements in the vector. The vector holds its own copies and will free them if the corresponding member or the vector itself is deleted. You can force the vector to free all of its items with clear().

QValueVector is shared implicitly, which means it can be copied in constant time. If multiple QValueVector instances share the same data and one needs to modify its contents, this modifying instance makes a copy and modifies its private copy; it thus does not affect the other instances. This is often called "copy on write". If a QValueVector is being used in a multi-threaded program, you must protect all access to the vector. See QMutex.

There are several ways to insert elements into the vector. The push_back() function insert elements into the end of the vector. The insert() can be used to add elements at specific positions within the vector (normally, inserting elements at the end() of the vector is fastest).

Items can be also be removed from the vector in several ways. There are several variants of the erase() function which removes a specific element, or range of elements, from the vector.

Vectors can be also sorted with various STL algorithms , or it can be sorted using the Qt Template Library. For example with qBubbleSort():

Example:

```
    QValueVector v( 4 );
    v.push_back( 5 );
    v.push_back( 8 );
    v.push_back( 3 );
    v.push_back( 4 );
    qBubbleSort( v );
```

QValueVector stores its elements in contiguous memory. This means that you can use a QValueVector in any situation that requires an array.

See also Qt Template Library Classes, Implicitly and Explicitly Shared Classes and Non-GUI Classes.

Member Type Documentation

QValueVector::const_iterator

The vector's const iterator type.

QValueVector::const_pointer

The const pointer to T type.

QValueVector::const_reference

The const reference to T type.

QValueVector::difference_type

A signed integral type used to represent the distance between two iterators.

QValueVector::iterator

The vector's iterator type.

QValueVector::pointer

The pointer to T type.

QValueVector::reference

The reference to T type.

QValueVector::size_type

An unsigned integral type, used to represent various sizes.

QValueVector::value_type

The type of the object stored in the vector.

Member Function Documentation

QValueVector::QValueVector ()

Constructs an empty vector without any elements. To create a vector which reserves an initial amount of space for elements, use `QValueVector(size_type n)`.

QValueVector::QValueVector (const QValueVector<T> & v)

Constructs a copy of *v*.

This operation costs O(1) time because QValueVector is shared implicitly.

The first modification to the vector does however take O(n) time.

QValueVector::QValueVector (size_type n, const T & val = T ( ))

Constructs a vector with an initial size of *n* elements. Each element is initialized with the value of *val*.

QValueVector::QValueVector (std::vector<T> & v)

Constructs a copy of *v*.

This operation costs O(1) time because QValueVector is shared implicitly.

The first modification to the vector does however take O(n) time.

QValueVector::~QValueVector ()

Destroys the vector, destroying all elements and freeing the memory. References to the values in the vector and all iterators of this vector become invalidated. Note that it is impossible for an iterator to check whether or not it is valid - QValueVector is tuned for performance, not error checking.

reference QValueVector::at (size_type i, bool * ok = 0)

Returns a reference to the element with index *i*. If *ok* is non-null, and the index *i* is out of range, `*<em>ok</em>` is set to FALSE and the returned reference is undefined. If the index *i* is within the range of the vector, and *ok* is non-null, `*<em>ok</em>` is set to TRUE and the returned reference is well defined.

const_reference QValueVector::at (size_type i, bool * ok = 0) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const reference to the element with index *i*. If *ok* is non-null, and the index *i* is out of range, `*<em>ok</em>` is set to FALSE and the returned reference is undefined. If the index *i* is within the range of the vector, and *ok* is non-null, `*<em>ok</em>` is set to TRUE and the returned reference is well defined.

reference QValueVector::back ()

Returns a reference to the last element in the vector. If there is no last element, this function has undefined behavior.

const_reference QValueVector::back () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const reference to the last element in the vector. If there is no last element, this function has undefined behavior.

iterator QValueVector::begin ()

Returns an iterator pointing to the beginning of the vector. If the vector is empty(), the returned iterator will equal end().

const_iterator QValueVector::begin () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const iterator pointing to the beginning of the vector. If the vector is empty(), the returned iterator will equal end().

size_type QValueVector::capacity () const

Returns the maximum number of elements possible without memory reallocation. If memory reallocation takes place, some or all iterators may become invalidated.

void QValueVector::clear ()

Removes all elements from the vector.

void QValueVector::detach () [protected]

If the vector does not share its data with another QValueVector instance, nothing happens. Otherwise the function creates a new copy of this data and detaches from the shared one. This function is called whenever the vector is modified. The implicit sharing mechanism is implemented this way.

bool QValueVector::empty () const

Returns TRUE if the vector is empty, otherwise FALSE. Equivalent to size()==0, but is faster.

iterator QValueVector::end ()

Returns an iterator pointing behind the last element of the vector.

const_iterator QValueVector::end () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const iterator pointing behind the last element of the vector.

iterator QValueVector::erase (iterator pos)

Removes the element at position *pos* and returns the position of the next element.

iterator QValueVector::erase (iterator first, iterator last)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Removes all elements from *first* up to but not including *last* and returns the position of the next element.

reference QValueVector::front ()

Returns a reference to the first element in the vector. If there is no first element, this function has undefined behavior.

const_reference QValueVector::front () const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const reference to the first element in the vector. If there is no first element, this function has undefined behavior.

iterator QValueVector::insert (iterator pos, const T & x)

Inserts a copy of *x* at the position immediately before *pos*.

iterator QValueVector::insert (iterator pos, size_type n, const T & x)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Inserts *n* copies of *x* immediately before position x.

QValueVector<T> & QValueVector::operator= (const QValueVector<T> & v)

Assigns *v* to this vector and returns a reference to this vector.

All iterators of the current vector become invalidated by this operation. The cost of such an assignment is O(1) since QValueVector is implicitly shared.

QValueVector<T> & QValueVector::operator= (const std::vector<T> & v)

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Assigns *v* to this vector and returns a reference to this vector.

All iterators of the current vector become invalidated by this operation. The cost of such an assignment is O(1) since QValueVector is implicitly shared.

bool QValueVector::operator== (const QValueVector<T> & x)

Returns TRUE if each element in this vector equals each corresponding element in *x*, otherwise FALSE is returned.

reference QValueVector::operator[] (size_type i)

Returns a reference to the element at index *i*. If *i* is out of range, this function has undefined behavior.

const_reference QValueVector::operator[] (size_type i) const

This is an overloaded member function, provided for convenience. It behaves essentially like the above function.

Returns a const reference to the element at index *i*. If *i* is out of range, this function has undefined behavior.

void QValueVector::pop_back ()

Removes the last element from the vector.

void QValueVector::push_back (const T & x)

Appends a copy of *x* to the end of the vector.

void QValueVector::reserve (size_type n)

Increases the vector's capacity. If *n* is less than or equal to capacity(), nothing happens. Otherwise, additional memory is allocated so that capacity() will be increased to a value greater than or equal to *n*. All iterators will then become invalidated. Note that the vector's size() and the values of existing elements remain unchanged.

void QValueVector::resize (size_type n, const T & val = T ( ))

Changes the size of the vector to *n*. If n is greater than the current size(), elements are added to the end and initialized with the value of *val*. If *n* is less than size(), elements are removed from the end. If *n* is equal to size() nothing happens.

size_type QValueVector::size () const

Returns the number of elements in the vector.

QVariant Class Reference

The QVariant class acts like a union for the most common Qt data types.

`#include <qvariant.h>`

Public Members

- enum **Type** { Invalid, Map, List, String, StringList, Font, Pixmap, Brush, Rect, Size, Color, Palette, ColorGroup, IconSet, Point, Image, Int, UInt, Bool, Double, CString, PointArray, Region, Bitmap, Cursor, SizePolicy, Date, Time, DateTime, ByteArray, BitArray, KeySequence }
- **QVariant** ()
- **~QVariant** ()
- **QVariant** (const QVariant & p)
- **QVariant** (QDataStream & s)
- **QVariant** (const QString & val)
- **QVariant** (const QCString & val)
- **QVariant** (const char * val)
- **QVariant** (const QStringList & val)
- **QVariant** (const QFont & val)
- **QVariant** (const QPixmap & val)
- **QVariant** (const QImage & val)
- **QVariant** (const QBrush & val)
- **QVariant** (const QPoint & val)
- **QVariant** (const QRect & val)
- **QVariant** (const QSize & val)
- **QVariant** (const QColor & val)
- **QVariant** (const QPalette & val)
- **QVariant** (const QColorGroup & val)
- **QVariant** (const QIconSet & val)
- **QVariant** (const QPointArray & val)
- **QVariant** (const QRegion & val)
- **QVariant** (const QBitmap & val)
- **QVariant** (const QCursor & val)
- **QVariant** (const QDate & val)
- **QVariant** (const QTime & val)
- **QVariant** (const QDateTime & val)
- **QVariant** (const QByteArray & val)
- **QVariant** (const QBitArray & val)
- **QVariant** (const QKeySequence & val)
- **QVariant** (const QValueList<QVariant> & val)

- **QVariant** (const QMap<QString, QVariant> & val)
- **QVariant** (int val)
- **QVariant** (uint val)
- **QVariant** (bool val, int)
- **QVariant** (double val)
- **QVariant** (QSizePolicy val)
- QVariant & **operator=** (const QVariant & variant)
- bool **operator==** (const QVariant & v) const
- bool **operator!=** (const QVariant & v) const
- Type **type** () const
- const char * **typeName** () const
- bool **canCast** (Type t) const
- bool **cast** (Type t)
- bool **isValid** () const
- void **clear** ()
- const QString **toString** () const
- const QCString **toCString** () const
- const QStringList **toStringList** () const
- const QFont **toFont** () const
- const QPixmap **toPixmap** () const
- const QImage **toImage** () const
- const QBrush **toBrush** () const
- const QPoint **toPoint** () const
- const QRect **toRect** () const
- const QSize **toSize** () const
- const QColor **toColor** () const
- const QPalette **toPalette** () const
- const QColorGroup **toColorGroup** () const
- const QIconSet **toIconSet** () const
- const QPointArray **toPointArray** () const
- const QBitmap **toBitmap** () const
- const QRegion **toRegion** () const
- const QCursor **toCursor** () const
- const QDate **toDate** () const
- const QTime **toTime** () const
- const QDateTime **toDateTime** () const
- const QByteArray **toByteArray** () const
- const QBitArray **toBitArray** () const
- const QKeySequence **toKeySequence** () const
- int **toInt** (bool * ok = 0) const
- uint **toUInt** (bool * ok = 0) const
- bool **toBool** () const
- double **toDouble** (bool * ok = 0) const
- const QValueList<QVariant> **toList** () const
- const QMap<QString, QVariant> **toMap** () const
- QSizePolicy **toSizePolicy** () const
- QValueListConstIterator<QString> **stringListBegin** () const
- QValueListConstIterator<QString> **stringListEnd** () const
- QValueListConstIterator<QVariant> **listBegin** () const
- QValueListConstIterator<QVariant> **listEnd** () const

- QMapConstIterator<QString, QVariant> **mapBegin** () const
- QMapConstIterator<QString, QVariant> **mapEnd** () const
- QMapConstIterator<QString, QVariant> **mapFind** (const QString & key) const
- QString & **asString** ()
- QCString & **asCString** ()
- QStringList & **asStringList** ()
- QFont & **asFont** ()
- QPixmap & **asPixmap** ()
- QImage & **asImage** ()
- QBrush & **asBrush** ()
- QPoint & **asPoint** ()
- QRect & **asRect** ()
- QSize & **asSize** ()
- QColor & **asColor** ()
- QPalette & **asPalette** ()
- QColorGroup & **asColorGroup** ()
- QIconSet & **asIconSet** ()
- QPointArray & **asPointArray** ()
- QBitmap & **asBitmap** ()
- QRegion & **asRegion** ()
- QCursor & **asCursor** ()
- QDate & **asDate** ()
- QTime & **asTime** ()
- QDateTime & **asDateTime** ()
- QByteArray & **asByteArray** ()
- QBitArray & **asBitArray** ()
- QKeySequence & **asKeySequence** ()
- int & **asInt** ()
- uint & **asUInt** ()
- bool & **asBool** ()
- double & **asDouble** ()
- QValueList<QVariant> & **asList** ()
- QMap<QString, QVariant> & **asMap** ()
- QSizePolicy & **asSizePolicy** ()

Static Public Members

- const char * **typeToName** (Type typ)
- Type **nameToType** (const char * name)

Detailed Description

The QVariant class acts like a union for the most common Qt data types.

Because C++ forbids unions from including types that have non-default constructors or destructors, most interesting Qt classes cannot be used in unions. This is a problem when using QObject::property(), among other things.

This class provides union functionality for property() and most other needs that might be solved by a union including e.g. QWidget.

A QVariant object can hold any one type() at a time. For example, you can find out what type, T, it holds, convert it to a different type using one of the asT() functions, e.g. asSize(), get its value using one of the toT() functions, e.g. toSize(), and check whether the type can be converted to a particular type using canCast().

The methods named toT() (for any supported T, see the Type documentation for a list) are const. If you ask for the stored type, they return a copy of the stored object. If you ask for a type that can be generated from the stored type, toT() copies and converts and leaves the object itself unchanged. If you ask for a type that cannot be generated from the stored type, the result depends on the type (see the function documentation for details).

Note that three data types supported by QVariant are explicitly shared, namely QImage, QPointArray, and QCString, and in these cases the toT() methods return a shallow copy. In almost all cases you must make a deep copy of the returned values before modifying them.

The asT() functions are not const. They do conversion like the toT() methods, set the variant to hold the converted value, and return a reference to the new contents of the variant.

Here is example code to demonstrate the use of QVariant:

```
    QDataStream out(...);
    QVariant v(123);          // The variant now contains an int
    int x = v.toInt();        // x = 123
    out << v;                 // Writes a type tag and an int to out
    v = QVariant("hello");    // The variant now contains a QCString
    v = QVariant(tr("hello"));// The variant now contains a QString
    int y = v.toInt();        // y = 0 since v cannot be converted to an int
    QString s = v.toString(); // s = tr("hello")  (see QObject::tr())
    out << v;                 // Writes a type tag and a QString to out
    ...
    QDataStream in(...);      // (opening the previously written stream)
    in >> v;                  // Reads an Int variant
    int z = v.toInt();        // z = 123
    qDebug("Type is %s",      // prints "Type is int"
           v.typeName());
    v.asInt() += 100;         // The variant now hold the value 223.
    v = QVariant( QStringList() );
    v.asStringList().append( "Hello" );
```

You can even have store QValueLists and QMaps in a variant, so you can easily construct arbitrarily complex data structures of arbitrary types. This is very powerful and versatile, but may prove less memory and speed efficient than storing specific types in standard data structures. (See the Collection Classes.)

See also Miscellaneous Classes and Object Model.

Member Type Documentation

QVariant::Type

This enum type defines the types of variable that a QVariant can contain. The supported enum values and the associated types are

- `QVariant::Invalid` - no type
- `QVariant::List` - a QValueList
- `QVariant::Map` - a QMap
- `QVariant::String` - a QString
- `QVariant::StringList` - a QStringList

- `QVariant::Font` - a QFont
- `QVariant::Pixmap` - a QPixmap
- `QVariant::Brush` - a QBrush
- `QVariant::Rect` - a QRect
- `QVariant::Size` - a QSize
- `QVariant::Color` - a QColor
- `QVariant::Palette` - a QPalette
- `QVariant::ColorGroup` - a QColorGroup
- `QVariant::IconSet` - a QIconSet
- `QVariant::Point` - a QPoint
- `QVariant::Image` - a QImage
- `QVariant::Int` - an int
- `QVariant::UInt` - an unsigned int
- `QVariant::Bool` - a bool
- `QVariant::Double` - a double
- `QVariant::CString` - a QCString
- `QVariant::PointArray` - a QPointArray
- `QVariant::Region` - a QRegion
- `QVariant::Bitmap` - a QBitmap
- `QVariant::Cursor` - a QCursor
- `QVariant::Date` - a QDate
- `QVariant::Time` - a QTime
- `QVariant::DateTime` - a QDateTime
- `QVariant::ByteArray` - a QByteArray
- `QVariant::BitArray` - a QBitArray
- `QVariant::SizePolicy` - a QSizePolicy
- `QVariant::KeySequence` - a QKeySequence

Note that Qt's definition of bool depends on the compiler. qglobal.h has the system-dependent definition of bool.

Member Function Documentation

QVariant::QVariant ()

Constructs an invalid variant.

QVariant::QVariant (const QVariant & p)

Constructs a copy of the variant, *p*, passed as the argument to this constructor. Usually this is a deep copy, but a shallow copy is made if the stored data type is explicitly shared, as e.g. QImage is.

QVariant::QVariant (QDataStream & s)

Reads the variant from the data stream, *s*.

QVariant::QVariant (const QString & val)

Constructs a new variant with a string value, *val*.

QVariant::QVariant (const QCString & val)

Constructs a new variant with a C-string value, *val*.

If you want to modify the QCString after you've passed it to this constructor, we recommend passing a deep copy (see QCString::copy()).

QVariant::QVariant (const char * val)

Constructs a new variant with a C-string value of *val* if *val* is non-null. The variant creates a deep copy of *val*.

If *val* is null, the resulting variant has type Invalid.

QVariant::QVariant (const QStringList & val)

Constructs a new variant with a string list value, *val*.

QVariant::QVariant (const QFont & val)

Constructs a new variant with a font value, *val*.

QVariant::QVariant (const QPixmap & val)

Constructs a new variant with a pixmap value, *val*.

QVariant::QVariant (const QImage & val)

Constructs a new variant with an image value, *val*.

Because QImage is explicitly shared, you may need to pass a deep copy to the variant using QImage::copy(), e.g. if you intend changing the image you've passed later on.

QVariant::QVariant (const QBrush & val)

Constructs a new variant with a brush value, *val*.

QVariant::QVariant (const QPoint & val)

Constructs a new variant with a point value, *val*.

QVariant::QVariant (const QRect & val)

Constructs a new variant with a rect value, *val*.

QVariant::QVariant (const QSize & val)

Constructs a new variant with a size value, *val*.

QVariant::QVariant (const QColor & val)

Constructs a new variant with a color value, *val*.

QVariant::QVariant (const QPalette & val)

Constructs a new variant with a color palette value, *val*.

QVariant::QVariant (const QColorGroup & val)

Constructs a new variant with a color group value, *val*.

QVariant::QVariant (const QIconSet & val)

Constructs a new variant with an icon set value, *val*.

QVariant::QVariant (const QPointArray & val)

Constructs a new variant with a point array value, *val*.

Because QPointArray is explicitly shared, you may need to pass a deep copy to the variant using QPointArray::copy(), e.g. if you intend changing the point array you've passed later on.

QVariant::QVariant (const QRegion & val)

Constructs a new variant with a region value, *val*.

QVariant::QVariant (const QBitmap & val)

Constructs a new variant with a bitmap value, *val*.

QVariant::QVariant (const QCursor & val)

Constructs a new variant with a cursor value, *val*.

QVariant::QVariant (const QDate & val)

Constructs a new variant with a date value, *val*.

QVariant::QVariant (const QTime & val)

Constructs a new variant with a time value, *val*.

QVariant::QVariant (const QDateTime & val)

Constructs a new variant with a date/time value, *val*.

QVariant::QVariant (const QByteArray & val)

Constructs a new variant with a bytearray value, *val*.

QVariant::QVariant (const QBitArray & val)

Constructs a new variant with a bitarray value, *val*.

QVariant::QVariant (const QKeySequence & val)

Constructs a new variant with a key sequence value, *val*.

QVariant::QVariant (const QValueList<QVariant> & val)

Constructs a new variant with a list value, *val*.

QVariant::QVariant (const QMap<QString, QVariant> & val)

Constructs a new variant with a map of QVariants, *val*.

QVariant::QVariant (int val)

Constructs a new variant with an integer value, *val*.

QVariant::QVariant (uint val)

Constructs a new variant with an unsigned integer value, *val*.

QVariant::QVariant (bool val, int)

Constructs a new variant with a boolean value, *val*. The integer argument is a dummy, necessary for compatibility with some compilers.

QVariant::QVariant (double val)

Constructs a new variant with a floating point value, *val*.

QVariant::QVariant (QSizePolicy val)

Constructs a new variant with a size policy value, *val*.

QVariant::~QVariant ()

Destroys the QVariant and the contained object.

Note that subclasses that reimplement clear() should reimplement the destructor to call clear(). This destructor calls clear(), but because it is the destructor, QVariant::clear() is called rather than a subclass's clear().

QBitArray & QVariant::asBitArray ()

Tries to convert the variant to hold a QBitArray value. If that is not possible then the variant is set to an empty bitarray.

Returns a reference to the stored bitarray.

See also toBitArray() [p. 260].

QBitmap & QVariant::asBitmap ()

Tries to convert the variant to hold a bitmap value. If that is not possible the variant is set to a null bitmap.

Returns a reference to the stored bitmap.

See also toBitmap() [p. 260].

bool & QVariant::asBool ()

Returns the variant's value as bool reference.

QBrush & QVariant::asBrush ()

Tries to convert the variant to hold a brush value. If that is not possible the variant is set to a default black brush.

Returns a reference to the stored brush.

See also toBrush() [p. 260].

QByteArray & QVariant::asByteArray ()

Tries to convert the variant to hold a QByteArray value. If that is not possible then the variant is set to an empty bytearray.

Returns a reference to the stored bytearray.

See also toByteArray() [p. 260].

QCString & QVariant::asCString ()

Tries to convert the variant to hold a string value. If that is not possible the variant is set to an empty string.

Returns a reference to the stored string.

See also toCString() [p. 260].

QColor & QVariant::asColor ()

Tries to convert the variant to hold a QColor value. If that is not possible the variant is set to an invalid color.

Returns a reference to the stored color.

See also toColor() [p. 260] and QColor::isValid() [Graphics with Qt].

QColorGroup & QVariant::asColorGroup ()

Tries to convert the variant to hold a QColorGroup value. If that is not possible the variant is set to a color group with all colors set to black.

Returns a reference to the stored color group.

See also toColorGroup() [p. 261].

QCursor & QVariant::asCursor ()

Tries to convert the variant to hold a QCursor value. If that is not possible the variant is set to a default arrow cursor.

Returns a reference to the stored cursor.

See also toCursor() [p. 261].

QDate & QVariant::asDate ()

Tries to convert the variant to hold a QDate value. If that is not possible then the variant is set to an invalid date.

Returns a reference to the stored date.

See also toDate() [p. 261].

QDateTime & QVariant::asDateTime ()

Tries to convert the variant to hold a QDateTime value. If that is not possible then the variant is set to an invalid date/time.

Returns a reference to the stored date/time.

See also toDateTime() [p. 261].

double & QVariant::asDouble ()

Returns the variant's value as double reference.

QFont & QVariant::asFont ()

Tries to convert the variant to hold a QFont. If that is not possible the variant is set to a default font.

Returns a reference to the stored font.

See also toFont() [p. 261].

QIconSet & QVariant::asIconSet ()

Tries to convert the variant to hold a QIconSet value. If that is not possible the variant is set to an empty iconset.

Returns a reference to the stored iconset.

See also toIconSet() [p. 261].

QImage & QVariant::asImage ()

Tries to convert the variant to hold an image value. If that is not possible the variant is set to a null image.

Returns a reference to the stored image.

See also toImage() [p. 261].

int & QVariant::asInt ()

Returns the variant's value as int reference.

QKeySequence & QVariant::asKeySequence ()

Tries to convert the variant to hold a QKeySequence value. If that is not possible then the variant is set to an empty key sequence.

Returns a reference to the stored key sequence.

See also toKeySequence() [p. 262].

QValueList<QVariant> & QVariant::asList ()

Returns the variant's value as variant list reference.

QMap<QString, QVariant> & QVariant::asMap ()

Returns the variant's value as variant map reference.

QPalette & QVariant::asPalette ()

Tries to convert the variant to hold a QPalette value. If that is not possible the variant is set to a palette with black colors only.

Returns a reference to the stored palette.

See also toString() [p. 263].

QPixmap & QVariant::asPixmap ()

Tries to convert the variant to hold a pixmap value. If that is not possible the variant is set to a null pixmap.

Returns a reference to the stored pixmap.

See also toPixmap() [p. 262].

QPoint & QVariant::asPoint ()

Tries to convert the variant to hold a point value. If that is not possible the variant is set to a null point.

Returns a reference to the stored point.

See also toPoint() [p. 262].

QPointArray & QVariant::asPointArray ()

Tries to convert the variant to hold a QPointArray value. If that is not possible the variant is set to an empty point array.

Returns a reference to the stored point array.

See also toPointArray() [p. 262].

QRect & QVariant::asRect ()

Tries to convert the variant to hold a rectangle value. If that is not possible the variant is set to an empty rectangle.

Returns a reference to the stored rectangle.

See also toRect() [p. 263].

QRegion & QVariant::asRegion ()

Tries to convert the variant to hold a QRegion value. If that is not possible the variant is set to a null region.

Returns a reference to the stored region.

See also toRegion() [p. 263].

QSize & QVariant::asSize ()

Tries to convert the variant to hold a QSize value. If that is not possible the variant is set to an invalid size.

Returns a reference to the stored size.

See also toSize() [p. 263] and QSize::isValid() [Graphics with Qt].

QSizePolicy & QVariant::asSizePolicy ()

Tries to convert the variant to hold a QSizePolicy value. If that fails, the variant is set to an arbitrary size policy.

QString & QVariant::asString ()

Tries to convert the variant to hold a string value. If that is not possible the variant is set to an empty string.

Returns a reference to the stored string.

See also toString() [p. 263].

QStringList & QVariant::asStringList ()

Tries to convert the variant to hold a QStringList value. If that is not possible the variant is set to an empty string list.

Returns a reference to the stored string list.

See also toStringList() [p. 263].

QTime & QVariant::asTime ()

Tries to convert the variant to hold a QTime value. If that is not possible then the variant is set to an invalid time.

Returns a reference to the stored time.

See also toTime() [p. 263].

uint & QVariant::asUInt ()

Returns the variant's value as unsigned int reference.

bool QVariant::canCast (Type t) const

Returns TRUE if the variant's type can be cast to the requested type, *t*. Such casting is done automatically when calling the toInt(), toBool(), ... or asInt(), asBool(), ... methods.

The following casts are done automatically:

- Bool => Double, Int, UInt
- CString => String
- Date => String
- DateTime => String, Date, Time
- Double => String, Int, Bool, UInt
- Int => String, Double, Bool, UInt
- List => StringList (if the list contains strings or something that can be cast to a string)
- String => CString, Int, Uint, Double, Date, Time, DateTime
- StringList => List
- Time => String
- UInt => String, Double, Bool, Int

bool QVariant::cast (Type t)

Casts the variant to the requested type. If the cast cannot be done, the variant is set to the default value of the requested type (e.g. an empty string if the requested type *t* is QVariant::String, an empty point array if the requested type *t* is QVariant::PointArray, etc). Returns TRUE if the current type of the variant was successfully casted; otherwise returns FALSE.

See also canCast() [p. 258].

void QVariant::clear ()

Convert this variant to type Invalid and free up any resources used.

bool QVariant::isValid () const

Returns TRUE if the storage type of this variant is not QVariant::Invalid; otherwise returns FALSE.

QValueListConstIterator<QVariant> QVariant::listBegin () const

Returns an iterator to the first item in the list if the variant's type is appropriate, or else a null iterator.

QValueListConstIterator<QVariant> QVariant::listEnd () const

Returns the end iterator for the list if the variant's type is appropriate, or else a null iterator.

QMapConstIterator<QString, QVariant> QVariant::mapBegin () const

Returns an iterator to the first item in the map, if the variant's type is appropriate, or else a null iterator.

QMapConstIterator<QString, QVariant> QVariant::mapEnd () const

Returns the end iterator for the map, if the variant's type is appropriate, or else a null iterator.

QMapConstIterator<QString, QVariant> QVariant::mapFind (const QString & key) const

Returns an iterator to the item in the map with *key* as key, if the variant's type is appropriate and *key* is a valid key, or else a null iterator.

Type QVariant::nameToType (const char * name) [static]

Converts the string representation of the storage type gven in *name*, to its enum representation.

If the string representation cannot be converted to any enum representation, the variant is set to Invalid.

bool QVariant::operator!= (const QVariant & v) const

Compares this QVariant with *v* and returns TRUE if they are not equal; otherwise returns FALSE.

QVariant & QVariant::operator= (const QVariant & variant)

Assigns the value of the variant *variant* to this variant.

This is a deep copy of the variant, but note that if the variant holds an explicitly shared type such as QImage, a shallow copy is performed.

bool QVariant::operator== (const QVariant & v) const

Compares this QVariant with *v* and returns TRUE if they are equal; otherwise returns FALSE.

QValueListConstIterator<QString> QVariant::stringListBegin () const

Returns an iterator to the first string in the list if the variant's type is StringList, or else a null iterator.

QValueListConstIterator<QString> QVariant::stringListEnd () const

Returns the end iterator for the list if the variant's type is StringList, or else a null iterator.

const QBitArray QVariant::toBitArray () const

Returns the variant as a QBitArray if the variant has type() BitArray, or an empty bitarray otherwise.

See also asBitArray() [p. 254].

const QBitmap QVariant::toBitmap () const

Returns the variant as a QBitmap if the variant has type() Bitmap, or a null QBitmap otherwise.

See also asBitmap() [p. 254].

bool QVariant::toBool () const

Returns the variant as a bool if the variant has type() Bool.

Returns TRUE if the variant has type Int, UInt or Double and its value is non-zero; otherwise returns FALSE.

See also asBool() [p. 254].

const QBrush QVariant::toBrush () const

Returns the variant as a QBrush if the variant has type() Brush, or a default brush (with all black colors) otherwise.

See also asBrush() [p. 254].

const QByteArray QVariant::toByteArray () const

Returns the variant as a QByteArray if the variant has type() ByteArray, or an empty bytearray otherwise.

See also asByteArray() [p. 254].

const QCString QVariant::toCString () const

Returns the variant as a QCString if the variant has type() CString or String, or a 0 otherwise.

See also asCString() [p. 254].

const QColor QVariant::toColor () const

Returns the variant as a QColor if the variant has type() Color, or an invalid color otherwise.

See also asColor() [p. 255].

const QColorGroup QVariant::toColorGroup () const

Returns the variant as a QColorGroup if the variant has type() ColorGroup, or a completely black color group otherwise.

See also asColorGroup() [p. 255].

const QCursor QVariant::toCursor () const

Returns the variant as a QCursor if the variant has type() Cursor, or the default arrow cursor otherwise.

See also asCursor() [p. 255].

const QDate QVariant::toDate () const

Returns the variant as a QDate if the variant has type() Date, DateTime or String, or an invalid date otherwise.

Note that if the type() is String an invalid date will be returned if the string cannot be parsed as an Qt::ISODate format date.

See also asDate() [p. 255].

const QDateTime QVariant::toDateTime () const

Returns the variant as a QDateTime if the variant has type() DateTime or String, or an invalid date/time otherwise.

Note that if the type() is String an invalid date/time will be returned if the string cannot be parsed as an Qt::ISODate format date/time.

See also asDateTime() [p. 255].

double QVariant::toDouble (bool * ok = 0) const

Returns the variant as a double if the variant has type() String, CString, Double, Int, UInt, or Bool; or 0.0 otherwise.

If *ok* is non-null, **ok* is set to TRUE if the value could be converted to a double and FALSE otherwise.

See also asDouble() [p. 255].

const QFont QVariant::toFont () const

Returns the variant as a QFont if the variant has type() Font, or the default font otherwise.

See also asFont() [p. 255].

const QIconSet QVariant::toIconSet () const

Returns the variant as a QIconSet if the variant has type() IconSet, or an icon set of null pixmaps otherwise.

See also asIconSet() [p. 256].

const QImage QVariant::toImage () const

Returns the variant as a QImage if the variant has type() Image, or a null image otherwise.

See also asImage() [p. 256].

int QVariant::toInt (bool * ok = 0) const

Returns the variant as an int if the variant has type() String, CString, Int, UInt, Double, Bool or KeySequence; or 0 otherwise.

If *ok* is non-null, **ok* is set to TRUE if the value could be converted to an int and FALSE otherwise.

See also asInt() [p. 256] and canCast() [p. 258].

const QKeySequence QVariant::toKeySequence () const

Returns the variant as a QKeySequence if the variant has type() KeySequence, Int or String, or an empty key sequence otherwise.

Note that not all Ints and Strings are valid key sequences and in such cases an empty key sequence will be returned.

See also asKeySequence() [p. 256].

const QValueList<QVariant> QVariant::toList () const

Returns the variant as a QValueList if the variant has type() List or StringList, or an empty list otherwise.

See also asList() [p. 256].

const QMap<QString, QVariant> QVariant::toMap () const

Returns the variant as a QMap if the variant has type() Map, or an empty map otherwise.

See also asMap() [p. 256].

const QPalette QVariant::toPalette () const

Returns the variant as a QPalette if the variant has type() Palette, or a completely black palette otherwise.

See also asPalette() [p. 256].

const QPixmap QVariant::toPixmap () const

Returns the variant as a QPixmap if the variant has type() Pixmap, or a null pixmap otherwise.

See also asPixmap() [p. 256].

const QPoint QVariant::toPoint () const

Returns the variant as a QPoint if the variant has type() Point, or a point (0, 0) otherwise.

See also asPoint() [p. 257].

const QPointArray QVariant::toPointArray () const

Returns the variant as a QPointArray if the variant has type() PointArray, or an empty QPointArray otherwise.

See also asPointArray() [p. 257].

const QRect QVariant::toRect () const

Returns the variant as a QRect if the variant has type() Rect, or an empty rectangle otherwise.

See also asRect() [p. 257].

const QRegion QVariant::toRegion () const

Returns the variant as a QRegion if the variant has type() Region, or an empty QRegion otherwise.

See also asRegion() [p. 257].

const QSize QVariant::toSize () const

Returns the variant as a QSize if the variant has type() Size, or an invalid size otherwise.

See also asSize() [p. 257].

QSizePolicy QVariant::toSizePolicy () const

Returns the variant as a QSizePolicy if the variant has type() SizePolicy, or an undefined (but legal) size policy otherwise.

const QString QVariant::toString () const

Returns the variant as a QString if the variant has type() String, CString, ByteArray, Int, Uint, Bool, Double, Date, Time, or DateTime, or QString::null otherwise.

See also asString() [p. 257].

const QStringList QVariant::toStringList () const

Returns the variant as a QStringList if the variant has type() StringList or List of a type that can be converted to QString, or an empty list otherwise.

See also asStringList() [p. 258].

const QTime QVariant::toTime () const

Returns the variant as a QTime if the variant has type() Time, DateTime or String, or an invalid time otherwise.

Note that if the type() is String an invalid time will be returned if the string cannot be parsed as an Qt::ISODate format time.

See also asTime() [p. 258].

uint QVariant::toUInt (bool * ok = 0) const

Returns the variant as an unsigned int if the variant has type() String, CString, UInt, Int, Double, or Bool; or 0 otherwise.

If *ok* is non-null, **ok* is set to TRUE if the value could be converted to a uint and FALSE otherwise.

See also asUInt() [p. 258].

Type QVariant::type () const

Returns the storage type of the value stored in the variant. Usually it's best to test with canCast() whether the variant can deliver the data type you are interested in.

const char * QVariant::typeName () const

Returns the name of the type stored in the variant. The returned strings describe the C++ datatype used to store the data: for example, "QFont", "QString", or "QValueList". An Invalid variant returns 0.

const char * QVariant::typeToName (Type typ) [static]

Converts the enum representation of the storage type, *typ*, to its string representation.

Index