QString Class

The QString class provides a Unicode character string. More...

Header:	#include <QString>
Inherited By:	QConstString, QDBusObjectPath, and QDBusSignature

Note: All functions in this class are reentrant, except for ascii(), latin1(), utf8(), and local8Bit(), which are nonreentrant.

- List of all members, including inherited members
- Compatibility members

Public Types

class	**Null**
typedef	**ConstIterator**
typedef	**Iterator**
enum	**NormalizationForm** { NormalizationForm_D, NormalizationForm_C, NormalizationForm_KD, NormalizationForm_KC }
enum	**SectionFlag** { SectionDefault, SectionSkipEmpty, SectionIncludeLeadingSep, SectionIncludeTrailingSep, SectionCaseInsensitiveSeps }
flags	**SectionFlags**
enum	**SplitBehavior** { KeepEmptyParts, SkipEmptyParts }
typedef	**const_iterator**
typedef	**const_reference**
typedef	**iterator**
typedef	**reference**
typedef	**value_type**

Public Functions

	QString()
	QString(const QChar * unicode, int size)
	QString(const QChar * unicode)
	QString(QChar ch)
	QString(int size, QChar ch)
	QString(const QLatin1String & str)
	QString(const QString & other)
	QString(const char * str)
	QString(const QByteArray & ba)
	~QString()
QString &	append(const QString & str)
QString &	append(const QStringRef & reference)
QString &	append(const QLatin1String & str)
QString &	append(const QByteArray & ba)
QString &	append(const char * str)
QString &	append(QChar ch)
QString	arg(const QString & a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(const QString & a1, const QString & a2) const
QString	arg(const QString & a1, const QString & a2, const QString & a3) const
QString	arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4) const
QString	arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5) const
QString	arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6) const
QString	arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6, const QString & a7) const
QString	arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6, const QString & a7, const QString & a8) const
QString	arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6, const QString & a7, const QString & a8, const QString & a9) const
QString	arg(int a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(uint a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(long a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(ulong a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(qlonglong a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(qulonglong a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(short a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(ushort a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(QChar a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(char a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(' ')) const
QString	arg(double a, int fieldWidth = 0, char format = 'g', int precision = -1, const QChar & fillChar = QLatin1Char(' ')) const
const QChar	at(int position) const
iterator	begin()
const_iterator	begin() const
int	capacity() const
void	chop(int n)

void	**clear**()
int	**compare**(const QString & other) const
int	**compare**(const QString & other, Qt::CaseSensitivity cs) const
int	**compare**(const QLatin1String & other, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**compare**(const QStringRef & ref, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
const_iterator	**constBegin**() const
const QChar *	**constData**() const
const_iterator	**constEnd**() const
bool	**contains**(const QString & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**contains**(const QStringRef & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**contains**(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**contains**(const QRegExp & rx) const
bool	**contains**(QRegExp & rx) const
int	**count**(const QString & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**count**(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**count**(const QStringRef & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**count**(const QRegExp & rx) const
int	**count**() const
QChar *	**data**()
const QChar *	**data**() const
iterator	**end**()
const_iterator	**end**() const
bool	**endsWith**(const QString & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**endsWith**(const QStringRef & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**endsWith**(const QLatin1String & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**endsWith**(const QChar & c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
QString &	**fill**(QChar ch, int size = -1)
int	**indexOf**(const QString & str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**indexOf**(const QLatin1String & str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**indexOf**(QChar ch, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**indexOf**(const QStringRef & str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**indexOf**(const QRegExp & rx, int from = 0) const
int	**indexOf**(QRegExp & rx, int from = 0) const
QString &	**insert**(int position, const QString & str)
QString &	**insert**(int position, const QLatin1String & str)
QString &	**insert**(int position, const QChar * unicode, int size)
QString &	**insert**(int position, QChar ch)
bool	**isEmpty**() const
bool	**isNull**() const
bool	**isRightToLeft**() const
int	**lastIndexOf**(const QString & str, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**lastIndexOf**(const QLatin1String & str, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**lastIndexOf**(QChar ch, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**lastIndexOf**(const QStringRef & str, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int	**lastIndexOf**(const QRegExp & rx, int from = -1) const
int	**lastIndexOf**(QRegExp & rx, int from = -1) const

QString	**left**(int n) const
QString	**leftJustified**(int width, QChar fill = QLatin1Char(' '), bool truncate = false) const
QStringRef	**leftRef**(int n) const
int	**length**() const
int	**localeAwareCompare**(const QStringRef & other) const
int	**localeAwareCompare**(const QString & other) const
QString	**mid**(int position, int n = -1) const
QStringRef	**midRef**(int position, int n = -1) const
QString	**normalized**(NormalizationForm mode) const
QString	**normalized**(NormalizationForm mode, QChar::UnicodeVersion version) const
QString &	**prepend**(const QString & str)
QString &	**prepend**(const QLatin1String & str)
QString &	**prepend**(const QByteArray & ba)
QString &	**prepend**(const char * str)
QString &	**prepend**(QChar ch)
void	**push_back**(const QString & other)
void	**push_back**(QChar ch)
void	**push_front**(const QString & other)
void	**push_front**(QChar ch)
QString &	**remove**(int position, int n)
QString &	**remove**(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**remove**(const QString & str, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**remove**(const QRegExp & rx)
QString	**repeated**(int times) const
QString &	**replace**(int position, int n, const QString & after)
QString &	**replace**(int position, int n, const QChar * unicode, int size)
QString &	**replace**(int position, int n, QChar after)
QString &	**replace**(const QString & before, const QString & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(const QChar * before, int blen, const QChar * after, int alen, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(QChar ch, const QString & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(QChar before, QChar after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(const QLatin1String & before, const QLatin1String & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(const QLatin1String & before, const QString & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(const QString & before, const QLatin1String & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(QChar c, const QLatin1String & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &	**replace**(const QRegExp & rx, const QString & after)
void	**reserve**(int size)
void	**resize**(int size)
QString	**right**(int n) const
QString	**rightJustified**(int width, QChar fill = QLatin1Char(' '), bool truncate = false) const
QStringRef	**rightRef**(int n) const
QString	**section**(QChar sep, int start, int end = -1, SectionFlags flags = SectionDefault) const
QString	**section**(const QString & sep, int start, int end = -1, SectionFlags flags = SectionDefault) const
QString	**section**(const QRegExp & reg, int start, int end = -1, SectionFlags flags = SectionDefault) const
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**(qlonglong n, int base = 10)
QString &	**setNum**(qulonglong n, int base = 10)
QString &	**setNum**(short n, int base = 10)
QString &	**setNum**(ushort n, int base = 10)
QString &	**setNum**(double n, char format = 'g', int precision = 6)
QString &	**setNum**(float n, char format = 'g', int precision = 6)
QString &	**setRawData**(const QChar * unicode, int size)
QString &	**setUnicode**(const QChar * unicode, int size)
QString &	**setUtf16**(const ushort * unicode, int size)
QString	**simplified**() const
int	**size**() const
QStringList	**split**(const QString & sep, SplitBehavior behavior = KeepEmptyParts, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
QStringList	**split**(const QChar & sep, SplitBehavior behavior = KeepEmptyParts, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
QStringList	**split**(const QRegExp & rx, SplitBehavior behavior = KeepEmptyParts) const
QString &	**sprintf**(const char * cformat, ...)
void	**squeeze**()
bool	**startsWith**(const QString & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**startsWith**(const QLatin1String & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**startsWith**(const QChar & c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool	**startsWith**(const QStringRef & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
void	**swap**(QString & other)
QByteArray	**toAscii**() const
QString	**toCaseFolded**() const
double	**toDouble**(bool * ok = 0) const
float	**toFloat**(bool * ok = 0) const
int	**toInt**(bool * ok = 0, int base = 10) const
QByteArray	**toLatin1**() const
QByteArray	**toLocal8Bit**() const
long	**toLong**(bool * ok = 0, int base = 10) const
qlonglong	**toLongLong**(bool * ok = 0, int base = 10) const
QString	**toLower**() const
short	**toShort**(bool * ok = 0, int base = 10) const
std::string	**toStdString**() const
std::wstring	**toStdWString**() const
uint	**toUInt**(bool * ok = 0, int base = 10) const
ulong	**toULong**(bool * ok = 0, int base = 10) const
qulonglong	**toULongLong**(bool * ok = 0, int base = 10) const
ushort	**toUShort**(bool * ok = 0, int base = 10) const
QVector<uint>	**toUcs4**() const
QString	**toUpper**() const
QByteArray	**toUtf8**() const
int	**toWCharArray**(wchar_t * array) const
QString	**trimmed**() const
void	**truncate**(int position)

const QChar *	**unicode**() const
const ushort *	**utf16**() const
QString &	**vsprintf**(const char * cformat, va_list ap)
bool	**operator!=**(const QString & other) const
bool	**operator!=**(const QLatin1String & other) const
bool	**operator!=**(const QByteArray & other) const
bool	**operator!=**(const char * other) const
QString &	**operator+=**(const QString & other)
QString &	**operator+=**(const QLatin1String & str)
QString &	**operator+=**(const QByteArray & ba)
QString &	**operator+=**(const char * str)
QString &	**operator+=**(const QStringRef & str)
QString &	**operator+=**(char ch)
QString &	**operator+=**(QChar ch)
bool	**operator<**(const QString & other) const
bool	**operator<**(const QLatin1String & other) const
bool	**operator<**(const QByteArray & other) const
bool	**operator<**(const char * other) const
bool	**operator<=**(const QString & other) const
bool	**operator<=**(const QLatin1String & other) const
bool	**operator<=**(const QByteArray & other) const
bool	**operator<=**(const char * other) const
QString &	**operator=**(const QString & other)
QString &	**operator=**(QString && other)
QString &	**operator=**(const QLatin1String & str)
QString &	**operator=**(const QByteArray & ba)
QString &	**operator=**(const char * str)
QString &	**operator=**(char ch)
QString &	**operator=**(QChar ch)
bool	**operator==**(const QString & other) const
bool	**operator==**(const QLatin1String & other) const
bool	**operator==**(const QByteArray & other) const
bool	**operator==**(const char * other) const
bool	**operator>**(const QString & other) const
bool	**operator>**(const QLatin1String & other) const
bool	**operator>**(const QByteArray & other) const
bool	**operator>**(const char * other) const
bool	**operator>=**(const QString & other) const
bool	**operator>=**(const QLatin1String & other) const
bool	**operator>=**(const QByteArray & other) const
bool	**operator>=**(const char * other) const
QCharRef	**operator[]**(int position)
const QChar	**operator[]**(int position) const
QCharRef	**operator[]**(uint position)
const QChar	**operator[]**(uint position) const

Static Public Members

int	**compare**(const QString & s1, const QString & s2, Qt::CaseSensitivity cs)
int	**compare**(const QString & s1, const QString & s2)
int	**compare**(const QString & s1, const QLatin1String & s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)
int	**compare**(const QLatin1String & s1, const QString & s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)
int	**compare**(const QString & s1, const QStringRef & s2, Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString	**fromAscii**(const char * str, int size = -1)
QString	**fromLatin1**(const char * str, int size = -1)
QString	**fromLocal8Bit**(const char * str, int size = -1)
QString	**fromRawData**(const QChar * unicode, int size)
QString	**fromStdString**(const std::string & str)
QString	**fromStdWString**(const std::wstring & str)
QString	**fromUcs4**(const uint * unicode, int size = -1)
QString	**fromUtf8**(const char * str, int size = -1)
QString	**fromUtf16**(const ushort * unicode, int size = -1)
QString	**fromWCharArray**(const wchar_t * string, int size = -1)
int	**localeAwareCompare**(const QString & s1, const QString & s2)
int	**localeAwareCompare**(const QString & s1, const QStringRef & s2)
QString	**number**(long n, int base = 10)
QString	**number**(double n, char format = 'g', int precision = 6)
QString	**number**(ulong n, int base = 10)
QString	**number**(int n, int base = 10)
QString	**number**(uint n, int base = 10)
QString	**number**(qlonglong n, int base = 10)
QString	**number**(qulonglong n, int base = 10)

Related Non-Members

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+**(char ch, const QString & s)
const QString	**operator+**(const QString & s, char ch)
bool	**operator<**(const char * s1, const QString & s2)
QDataStream &	**operator<<**(QDataStream & stream, const QString & string)
bool	**operator<=**(const char * s1, const QString & s2)
bool	**operator==**(const char * s1, const QString & s2)
bool	**operator>**(const char * s1, const QString & s2)
bool	**operator>=**(const char * s1, const QString & s2)
QDataStream &	**operator>>**(QDataStream & stream, QString & string)

Macros

	QT_NO_CAST_FROM_ASCII
	QT_NO_CAST_TO_ASCII

Detailed Description

The QString class provides a Unicode character string.

QString stores a string of 16-bit QChars, where each QChar corresponds one Unicode 4.0 character. (Unicode characters with code values above 65535 are stored using surrogate pairs, i.e., two consecutive QChars.)

Unicode is an international standard that supports most of the writing systems in use today. It is a superset of US-ASCII (ANSI X3.4-1986) and Latin-1 (ISO 8859-1), and all the US-ASCII/Latin-1 characters are available at the same code positions.

Behind the scenes, QString uses implicit sharing (copy-on-write) to reduce memory usage and to avoid the needless copying of data. This also helps reduce the inherent overhead of storing 16-bit characters instead of 8-bit characters.

In addition to QString, Qt also provides the QByteArray class to store raw bytes and traditional 8-bit '\0'-terminated strings. For most purposes, QString is the class you want to use. It is used throughout the Qt API, and the Unicode support ensures that your applications will be easy to translate if you want to expand your application's market at some point. The two main cases where QByteArray is appropriate are when you need to store raw binary data, and when memory conservation is critical (e.g., with Qt for Embedded Linux).

Initializing a String

One way to initialize a QString is simply to pass a `const char *` to its constructor. For example, the following code creates a QString of size 5 containing the data "Hello":

```
QString str = "Hello";
```

QString converts the `const char *` data into Unicode using the fromAscii() function. By default, fromAscii() treats character above 128 as Latin-1 characters, but this can be changed by calling QTextCodec::setCodecForCStrings().

In all of the QString functions that take `const char *` parameters, the `const char *` is interpreted as a classic C-style '\0'-terminated string. It is legal for the `const char *` parameter to be 0.

You can also provide string data as an array of QChars:

```
static const QChar data[4] = { 0x0055, 0x006e, 0x10e3, 0x03a3 };
QString str(data, 4);
```

QString makes a deep copy of the QChar data, so you can modify it later without experiencing side effects. (If for performance reasons you don't want to take a deep copy of the character data, use QString::fromRawData() instead.)

Another approach is to set the size of the string using resize() and to initialize the data character per character. QString uses 0-based indexes, just like C++ arrays. To access the character at a particular index position, you can use operator[](). On non-const strings, operator[]() returns a reference to a character that can be used on the left side of an assignment. For example:

```
QString str;
```

```
str.resize(4);

str[0] = QChar('U');
str[1] = QChar('n');
str[2] = QChar(0x10e3);
str[3] = QChar(0x03a3);
```

For read-only access, an alternative syntax is to use the at() function:

```
QString str;

for (int i = 0; i < str.size(); ++i) {
    if (str.at(i) >= QChar('a') && str.at(i) <= QChar('f'))
        qDebug() << "Found character in range [a-f]";
}
```

The at() function can be faster than operator[](), because it never causes a deep copy to occur. Alternatively, use the left(), right(), or mid() functions to extract several characters at a time.

A QString can embed '\0' characters (QChar::Null). The size() function always returns the size of the whole string, including embedded '\0' characters.

After a call to the resize() function, newly allocated characters have undefined values. To set all the characters in the string to a particular value, use the fill() function.

QString provides dozens of overloads designed to simplify string usage. For example, if you want to compare a QString with a string literal, you can write code like this and it will work as expected:

```
QString str;

if (str == "auto" || str == "extern"
        || str == "static" || str == "register") {
    // ...
}
```

You can also pass string literals to functions that take QStrings as arguments, invoking the QString(const char *) constructor. Similarly, you can pass a QString to a function that takes a `const char *` argument using the qPrintable() macro which returns the given QString as a `const char *`. This is equivalent to calling <QString>.toLocal8Bit().constData().

Manipulating String Data

QString provides the following basic functions for modifying the character data: append(), prepend(), insert(), replace(), and remove(). For example:

```
QString str = "and";
str.prepend("rock ");     // str == "rock and"
str.append(" roll");        // str == "rock and roll"
str.replace(5, 3, "&");   // str == "rock & roll"
```

If you are building a QString gradually and know in advance approximately how many characters the QString will contain, you can call reserve(), asking QString to preallocate a certain amount of memory. You can also call capacity() to find out how much memory QString actually allocated.

The replace() and remove() functions' first two arguments are the position from which to start erasing and the number of characters that should be erased. If you want to replace all occurrences of a particular substring with another, use one of the two-parameter replace() overloads.

A frequent requirement is to remove whitespace characters from a string ('\n', '\t', ' ', etc.). If you want to remove whitespace from both ends of a QString, use the trimmed() function. If you want to remove whitespace from both ends and replace multiple consecutive whitespaces with a single space character within the string, use simplified().

If you want to find all occurrences of a particular character or substring in a QString, use the indexOf() or lastIndexOf() functions. The former searches forward starting from a given index position, the latter searches backward. Both return the index position of the character or substring if they find it; otherwise, they return -1. For example, here's a typical loop that finds all occurrences of a particular substring:

```
QString str = "We must be <b>bold</b>, very <b>bold</b>";
int j = 0;

while ((j = str.indexOf("<b>", j)) != -1) {
    qDebug() << "Found <b> tag at index position" << j;
    ++j;
}
```

QString provides many functions for converting numbers into strings and strings into numbers. See the arg() functions, the setNum() functions, the number() static functions, and the toInt(), toDouble(), and similar functions.

To get an upper- or lowercase version of a string use toUpper() or toLower().

Lists of strings are handled by the QStringList class. You can split a string into a list of strings using the split() function, 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 QRegExp using the QStringList::filter() function.

Querying String Data

If you want to see if a QString starts or ends with a particular substring use startsWith() or endsWith(). If you simply want to check whether a QString contains a particular character or substring, use the contains() function. If you want to find out how many times a particular character or substring occurs in the string, use count().

QStrings can be compared using overloaded operators such as operator<(), operator<=(), operator==(), operator>=(), and so on. Note that the comparison is based exclusively on the numeric Unicode values of the characters. It is very fast, but is not what a human would expect; the QString::localeAwareCompare() function is a better choice for sorting user-interface strings.

To obtain a pointer to the actual character data, call data() or constData(). These functions return a pointer to the beginning of the QChar data. The pointer is guaranteed to remain valid until a non-const function is called on the QString.

Converting Between 8-Bit Strings and Unicode Strings

QString provides the following four functions that return a `const char *` version of the string as QByteArray: toAscii(), toLatin1(), toUtf8(), and toLocal8Bit().

- toAscii() returns an 8-bit string encoded using the codec specified by QTextCodec::codecForCStrings (by default, that is Latin 1).
- toLatin1() returns a Latin-1 (ISO 8859-1) encoded 8-bit string.
- toUtf8() returns a UTF-8 encoded 8-bit string. UTF-8 is a superset of US-ASCII (ANSI X3.4-1986) that supports the entire Unicode character set through multibyte sequences.
- toLocal8Bit() returns an 8-bit string using the system's local encoding.

To convert from one of these encodings, QString provides fromAscii(), fromLatin1(), fromUtf8(), and fromLocal8Bit(). Other encodings are supported through the QTextCodec class.

As mentioned above, QString provides a lot of functions and operators that make it easy to interoperate with `const char *` strings. But this functionality is a double-edged sword: It makes QString more convenient to use if all strings are US-ASCII or Latin-1, but there is

this functionality is a double-edged sword. It makes QString more convenient to use if all strings are US-ASCII or Latin-1, but there is always the risk that an implicit conversion from or to `const char *` is done using the wrong 8-bit encoding. To minimize these risks, you can turn off these implicit conversions by defining the following two preprocessor symbols:

- `QT_NO_CAST_FROM_ASCII` disables automatic conversions from C string literals and pointers to Unicode.
- `QT_NO_CAST_TO_ASCII` disables automatic conversion from QString to C strings.

One way to define these preprocessor symbols globally for your application is to add the following entry to your qmake project file:

```
DEFINES += QT_NO_CAST_FROM_ASCII \
           QT_NO_CAST_TO_ASCII
```

You then need to explicitly call fromAscii(), fromLatin1(), fromUtf8(), or fromLocal8Bit() to construct a QString from an 8-bit string, or use the lightweight QLatin1String class, for example:

```
QString url = QLatin1String("http://www.unicode.org/");
```

Similarly, you must call toAscii(), toLatin1(), toUtf8(), or toLocal8Bit() explicitly to convert the QString to an 8-bit string. (Other encodings are supported through the QTextCodec class.)

Note for C Programmers

Due to C++'s type system and the fact that QString is implicitly shared, QStrings may be treated like `int`s or other basic types. For example:

```
QString Widget::boolToString(bool b)
{
    QString result;
    if (b)
        result = "True";
    else
        result = "False";
    return result;
}
```

The `result` variable, is a normal variable allocated on the stack. When `return` is called, and 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.

Distinction Between Null and Empty Strings

For historical reasons, QString distinguishes between a null string and an empty string. A null string is a string that is initialized using QString's default constructor or by passing (const char *)0 to the constructor. An empty string is any string with size 0. A null string is always empty, but an empty string isn't necessarily null:

```
QString().isNull();               // returns true
QString().isEmpty();              // returns true

QString("").isNull();             // returns false
QString("").isEmpty();            // returns true

QString("abc").isNull();          // returns false
QString("abc").isEmpty();         // returns false
```

All functions except isNull() treat null strings the same as empty strings. For example, toAscii().constData() returns a pointer to a '\0' character for a null string (not a null pointer), and QString() compares equal to QString(""). We recommend that you always use the isEmpty() function and avoid isNull().

Argument Formats

In member functions where an argument format can be specified (e.g., arg(), number()), the argument format can be one of the following:

Format	Meaning
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

A precision is also specified with the argument format. For the 'e', 'E', and 'f' formats, the precision represents the number of digits after the decimal point. For the 'g' and 'G' formats, the precision represents the maximum number of significant digits (trailing zeroes are omitted).

More Efficient String Construction

Using the QString '+' operator, it is easy to construct a complex string from multiple substrings. You will often write code like this:

```
QString foo;
QString type = "long";

foo->setText(QLatin1String("vector<") + type + QLatin1String(">::iterator"));

if (foo.startsWith("(" + type + ") 0x"))
    ...
```

There is nothing wrong with either of these string constructions, but there are a few hidden inefficiencies. Beginning with Qt 4.6, you can eliminate them.

First, multiple uses of the '+' operator usually means multiple memory allocations. When concatenating n substrings, where n > 2, there can be as many as n - 1 calls to the memory allocator.

Second, QLatin1String does not store its length internally but calls qstrlen() when it needs to know its length.

In 4.6, an internal template class QStringBuilder has been added along with a few helper functions. This class is marked internal and does not appear in the documentation, because you aren't meant to instantiate it in your code. Its use will be automatic, as described below. The class is found in src/corelib/tools/qstringbuilder.cpp if you want to have a look at it.

QStringBuilder uses expression templates and reimplements the '%' operator so that when you use '%' for string concatenation instead of '+', multiple substring concatenations will be postponed until the final result is about to be assigned to a QString. At this point, the amount of memory required for the final result is known. The memory allocator is then called once to get the required space, and the substrings are copied into it one by one.

QLatin1Literal is a second internal class that can replace QLatin1String, which can't be changed for compatibility reasons. QLatin1Literal stores its length, thereby saving time when QStringBuilder computes the amount of memory required for the final string.

Additional efficiency is gained by inlining and reduced reference counting (the QString created from a QStringBuilder typically has a ref count of 1, whereas QString::append() needs an extra test).

There are three ways you can access this improved method of string construction. The straightforward way is to include QStringBuilder wherever you want to use it, and use the '%' operator instead of '+' when concatenating strings:

QStringBuilder wherever you want to use it, and use the '%' operator instead of '+' when concatenating strings:

```
#include <QStringBuilder>

QString hello("hello");
QStringRef el(&hello, 2, 3);
QLatin1String world("world");
QString message =  hello % el % world % QChar('!');
```

A more global approach which is the most convenient but not entirely source compatible, is to this define in your .pro file:

```
DEFINES *= QT_USE_QSTRINGBUILDER
```

and the '+' will automatically be performed as the QStringBuilder '%' everywhere.

See also fromRawData(), QChar, QLatin1String, QByteArray, and QStringRef.

Member Type Documentation

typedef QString::ConstIterator

Qt-style synonym for QString::const_iterator.

typedef QString::Iterator

Qt-style synonym for QString::iterator.

enum QString::NormalizationForm

This enum describes the various normalized forms of Unicode text.

Constant	Value	Description
QString::NormalizationForm_D	0	Canonical Decomposition
QString::NormalizationForm_C	1	Canonical Decomposition followed by Canonical Composition
QString::NormalizationForm_KD	2	Compatibility Decomposition
QString::NormalizationForm_KC	3	Compatibility Decomposition followed by Canonical Composition

See also normalized() and Unicode Standard Annex #15.

enum QString::SectionFlag
flags QString::SectionFlags

This enum specifies flags that can be used to affect various aspects of the section() function's behavior with respect to separators and empty fields.

Constant	Value	Description
QString::SectionDefault	0x00	Empty fields are counted, leading and trailing separators are not included,

`QString::SectionDefault`	0x00	and the separator is compared case sensitively.
`QString::SectionSkipEmpty`	0x01	Treat empty fields as if they don't exist, i.e. they are not considered as far as start and end are concerned.
`QString::SectionIncludeLeadingSep`	0x02	Include the leading separator (if any) in the result string.
`QString::SectionIncludeTrailingSep`	0x04	Include the trailing separator (if any) in the result string.
`QString::SectionCaseInsensitiveSeps`	0x08	Compare the separator case-insensitively.

The SectionFlags type is a typedef for QFlags<SectionFlag>. It stores an OR combination of SectionFlag values.

See also section().

enum QString::SplitBehavior

This enum specifies how the split() function should behave with respect to empty strings.

Constant	Value	Description
`QString::KeepEmptyParts`	0	If a field is empty, keep it in the result.
`QString::SkipEmptyParts`	1	If a field is empty, don't include it in the result.

See also split().

typedef QString::const_iterator

The QString::const_iterator typedef provides an STL-style const iterator for QString.

See also QString::iterator.

typedef QString::const_reference

The QString::const_reference typedef provides an STL-style const reference for QString.

This typedef was introduced in Qt 4.8.

typedef QString::iterator

The QString::iterator typedef provides an STL-style non-const iterator for QString.

See also QString::const_iterator.

typedef QString::reference

The QString::const_reference typedef provides an STL-style reference for QString.

This typedef was introduced in Qt 4.8.

typedef QString::value_type

The QString::const_reference typedef provides an STL-style value type for QString.

This typedef was introduced in Qt 4.8.

Member Function Documentation

QString::QString()

Constructs a null string. Null strings are also empty.

See also isEmpty().

QString::QString(const QChar * unicode, int size)

Constructs a string initialized with the first size characters of the QChar array unicode.

QString makes a deep copy of the string data. The unicode data is copied as is and the Byte Order Mark is preserved if present.

QString::QString(const QChar * unicode)

Constructs a string initialized with the characters of the QChar array unicode, which must be terminated with a 0.

QString makes a deep copy of the string data. The unicode data is copied as is and the Byte Order Mark is preserved if present.

This function was introduced in Qt 4.7.

QString::QString(QChar ch)

Constructs a string of size 1 containing the character ch.

QString::QString(int size, QChar ch)

Constructs a string of the given size with every character set to ch.

See also fill().

QString::QString(const QLatin1String & str)

Constructs a copy of the Latin-1 string str.

See also fromLatin1().

QString::QString(const QString & other)

Constructs a copy of other.

This operation takes constant time, because QString is implicitly shared. This makes returning a QString from a function very fast. If a shared instance is modified, it will be copied (copy-on-write), and that takes linear time.

See also operator=().

QString::QString(const char * str)

Constructs a string initialized with the 8-bit string str. The given const char pointer is converted to Unicode using the fromAscii() function.

You can disable this constructor by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

See also fromAscii(), fromLatin1(), fromLocal8Bit(), and fromUtf8().

QString::QString(const QByteArray & ba)

Constructs a string initialized with the byte array ba. The given byte array is converted to Unicode using fromAscii(). Stops copying at the first 0 character, otherwise copies the entire byte array.

You can disable this constructor by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

See also fromAscii(), fromLatin1(), fromLocal8Bit(), and fromUtf8().

QString::~QString()

Destroys the string.

QString & QString::append(const QString & str)

Appends the string str onto the end of this string.

Example:

```
QString x = "free";
QString y = "dom";

x.append(y);
// x == "freedom"
```

This is the same as using the insert() function:

```
x.insert(x.size(), y);
```

The append() function is typically very fast (constant time), because QString preallocates extra space at the end of the string data so it can grow without reallocating the entire string each time.

See also operator+=(), prepend(), and insert().

QString & QString::append(const QStringRef & reference)

Appends the given string reference to this string and returns the result.

This function was introduced in Qt 4.4.

QString & QString::append(const QLatin1String & str)

This function overloads append().

Appends the Latin-1 string str to this string.

QString & QString::append(const QByteArray & ba)

This function overloads append().

Appends the byte array ba to this string. The given byte array is converted to Unicode using the fromAscii() function.

You can disable this function by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

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

This function overloads append().

Appends the string str to this string. The given const char pointer is converted to Unicode using the fromAscii() function.

You can disable this function by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::append(QChar ch)

This function overloads append().

Appends the character ch to this string.

QString QString::arg(const QString & a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(' ')) const

Returns a copy of this string with the lowest numbered place marker replaced by string a, i.e., %1, %2, ..., %99.

fieldWidth specifies the minimum amount of space that argument a shall occupy. If a requires less space than fieldWidth, it is padded to fieldWidth with character fillChar. A positive fieldWidth produces right-aligned text. A negative fieldWidth produces left-aligned text.

This example shows how we might create a `status` string for reporting progress while processing a list of files:

```
QString i;           // current file's number
QString total;       // number of files to process
QString fileName;    // current file's name

QString status = QString("Processing file %1 of %2: %3")
                 .arg(i).arg(total).arg(fileName);
```

First, `arg(i)` replaces %1. Then `arg(total)` replaces %2. Finally, `arg(fileName)` replaces %3.

One advantage of using arg() over sprintf() is that the order of the numbered place markers can change, if the application's strings are translated into other languages, but each arg() will still replace the lowest numbered unreplaced place marker, no matter where it appears.

Also, if place marker %i appears more than once in the string, the arg() replaces all of them.

If there is no unreplaced place marker remaining, a warning message is output and the result is undefined. Place marker numbers must be in the range 1 to 99.

QString QString::arg(const QString & a1, const QString & a2) const

This function overloads arg().

This is the same as `str.arg(a1).arg(a2)`, except that the strings a1 and a2 are replaced in one pass. This can make a difference if a1 contains e.g. %1:

```
QString str;
str = "%1 %2";

str.arg("%1f", "Hello");        // returns "%1f Hello"
str.arg("%1f").arg("Hello");    // returns "Hellof %2"
```

QString QString::arg(const QString & a1, const QString & a2, const QString & a3) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3)`, except that the strings a1, a2 and a3 are replaced in one pass.

QString QString::arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3).arg(a4)`, except that the strings a1, a2, a3 and a4 are replaced in one pass.

QString QString::arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3).arg(a4).arg(a5)`, except that the strings a1, a2, a3, a4, and a5 are replaced in one pass.

QString QString::arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3).arg(a4).arg(a5).arg(a6))`, except that the strings a1, a2, a3, a4, a5, and a6 are replaced in one pass.

QString QString::arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6, const QString & a7) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3).arg(a4).arg(a5).arg(a6).arg(a7)`, except that the strings a1, a2, a3, a4, a5, a6, and a7 are replaced in one pass.

QString QString::arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6, const QString & a7, const QString & a8) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3).arg(a4).arg(a5).arg(a6).arg(a7).arg(a8)`, except that the strings a1, a2, a3, a4, a5, a6, a7, and a8 are replaced in one pass.

QString QString::arg(const QString & a1, const QString & a2, const QString & a3, const QString & a4, const QString & a5, const QString & a6, const QString & a7, const QString & a8, const QString & a9) const

This function overloads arg().

This is the same as calling `str.arg(a1).arg(a2).arg(a3).arg(a4).arg(a5).arg(a6).arg(a7).arg(a8).arg(a9)`, except that the strings a1, a2, a3, a4, a5, a6, a7, a8, and a9 are replaced in one pass.

QString QString::arg(int a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

The a argument is expressed in base base, which is 10 by default and must be between 2 and 36. For bases other than 10, a is treated as an unsigned integer.

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The '%' can be followed by an 'L', in which case the sequence is replaced with a localized representation of a. The conversion uses the default locale, set by QLocale::setDefault(). If no default locale was specified, the "C" locale is used. The 'L' flag is ignored if base is not 10.

```
QString str;
str = QString("Decimal 63 is %1 in hexadecimal")
        .arg(63, 0, 16);
// str == "Decimal 63 is 3f in hexadecimal"

QLocale::setDefault(QLocale(QLocale::English, QLocale::UnitedStates));
str = QString("%1 %L2 %L3")
        .arg(12345)
        .arg(12345)
        .arg(12345, 0, 16);
// str == "12345 12,345 3039"
```

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(uint a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

The base argument specifies the base to use when converting the integer a into a string. The base must be between 2 and 36.

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(long a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The a argument is expressed in the given base, which is 10 by default and must be between 2 and 36.

The '%' can be followed by an 'L', in which case the sequence is replaced with a localized representation of a. The conversion uses the default locale. The default locale is determined from the system's locale settings at application startup. It can be changed using QLocale::setDefault(). The 'L' flag is ignored if base is not 10.

```
QString str;
str = QString("Decimal 63 is %1 in hexadecimal")
        .arg(63, 0, 16);
// str == "Decimal 63 is 3f in hexadecimal"

QLocale::setDefault(QLocale(QLocale::English, QLocale::UnitedStates));
str = QString("%1 %L2 %L3")
        .arg(12345)
        .arg(12345)
        .arg(12345, 0, 16);
// str == "12345 12,345 3039"
```

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(ulong a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The base argument specifies the base to use when converting the integer a to a string. The base must be between 2 and 36, with 8 giving octal, 10 decimal, and 16 hexadecimal numbers.

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(qlonglong a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The base argument specifies the base to use when converting the integer a into a string. The base must be between 2 and 36, with 8 giving octal, 10 decimal, and 16 hexadecimal numbers.

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(qulonglong a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The base argument specifies the base to use when converting the integer a into a string. base must be between 2 and 36, with 8 giving octal, 10 decimal, and 16 hexadecimal numbers.

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(short a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The base argument specifies the base to use when converting the integer a into a string. The base must be between 2 and 36, with 8 giving octal, 10 decimal, and 16 hexadecimal numbers.

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(ushort a, int fieldWidth = 0, int base = 10, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

The base argument specifies the base to use when converting the integer a into a string. The base must be between 2 and 36, with 8 giving octal, 10 decimal, and 16 hexadecimal numbers.

If fillChar is '0' (the number 0, ASCII 48), the locale's zero is used. For negative numbers, zero padding might appear before the minus sign.

QString QString::arg(QChar a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

QString QString::arg(char a, int fieldWidth = 0, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

The a argument is interpreted as a Latin-1 character.

QString QString::arg(double a, int fieldWidth = 0, char format = 'g', int precision = -1, const QChar & fillChar = QLatin1Char(' ')) const

This function overloads arg().

Argument a is formatted according to the specified format and precision. See Argument Formats for details.

fieldWidth specifies the minimum amount of space that a is padded to and filled with the character fillChar. A positive value produces right-aligned text; a negative value produces left-aligned text.

```
double d = 12.34;
QString str = QString("delta: %1").arg(d, 0, 'E', 3);
// str == "delta: 1.234E+01"
```

The '%' can be followed by an 'L', in which case the sequence is replaced with a localized representation of a. The conversion uses the default locale, set by QLocale::setDefaultLocale(). If no default locale was specified, the "C" locale is used.

If fillChar is '0' (the number 0, ASCII 48), this function will use the locale's zero to pad. For negative numbers, the zero padding will probably appear before the minus sign.

See also QLocale::toString().

const QChar QString::at(int position) const

Returns the character at the given index position in the string.

The position must be a valid index position in the string (i.e., 0 <= position < size()).

See also operator[]().

iterator QString::begin()

Returns an STL-style iterator pointing to the first character in the string.

See also constBegin() and end().

const_iterator QString::begin() const

This function overloads begin().

int QString::capacity() const

Returns the maximum number of characters that can be stored in the string without forcing a reallocation.

The sole purpose of this function is to provide a means of fine tuning QString's memory usage. In general, you will rarely ever need to call this function. If you want to know how many characters are in the string, call size().

See also reserve() and squeeze().

void QString::chop(int n)

Removes n characters from the end of the string.

If n is greater than size(), the result is an empty string.

Example:

```
QString str("LOGOUT\r\n");
str.chop(2);
// str == "LOGOUT"
```

If you want to remove characters from the beginning of the string, use remove() instead.

See also truncate(), resize(), and remove().

void QString::clear()

Clears the contents of the string and makes it empty.

See also resize() and isEmpty().

int QString::compare(const QString & s1, const QString & s2, Qt::CaseSensitivity cs) [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.

If cs is Qt::CaseSensitive, the comparison is case sensitive; otherwise the comparison is case insensitive.

Case sensitive 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-visible strings with localeAwareCompare().

```
int x = QString::compare("aUtO", "AuTo", Qt::CaseInsensitive);  // x == 0
int y = QString::compare("auto", "Car", Qt::CaseSensitive);     // y > 0
int z = QString::compare("auto", "Car", Qt::CaseInsensitive);   // z < 0
```

This function was introduced in Qt 4.2.

See also operator==(), operator<(), and operator>().

int QString::compare(const QString & s1, const QString & s2) [static]

This function overloads compare().

Performs a case sensitive compare of s1 and s2.

int QString::compare(const QString & s1, const QLatin1String & s2, Qt::CaseSensitivity cs = Qt::CaseSensitive) [static]

This function overloads compare().

Performs a comparison of s1 and s2, using the case sensitivity setting cs.

This function was introduced in Qt 4.2.

int QString::compare(const QLatin1String & s1, const QString & s2, Qt::CaseSensitivity cs = Qt::CaseSensitive) [static]

This function overloads compare().

Performs a comparison of s1 and s2, using the case sensitivity setting cs.

This function was introduced in Qt 4.2.

int QString::compare(const QString & other) const

This function overloads compare().

Lexically compares this string with the other string and returns an integer less than, equal to, or greater than zero if this string is less than, equal to, or greater than the other string.

Equivalent to `compare(*this, other)`.

int QString::compare(const QString & other, Qt::CaseSensitivity cs) const

This function overloads compare().

Same as compare(*this, other, cs).

This function was introduced in Qt 4.2.

int QString::compare(const QLatin1String & other, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads compare().

Same as compare(*this, other, cs).

This function was introduced in Qt 4.2.

int QString::compare(const QStringRef & ref, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads compare().

Compares the string reference, ref, with the string and returns an integer less than, equal to, or greater than zero if the string is less than, equal to, or greater than ref.

int QString::compare(const QString & s1, const QStringRef & s2, Qt::CaseSensitivity cs = Qt::CaseSensitive) [static]

This function overloads compare().

const_iterator QString::constBegin() const

Returns a const STL-style iterator pointing to the first character in the string.

Returns a const STL-style iterator pointing to the first character in the string.

See also begin() and constEnd().

const QChar * QString::constData() const

Returns a pointer to the data stored in the QString. The pointer can be used to access the characters that compose the string. For convenience, the data is '\0'-terminated.

Note that the pointer remains valid only as long as the string is not modified.

See also data() and operator[]().

const_iterator QString::constEnd() const

Returns a const STL-style iterator pointing to the imaginary item after the last item in the list.

See also constBegin() and end().

bool QString::contains(const QString & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns true if this string contains an occurrence of the string str; otherwise returns false.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString str = "Peter Pan";
str.contains("peter", Qt::CaseInsensitive);    // returns true
```

See also indexOf() and count().

bool QString::contains(const QStringRef & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns true if this string contains an occurrence of the string reference str; otherwise returns false.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.8.

See also indexOf() and count().

bool QString::contains(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads contains().

Returns true if this string contains an occurrence of the character ch; otherwise returns false.

bool QString::contains(const QRegExp & rx) const

This function overloads contains().

Returns true if the regular expression rx matches somewhere in this string; otherwise returns false.

bool QString::contains(QRegExp & rx) const

This function overloads contains().

Returns true if the regular expression rx matches somewhere in this string; otherwise returns false.

If there is a match, the rx regular expression will contain the matched captures (see QRegExp::matchedLength, QRegExp::cap).

This function was introduced in Qt 4.5.

int QString::count(const QString & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns the number of (potentially overlapping) occurrences of the string str in this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

See also contains() and indexOf().

int QString::count(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads count().

Returns the number of occurrences of character ch in the string.

int QString::count(const QStringRef & str, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads count().

Returns the number of (potentially overlapping) occurrences of the string reference str in this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.8.

See also contains() and indexOf().

int QString::count(const QRegExp & rx) const

This function overloads count().

Returns the number of times the regular expression rx matches in the string.

This function counts overlapping matches, so in the example below, there are four instances of "ana" or "ama":

```
QString str = "banana and panama";
str.count(QRegExp("a[nm]a"));    // returns 4
```

int QString::count() const

This function overloads count().

Same as size().

QChar * QString::data()

Returns a pointer to the data stored in the QString. The pointer can be used to access and modify the characters that compose the string. For convenience, the data is '\0'-terminated.

Example:

```
QString str = "Hello world";
QChar *data = str.data();
while (!data->isNull()) {
    qDebug() << data->unicode();
    ++data;
}
```

Note that the pointer remains valid only as long as the string is not modified by other means. For read-only access, constData() is faster because it never causes a deep copy to occur.

See also constData() and operator[]().

const QChar * QString::data() const

This is an overloaded function.

iterator QString::end()

Returns an STL-style iterator pointing to the imaginary character after the last character in the string.

See also begin() and constEnd().

const_iterator QString::end() const

This function overloads end().

bool QString::endsWith(const QString & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns true if the string ends with s; otherwise returns false.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

```
QString str = "Bananas";
str.endsWith("anas");         // returns true
str.endsWith("pple");         // returns false
```

See also startsWith().

bool QString::endsWith(const QStringRef & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads endsWith().

Returns true if the string ends with the string reference s; otherwise returns false.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.8.

See also startsWith().

bool QString::endsWith(const QLatin1String & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads endsWith().

bool QString::endsWith(const QChar & c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns true if the string ends with c; otherwise returns false.

This function overloads endsWith().

QString & QString::fill(QChar ch, int size = -1)

Sets every character in the string to character ch. If size is different from -1 (default), the string is resized to size beforehand.

Example:

```
QString str = "Berlin";
str.fill('z');
// str == "zzzzzz"

str.fill('A', 2);
// str == "AA"
```

See also resize().

QString QString::fromAscii(const char * str, int size = -1) [static]

Returns a QString initialized with the first size characters from the string str.

If size is -1 (default), it is taken to be qstrlen(str).

Note that, despite the name, this function actually uses the codec defined by QTextCodec::setCodecForCStrings() to convert str to Unicode. Depending on the codec, it may not accept valid US-ASCII (ANSI X3.4-1986) input. If no codec has been set, this function does the same as fromLatin1().

See also toAscii(), fromLatin1(), fromUtf8(), and fromLocal8Bit().

QString QString::fromLatin1(const char * str, int size = -1) [static]

Returns a QString initialized with the first size characters of the Latin-1 string str.

If size is -1 (default), it is taken to be qstrlen(str).

See also toLatin1(), fromAscii(), fromUtf8(), and fromLocal8Bit().

QString QString::fromLocal8Bit(const char * str, int size = -1) [static]

Returns a QString initialized with the first size characters of the 8-bit string str.

If size is -1 (default), it is taken to be qstrlen(str).

QTextCodec::codecForLocale() is used to perform the conversion.

See also toLocal8Bit(), fromAscii(), fromLatin1(), and fromUtf8().

QString QString::fromRawData(const QChar * unicode, int size) [static]

Constructs a QString that uses the first size Unicode characters in the array unicode. The data in unicode is not copied. The caller must be able to guarantee that unicode will not be deleted or modified as long as the QString (or an unmodified copy of it) exists.

Any attempts to modify the QString or copies of it will cause it to create a deep copy of the data, ensuring that the raw data isn't modified.

Here's an example of how we can use a QRegExp on raw data in memory without requiring to copy the data into a QString:

```
QRegExp pattern;
static const QChar unicode[] = {
        0x005A, 0x007F, 0x00A4, 0x0060,
        0x1009, 0x0020, 0x0020};
int size = sizeof(unicode) / sizeof(QChar);

QString str = QString::fromRawData(unicode, size);
if (str.contains(QRegExp(pattern))) {
    // ...
}
```

Warning: A string created with fromRawData() is not '\0'-terminated, unless the raw data contains a '\0' character at position size. This means unicode() will not return a '\0'-terminated string (although utf16() does, at the cost of copying the raw data).

See also fromUtf16() and setRawData().

QString QString::fromStdString(const std::string & str) [static]

Returns a copy of the str string. The given string is converted to Unicode using the fromAscii() function.

This constructor is only available if Qt is configured with STL compatibility enabled.

See also fromAscii(), fromLatin1(), fromLocal8Bit(), and fromUtf8().

QString QString::fromStdWString(const std::wstring & str) [static]

Returns a copy of the str string. The given string is assumed to be encoded in utf16 if the size of wchar_t is 2 bytes (e.g. on windows) and ucs4 if the size of wchar_t is 4 bytes (most Unix systems).

This method is only available if Qt is configured with STL compatibility enabled.

See also fromUtf16(), fromLatin1(), fromLocal8Bit(), fromUtf8(), and fromUcs4().

QString QString::fromUcs4(const uint * unicode, int size = -1) [static]

Returns a QString initialized with the first size characters of the Unicode string unicode (ISO-10646-UCS-4 encoded).

If size is -1 (default), unicode must be terminated with a 0.

This function was introduced in Qt 4.2.

See also toUcs4(), fromUtf16(), utf16(), setUtf16(), and fromWCharArray().

QString QString::fromUtf8(const char * str, int size = -1) [static]

Returns a QString initialized with the first size bytes of the UTF-8 string str.

If size is -1 (default), it is taken to be qstrlen(str).

UTF-8 is a Unicode codec and can represent all characters in a Unicode string like QString. However, invalid sequences are possible with UTF-8 and, if any such are found, they will be replaced with one or more "replacement characters", or suppressed. These include non-Unicode sequences, non-characters, overlong sequences or surrogate codepoints encoded into UTF-8.

Non-characters are codepoints that the Unicode standard reserves and must not be used in text interchange. They are the last two codepoints in each Unicode Plane (U+FFFE, U+FFFF, U+1FFFE, U+1FFFF, U+2FFFE, etc.), as well as 16 codepoints in the range U+FDD0..U+FDDF, inclusive.

See also toUtf8(), fromAscii(), fromLatin1(), and fromLocal8Bit().

QString QString::fromUtf16(const ushort * unicode, int size = -1) [static]

Returns a QString initialized with the first size characters of the Unicode string unicode (ISO-10646-UTF-16 encoded).

If size is -1 (default), unicode must be terminated with a 0.

This function checks for a Byte Order Mark (BOM). If it is missing, host byte order is assumed.

This function is slow compared to the other Unicode conversions. Use QString(const QChar *, int) or QString(const QChar *) if possible.

QString makes a deep copy of the Unicode data.

See also utf16() and setUtf16().

QString QString::fromWCharArray(const wchar_t * string, int size = -1) [static]

Returns a copy of the string, where the encoding of string depends on the size of wchar. If wchar is 4 bytes, the string is interpreted as ucs-4, if wchar is 2 bytes it is interpreted as ucs-2.

If size is -1 (default), the string has to be 0 terminated.

This function was introduced in Qt 4.2.

See also fromUtf16(), fromLatin1(), fromLocal8Bit(), fromUtf8(), fromUcs4(), and fromStdWString().

int QString::indexOf(const QString & str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive)

const

Returns the index position of the first occurrence of the string str in this string, searching forward from index position from. Returns -1 if str is not found.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString x = "sticky question";
QString y = "sti";
x.indexOf(y);               // returns 0
x.indexOf(y, 1);            // returns 10
x.indexOf(y, 10);           // returns 10
x.indexOf(y, 11);           // returns -1
```

If from is -1, the search starts at the last character; if it is -2, at the next to last character and so on.

See also lastIndexOf(), contains(), and count().

int QString::indexOf(const QLatin1String & str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns the index position of the first occurrence of the string str in this string, searching forward from index position from. Returns -1 if str is not found.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString x = "sticky question";
QString y = "sti";
x.indexOf(y);               // returns 0
x.indexOf(y, 1);            // returns 10
x.indexOf(y, 10);           // returns 10
x.indexOf(y, 11);           // returns -1
```

If from is -1, the search starts at the last character; if it is -2, at the next to last character and so on.

This function was introduced in Qt 4.5.

See also lastIndexOf(), contains(), and count().

int QString::indexOf(QChar ch, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads indexOf().

Returns the index position of the first occurrence of the character ch in the string, searching forward from index position from. Returns -1 if ch could not be found.

int QString::indexOf(const QStringRef & str, int from = 0, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads indexOf().

Returns the index position of the first occurrence of the string reference str in this string, searching forward from index position from. Returns -1 if str is not found.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.8.

int QString::indexOf(const QRegExp & rx, int from = 0) const

This function overloads indexOf().

Returns the index position of the first match of the regular expression rx in the string, searching forward from index position from. Returns -1 if rx didn't match anywhere.

Example:

```
QString str = "the minimum";
str.indexOf(QRegExp("m[aeiou]"), 0);       // returns 4
```

int QString::indexOf(QRegExp & rx, int from = 0) const

This function overloads indexOf().

Returns the index position of the first match of the regular expression rx in the string, searching forward from index position from. Returns -1 if rx didn't match anywhere.

If there is a match, the rx regular expression will contain the matched captures (see QRegExp::matchedLength, QRegExp::cap).

Example:

```
QString str = "the minimum";
str.indexOf(QRegExp("m[aeiou]"), 0);       // returns 4
```

This function was introduced in Qt 4.5.

QString & QString::insert(int position, const QString & str)

Inserts the string str at the given index position and returns a reference to this string.

Example:

```
QString str = "Meal";
str.insert(1, QString("ontr"));
// str == "Montreal"
```

If the given position is greater than size(), the array is first extended using resize().

See also append(), prepend(), replace(), and remove().

QString & QString::insert(int position, const QLatin1String & str)

This function overloads insert().

Inserts the Latin-1 string str at the given index position.

QString & QString::insert(int position, const QChar * unicode, int size)

This function overloads insert().

Inserts the first size characters of the QChar array unicode at the given index position in the string.

QString & QString::insert(int position, QChar ch)

This function overloads insert().

Inserts ch at the given index position in the string.

bool QString::isEmpty() const

Returns true if the string has no characters; otherwise returns false.

Example:

```
QString().isEmpty();            // returns true
QString("").isEmpty();          // returns true
QString("x").isEmpty();         // returns false
QString("abc").isEmpty();       // returns false
```

See also size().

bool QString::isNull() const

Returns true if this string is null; otherwise returns false.

Example:

```
QString().isNull();             // returns true
QString("").isNull();           // returns false
QString("abc").isNull();        // returns false
```

Qt makes a distinction between null strings and empty strings for historical reasons. For most applications, what matters is whether or not a string contains any data, and this can be determined using the isEmpty() function.

See also isEmpty().

bool QString::isRightToLeft() const

Returns true if the string is read right to left.

Returns true if the string is read right to left.

int QString::lastIndexOf(const QString & str, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns the index position of the last occurrence of the string str in this string, searching backward from index position from. If from is -1 (default), the search starts at the last character; if from is -2, at the next to last character and so on. Returns -1 if str is not found.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString x = "crazy azimuths";
QString y = "az";
x.lastIndexOf(y);           // returns 6
x.lastIndexOf(y, 6);        // returns 6
x.lastIndexOf(y, 5);        // returns 2
x.lastIndexOf(y, 1);        // returns -1
```

See also indexOf(), contains(), and count().

int QString::lastIndexOf(const QLatin1String & str, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads lastIndexOf().

Returns the index position of the last occurrence of the string str in this string, searching backward from index position from. If from is -1 (default), the search starts at the last character; if from is -2, at the next to last character and so on. Returns -1 if str is not found.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString x = "crazy azimuths";
QString y = "az";
x.lastIndexOf(y);           // returns 6
x.lastIndexOf(y, 6);        // returns 6
x.lastIndexOf(y, 5);        // returns 2
x.lastIndexOf(y, 1);        // returns -1
```

This function was introduced in Qt 4.5.

See also indexOf(), contains(), and count().

int QString::lastIndexOf(QChar ch, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads lastIndexOf().

Returns the index position of the last occurrence of the character ch, searching backward from position from.

int QString::lastIndexOf(const QStringRef & str, int from = -1, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads lastIndexOf().

Returns the index position of the last occurrence of the string reference str in this string, searching backward from index position from. If from is -1 (default), the search starts at the last character; if from is -2, at the next to last character and so on. Returns -1 if str is not found.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.8.

See also indexOf(), contains(), and count().

int QString::lastIndexOf(const QRegExp & rx, int from = -1) const

This function overloads lastIndexOf().

Returns the index position of the last match of the regular expression rx in the string, searching backward from index position from. Returns -1 if rx didn't match anywhere.

Example:

```
QString str = "the minimum";
str.lastIndexOf(QRegExp("m[aeiou]"));      // returns 8
```

int QString::lastIndexOf(QRegExp & rx, int from = -1) const

This function overloads lastIndexOf().

Returns the index position of the last match of the regular expression rx in the string, searching backward from index position from. Returns -1 if rx didn't match anywhere.

If there is a match, the rx regular expression will contain the matched captures (see QRegExp::matchedLength, QRegExp::cap).

Example:

```
QString str = "the minimum";
str.lastIndexOf(QRegExp("m[aeiou]"));      // returns 8
```

This function was introduced in Qt 4.5.

QString QString::left(int n) const

Returns a substring that contains the n leftmost characters of the string.

The entire string is returned if n is greater than size() or less than zero.

```
QString x = "Pineapple";
QString y = x.left(4);      // y == "Pine"
```

See also right(), mid(), and startsWith().

QString QString::leftJustified(int width, QChar fill = QLatin1Char(' '), bool truncate = false) const

Returns a string of size width that contains this string padded by the fill character.

If truncate is false and the size() of the string is more than width, then the returned string is a copy of the string.

```
QString s = "apple";
QString t = s.leftJustified(8, '.');    // t == "apple..."
```

If truncate is true and the size() of the string is more than width, then any characters in a copy of the string after position width are removed, and the copy is returned.

```
QString str = "Pineapple";
str = str.leftJustified(5, '.', true);    // str == "Pinea"
```

See also rightJustified().

QStringRef QString::leftRef(int n) const

Returns a substring reference to the n leftmost characters of the string.

If n is greater than size() or less than zero, a reference to the entire string is returned.

```
QString x = "Pineapple";
QStringRef y = x.leftRef(4);        // y == "Pine"
```

This function was introduced in Qt 4.4.

See also left(), rightRef(), midRef(), and startsWith().

int QString::length() const

Returns the number of characters in this string. Equivalent to size().

See also setLength() and resize().

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.

On Mac OS X since Qt 4.3, this function compares according the "Order for sorted lists" setting in the International prefereces panel.

See also compare() and QTextCodec::locale().

int QString::localeAwareCompare(const QStringRef & other) const

This function overloads localeAwareCompare().

Compares this string with the other string and returns an integer less than, equal to, or greater than zero if this string is less than, equal to, or greater than the other string.

The comparison is performed in a locale- and also platform-dependent manner. Use this function to present sorted lists of strings to the user.

Same as `localeAwareCompare(*this, other)`.

This function was introduced in Qt 4.5.

int QString::localeAwareCompare(const QString & s1, const QStringRef & s2) `[static]`

This function overloads localeAwareCompare().

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.

This function was introduced in Qt 4.5.

int QString::localeAwareCompare(const QString & other) const

This function overloads localeAwareCompare().

Compares this string with the other string and returns an integer less than, equal to, or greater than zero if this string is less than, equal to, or greater than the other string.

The comparison is performed in a locale- and also platform-dependent manner. Use this function to present sorted lists of strings to the user.

Same as `localeAwareCompare(*this, other)`.

QString QString::mid(int position, int n = -1) const

Returns a string that contains n characters of this string, starting at the specified position index.

Returns a null string if the position index exceeds the length of the string. If there are less than n characters available in the string starting at the given position, or if n is -1 (default), the function returns all characters that are available from the specified position.

Example:

```
QString x = "Nine pineapples";
QString y = x.mid(5, 4);            // y == "pine"
QString z = x.mid(5);               // z == "pineapples"
```

See also left() and right().

QStringRef QString::midRef(int position, int n = -1) const

Returns a substring reference to n characters of this string, starting at the specified position.

If the position exceeds the length of the string, an empty reference is returned.

If there are less than n characters available in the string, starting at the given position, or if n is -1 (default), the function returns all characters from the specified position onwards.

Example:

```
QString x = "Nine pineapples";
QStringRef y = x.midRef(5, 4);      // y == "pine"
QStringRef z = x.midRef(5);         // z == "pineapples"
```

This function was introduced in Qt 4.4.

See also mid(), leftRef(), and rightRef().

QString QString::normalized(NormalizationForm mode) const

Returns the string in the given Unicode normalization mode.

QString QString::normalized(NormalizationForm mode, QChar::UnicodeVersion version) const

This is an overloaded function.

Returns the string in the given Unicode normalization mode, according to the given version of the Unicode standard.

QString QString::number(long n, int base = 10) [static]

Returns a string equivalent of the number n according to the specified base.

The base is 10 by default and must be between 2 and 36. For bases other than 10, n is treated as an unsigned integer.

```
long a = 63;
QString s = QString::number(a, 16);             // s == "3f"
QString t = QString::number(a, 16).toUpper();     // t == "3F"
```

See also setNum().

QString QString::number(double n, char format = 'g', int precision = 6) [static]

Returns a string equivalent of the number n, formatted according to the specified format and precision. See Argument Formats for details.

Unlike QLocale::toString(), this function does not honor the user's locale settings.

See also setNum() and QLocale::toString().

QString QString::number(ulong n, int base = 10) [static]

This is an overloaded function.

QString QString::number(int n, int base = 10) [static]

This is an overloaded function.

QString QString::number(uint n, int base = 10) [static]

This is an overloaded function.

QString QString::number(qlonglong n, int base = 10) [static]

This is an overloaded function.

QString QString::number(qulonglong n, int base = 10) [static]

This is an overloaded function.

QString & QString::prepend(const QString & str)

Prepends the string str to the beginning of this string and returns a reference to this string.

Example:

```
QString x = "ship";
QString y = "air";
x.prepend(y);
// x == "airship"
```

See also append() and insert().

QString & QString::prepend(const QLatin1String & str)

This function overloads prepend().

Prepends the Latin-1 string str to this string.

QString & QString::prepend(const QByteArray & ba)

This function overloads prepend().

Prepends the byte array ba to this string. The byte array is converted to Unicode using the fromAscii() function.

You can disable this function by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::prepend(const char * str)

This function overloads prepend().

Prepends the string str to this string. The const char pointer is converted to Unicode using the fromAscii() function.

You can disable this function by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::prepend(QChar ch)

This function overloads prepend().

Prepends the character ch to this string.

void QString::push_back(const QString & other)

This function is provided for STL compatibility, appending the given other string onto the end of this string. It is equivalent to `append(other)`.

See also append().

void QString::push_back(QChar ch)

This is an overloaded function.

Appends the given ch character onto the end of this string.

void QString::push_front(const QString & other)

This function is provided for STL compatibility, prepending the given other string to the beginning of this string. It is equivalent to `prepend(other)`.

See also prepend().

void QString::push_front(QChar ch)

This is an overloaded function.

Prepends the given ch character to the beginning of this string.

QString & QString::remove(int position, int n)

Removes n characters from the string, starting at the given position index, and returns a reference to the string.

If the specified position index is within the string, but position + n is beyond the end of the string, the string is truncated at the specified position.

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

See also insert() and replace().

QString & QString::remove(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive)

Removes every occurrence of the character ch in this string, and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString t = "Ali Baba";
t.remove(QChar('a'), Qt::CaseInsensitive);
// t == "li Bb"
```

This is the same as `replace(ch, "", cs)`.

See also replace().

QString & QString::remove(const QString & str, Qt::CaseSensitivity cs = Qt::CaseSensitive)

Removes every occurrence of the given str string in this string, and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This is the same as `replace(str, "", cs)`.

See also replace().

QString & QString::remove(const QRegExp & rx)

Removes every occurrence of the regular expression rx in the string, and returns a reference to the string. For example:

```
QString r = "Telephone";
r.remove(QRegExp("[aeiou]."));
// r == "The"
```

See also indexOf(), lastIndexOf(), and replace().

QString QString::repeated(int times) const

Returns a copy of this string repeated the specified number of times.

If times is less than 1, an empty string is returned.

Example:

```
QString str("ab");
str.repeated(4);            // returns "abababab"
```

This function was introduced in Qt 4.5.

QString & QString::replace(int position, int n, const QString & after)

Replaces n characters beginning at index position with the string after and returns a reference to this string.

Example:

```
QString x = "Say yes!";
QString y = "no";
x.replace(4, 3, y);
// x == "Say no!"
```

See also insert() and remove().

QString & QString::replace(int position, int n, const QChar * unicode, int size)

This function overloads replace().

Replaces n characters beginning at index position with the first size characters of the QChar array unicode and returns a reference to this string.

QString & QString::replace(int position, int n, QChar after)

This function overloads replace().

Replaces n characters beginning at index position with the character after and returns a reference to this string.

QString & QString::replace(const QString & before, const QString & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the string before with the string after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Example:

```
QString str = "colour behaviour flavour neighbour";
str.replace(QString("ou"), QString("o"));
// str == "color behavior flavor neighbor"
```

Note: The replacement text is not rescanned after it is inserted.

Example:

```
QString equis = "xxxxxx";
equis.replace("xx", "x");
// equis == "xxx"
```

QString & QString::replace(const QChar * before, int blen, const QChar * after, int alen, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces each occurrence in this string of the first blen characters of before with the first alen characters of after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.5.

QString & QString::replace(QChar ch, const QString & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the character ch in the string with after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

QString & QString::replace(QChar before, QChar after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the character before with the character after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

QString & QString::replace(const QLatin1String & before, const QLatin1String & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the string before with the string after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Note: The text is not rescanned after a replacement.

This function was introduced in Qt 4.5.

QString & QString::replace(const QLatin1String & before, const QString & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the string before with the string after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Note: The text is not rescanned after a replacement.

This function was introduced in Qt 4.5.

QString & QString::replace(const QString & before, const QLatin1String & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the string before with the string after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Note: The text is not rescanned after a replacement.

This function was introduced in Qt 4.5.

QString & QString::replace(QChar c, const QLatin1String & after, Qt::CaseSensitivity cs = Qt::CaseSensitive)

This function overloads replace().

Replaces every occurrence of the character c with the string after and returns a reference to this string.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

Note: The text is not rescanned after a replacement.

This function was introduced in Qt 4.5.

QString & QString::replace(const QRegExp & rx, const QString & after)

This function overloads replace().

Replaces every occurrence of the regular expression rx in the string with after. Returns a reference to the string. For example:

```
QString s = "Banana";
s.replace(QRegExp("a[mn]"), "ox");
// s == "Boxoxa"
```

For regular expressions containing capturing parentheses, occurrences of **\1**, **\2**, ..., in after are replaced with rx.cap(1), cap(2), ...

```
QString t = "A <i>bon mot</i>.";
t.replace(QRegExp("<i>([^<]*)</i>"), "\\emph{\\1}");
// t == "A \\emph{bon mot}."
```

See also indexOf(), lastIndexOf(), remove(), and QRegExp::cap().

void QString::reserve(int size)

Attempts to allocate memory for at least size characters. If you know in advance how large the string will be, you can call this function, and if you resize the string often you are likely to get better performance. If size is an underestimate, the worst that will happen is that the QString will be a bit slower.

The sole purpose of this function is to provide a means of fine tuning QString's memory usage. In general, you will rarely ever need to call this function. If you want to change the size of the string, call resize().

this function. If you want to change the size of the string, call resize().

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 large enough to make a call to reserve() worthwhile:

```
QString result;
int maxSize;
bool condition;
QChar nextChar;

result.reserve(maxSize);

while (condition)
    result.append(nextChar);

result.squeeze();
```

See also squeeze() and capacity().

void QString::resize(int size)

Sets the size of the string to size characters.

If size is greater than the current size, the string is extended to make it size characters long with the extra characters added to the end. The new characters are uninitialized.

If size is less than the current size, characters are removed from the end.

Example:

```
QString s = "Hello world";
s.resize(5);
// s == "Hello"

s.resize(8);
// s == "Hello???" (where ? stands for any character)
```

If you want to append a certain number of identical characters to the string, use operator+=() as follows rather than resize():

```
QString t = "Hello";
t += QString(10, 'X');
// t == "HelloXXXXXXXXXX"
```

If you want to expand the string so that it reaches a certain width and fill the new positions with a particular character, use the leftJustified() function:

If size is negative, it is equivalent to passing zero.

```
QString r = "Hello";
r = r.leftJustified(10, ' ');
// r == "Hello     "
```

See also truncate() and reserve().

QString QString::right(int n) const

Returns a substring that contains the n rightmost characters of the string.

The entire string is returned if n is greater than size() or less than zero.

```
QString x = "Pineapple";
QString y = x.right(5);      // y == "apple"
```

See also left(), mid(), and endsWith().

QString QString::rightJustified(int width, QChar fill = QLatin1Char(' '), bool truncate = false) const

Returns a string of size() width that contains the fill character followed by the string. For example:

```
QString s = "apple";
QString t = s.rightJustified(8, '.');    // t == "...apple"
```

If truncate is false and the size() of the string is more than width, then the returned string is a copy of the string.

If truncate is true and the size() of the string is more than width, then the resulting string is truncated at position width.

```
QString str = "Pineapple";
str = str.rightJustified(5, '.', true);    // str == "Pinea"
```

See also leftJustified().

QStringRef QString::rightRef(int n) const

Returns a substring reference to the n rightmost characters of the string.

If n is greater than size() or less than zero, a reference to the entire string is returned.

```
QString x = "Pineapple";
QStringRef y = x.rightRef(5);       // y == "apple"
```

This function was introduced in Qt 4.4.

See also right(), leftRef(), midRef(), and endsWith().

QString QString::section(QChar sep, int start, int end = -1, SectionFlags 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. Fields are numbered 0, 1, 2, etc., counting from the left, and -1, -2, etc., counting from right to left.

The flags argument can be used to affect some aspects of the function's behavior, e.g. whether to be case sensitive, whether to skip empty fields and how to deal with leading and trailing separators; see SectionFlags.

```
QString str;
QString csv = "forename,middlename,surname,phone";
QString path = "/usr/local/bin/myapp"; // First field is empty
QString::SectionFlag flag = QString::SectionSkipEmpty;

str = csv.section(',', 2, 2);   // str == "surname"
str = path.section('/', 3, 4);  // str == "bin/myapp"
str = path.section('/', 3, 3, flag); // str == "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.

```
str = csv.section(',', -3, -2);  // str == "middlename,surname"
str = path.section('/', -1); // str == "myapp"
```

See also split().

QString QString::section(const QString & sep, int start, int end = -1, SectionFlags flags = SectionDefault) const

This function overloads section().

```
QString str;
QString data = "forename**middlename**surname**phone";

str = data.section("**", 2, 2); // str == "surname"
str = data.section("**", -3, -2); // str == "middlename**surname"
```

See also split().

QString QString::section(const QRegExp & reg, int start, int end = -1, SectionFlags flags = SectionDefault) const

This function overloads section().

This string is treated as a sequence of fields separated by the regular expression, reg.

```
QString line = "forename\tmiddlename  surname \t \t phone";
QRegExp sep("\\s+");
str = line.section(sep, 2, 2); // s == "surname"
str = line.section(sep, -3, -2); // s == "middlename  surname"
```

Warning: Using this QRegExp version is much more expensive than the overloaded string and character versions.

See also split() and simplified().

QString & QString::setNum(int n, int base = 10)

Sets the string to the printed value of n in the specified base, and returns a reference to the string.

The base is 10 by default and must be between 2 and 36. For bases other than 10, n is treated as an unsigned integer.

```
QString str;
str.setNum(1234);       // str == "1234"
```

The formatting always uses QLocale::C, i.e., English/UnitedStates. To get a localized string representation of a number, use QLocale::toString() with the appropriate locale.

QString & QString::setNum(uint n, int base = 10)

This is an overloaded function.

QString & QString::setNum(long n, int base = 10)

This is an overloaded function.

QString & QString::setNum(ulong n, int base = 10)

This is an overloaded function.

QString & QString::setNum(qlonglong n, int base = 10)

This is an overloaded function.

QString & QString::setNum(qulonglong n, int base = 10)

This is an overloaded function.

QString & QString::setNum(short n, int base = 10)

This is an overloaded function.

QString & QString::setNum(ushort n, int base = 10)

This is an overloaded function.

QString & QString::setNum(double n, char format = 'g', int precision = 6)

This is an overloaded function.

Sets the string to the printed value of n, formatted according to the given format and precision, and returns a reference to the string.

The format can be 'f', 'F', 'e', 'E', 'g' or 'G' (see the arg() function documentation for an explanation of the formats).

Unlike QLocale::toString(), this function doesn't honor the user's locale settings.

QString & QString::setNum(float n, char format = 'g', int precision = 6)

This is an overloaded function.

Sets the string to the printed value of n, formatted according to the given format and precision, and returns a reference to the string.

QString & QString::setRawData(const QChar * unicode, int size)

Resets the QString to use the first size Unicode characters in the array unicode. The data in unicode is not copied. The caller must be able to guarantee that unicode will not be deleted or modified as long as the QString (or an unmodified copy of it) exists.

This function can be used instead of fromRawData() to re-use existings QString objects to save memory re-allocations.

This function was introduced in Qt 4.7.

See also fromRawData().

QString & QString::setUnicode(const QChar * unicode, int size)

Resizes the string to size characters and copies unicode into the string.

If unicode is 0, nothing is copied, but the string is still resized to size.

See also unicode() and setUtf16().

QString & QString::setUtf16(const ushort * unicode, int size)

Resizes the string to size characters and copies unicode into the string.

If unicode is 0, nothing is copied, but the string is still resized to size.

Note that unlike fromUtf16(), this function does not consider BOMs and possibly differing byte ordering.

See also utf16() and setUnicode().

QString QString::simplified() const

Returns a string that has whitespace removed from the start and the end, and that has each sequence of internal whitespace replaced with a single space.

Whitespace means any character for which QChar::isSpace() returns true. This includes the ASCII characters '\t', '\n', '\v', '\f', '\r', and ' '.

Example:

```
QString str = "  lots\t of\nwhitespace\r\n ";
```

```
str = str.simplified();
// str == "lots of whitespace";
```

See also trimmed().

int QString::size() const

Returns the number of characters in this string.

The last character in the string is at position size() - 1. In addition, QString ensures that the character at position size() is always '\0', so that you can use the return value of data() and constData() as arguments to functions that expect '\0'-terminated strings.

Example:

```
QString str = "World";
int n = str.size();         // n == 5
str.data()[0];              // returns 'W'
str.data()[4];              // returns 'd'
str.data()[5];              // returns '\0'
```

See also isEmpty() and resize().

QStringList QString::split(const QString & sep, SplitBehavior behavior = KeepEmptyParts, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Splits the string into substrings wherever sep occurs, and returns the list of those strings. If sep does not match anywhere in the string, split() returns a single-element list containing this string.

cs specifies whether sep should be matched case sensitively or case insensitively.

If behavior is QString::SkipEmptyParts, empty entries don't appear in the result. By default, empty entries are kept.

Example:

```
QString str = "a,,b,c";

QStringList list1 = str.split(",");
// list1: [ "a", "", "b", "c" ]

QStringList list2 = str.split(",", QString::SkipEmptyParts);
// list2: [ "a", "b", "c" ]
```

See also QStringList::join() and section().

QStringList QString::split(const QChar & sep, SplitBehavior behavior = KeepEmptyParts, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This is an overloaded function.

QStringList QString::split(const QRegExp & rx, SplitBehavior behavior = KeepEmptyParts) const

This is an overloaded function.

Splits the string into substrings wherever the regular expression rx matches, and returns the list of those strings. If rx does not match anywhere in the string, split() returns a single-element list containing this string.

Here's an example where we extract the words in a sentence using one or more whitespace characters as the separator:

```
QString str;
QStringList list;

str = "Some  text\n\twith  strange whitespace.";
list = str.split(QRegExp("\\s+"));
// list: [ "Some", "text", "with", "strange", "whitespace." ]
```

Here's a similar example, but this time we use any sequence of non-word characters as the separator:

```
str = "This time, a normal English sentence.";
list = str.split(QRegExp("\\W+"), QString::SkipEmptyParts);
// list: [ "This", "time", "a", "normal", "English", "sentence" ]
```

Here's a third example where we use a zero-length assertion, **\b** (word boundary), to split the string into an alternating sequence of non-word and word tokens:

```
str = "Now: this sentence fragment.";
list = str.split(QRegExp("\\b"));
// list: [ "", "Now", ": ", "this", " ", "sentence", " ", "fragment", "." ]
```

See also QStringList::join() and section().

QString & QString::sprintf(const char * cformat, ...)

Safely builds a formatted string from the format string cformat and an arbitrary list of arguments.

The %lc escape sequence expects a unicode character of type ushort (as returned by QChar::unicode()). The %ls escape sequence expects a pointer to a zero-terminated array of unicode characters of type ushort (as returned by QString::utf16()).

Note: This function expects a UTF-8 string for %s and Latin-1 for the format string.

The format string supports most of the conversion specifiers provided by printf() in the standard C++ library. It doesn't honor the length modifiers (e.g. `h` for `short`, `ll` for `long long`). If you need those, use the standard snprintf() function instead:

```
size_t BufSize;
char buf[BufSize];

::snprintf(buf, BufSize, "%lld", 123456789LL);
QString str = QString::fromAscii(buf);
```

Warning: We do not recommend using QString::sprintf() in new Qt code. Instead, consider using QTextStream or arg(), both of which support Unicode strings seamlessly and are type-safe. Here's an example that uses QTextStream:

```
```

```
QString result;
QTextStream(&result) << "pi = " << 3.14;
// result == "pi = 3.14"
```

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.

See also arg().

void QString::squeeze()

Releases any memory not required to store the character data.

The sole purpose of this function is to provide a means of fine tuning QString's memory usage. In general, you will rarely ever need to call this function.

See also reserve() and capacity().

bool QString::startsWith(const QString & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

Returns true if the string starts with s; otherwise returns false.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

```
QString str = "Bananas";
str.startsWith("Ban");     // returns true
str.startsWith("Car");     // returns false
```

See also endsWith().

bool QString::startsWith(const QLatin1String & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads startsWith().

bool QString::startsWith(const QChar & c, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This function overloads startsWith().

Returns true if the string starts with c; otherwise returns false.

bool QString::startsWith(const QStringRef & s, Qt::CaseSensitivity cs = Qt::CaseSensitive) const

This is an overloaded function.

Returns true if the string starts with the string reference s; otherwise returns false.

If cs is Qt::CaseSensitive (default), the search is case sensitive; otherwise the search is case insensitive.

This function was introduced in Qt 4.8.

See also endsWith().

void QString::swap(QString & other)

Swaps string other with this string. This operation is very fast and never fails.

This function was introduced in Qt 4.8.

QByteArray QString::toAscii() const

Returns an 8-bit representation of the string as a QByteArray.

If a codec has been set using QTextCodec::setCodecForCStrings(), it is used to convert Unicode to 8-bit char; otherwise this function does the same as toLatin1().

Note that, despite the name, this function does not necessarily return an US-ASCII (ANSI X3.4-1986) string and its result may not be US-ASCII compatible.

See also fromAscii(), toLatin1(), toUtf8(), toLocal8Bit(), and QTextCodec.

QString QString::toCaseFolded() const

Returns the case folded equivalent of the string. For most Unicode characters this is the same as toLower().

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

Returns the string converted to a `double` value.

Returns 0.0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

```
QString str = "1234.56";
double val = str.toDouble();   // val == 1234.56
```

Various string formats for floating point numbers can be converted to double values:

```
bool ok;
double d;

d = QString( "1234.56e-02" ).toDouble(&ok); // ok == true, d == 12.3456
```

This function tries to interpret the string according to the current locale. The current locale is determined from the system at application startup and can be changed by calling QLocale::setDefault(). If the string cannot be interpreted according to the current locale, this function falls back on the "C" locale.

```
QLocale::setDefault(QLocale::C);
d = QString( "1234,56" ).toDouble(&ok); // ok == false
d = QString( "1234.56" ).toDouble(&ok); // ok == true, d == 1234.56
```

```
QLocale::setDefault(QLocale::German);
d = QString( "1234,56" ).toDouble(&ok); // ok == true, d == 1234.56
d = QString( "1234.56" ).toDouble(&ok); // ok == true, d == 1234.56
```

Due to the ambiguity between the decimal point and thousands group separator in various locales, this function does not handle thousands group separators. If you need to convert such numbers, see QLocale::toDouble().

```
QLocale::setDefault(QLocale::C);
d = QString( "1234,56" ).toDouble(&ok); // ok == false
```

See also number(), QLocale::setDefault(), QLocale::toDouble(), and trimmed().

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; otherwise *ok is set to true. Returns 0.0 if the conversion fails.

Example:

```
QString str1 = "1234.56";
str1.toFloat();             // returns 1234.56

bool ok;
QString str2 = "R2D2";
str2.toFloat(&ok);          // returns 0.0, sets ok to false
```

See also number(), toDouble(), and toInt().

int QString::toInt(bool * ok = 0, int base = 10) const

Returns the string converted to an `int` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
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(), toUInt(), and toDouble().

QByteArray QString::toLatin1() const

Returns a Latin-1 representation of the string as a QByteArray.

The returned byte array is undefined if the string contains non-Latin1 characters. Those characters may be suppressed or replaced with a question mark.

See also fromLatin1(), toAscii(), toUtf8(), toLocal8Bit(), and QTextCodec.

QByteArray QString::toLocal8Bit() const

Returns the local 8-bit representation of the string as a QByteArray. The returned byte array is undefined if the string contains characters not supported by the local 8-bit encoding.

QTextCodec::codecForLocale() is used to perform the conversion from Unicode. If the locale encoding could not be determined, this function does the same as toLatin1().

If this string contains any characters that cannot be encoded in the locale, the returned byte array is undefined. Those characters may be suppressed or replaced by another.

See also fromLocal8Bit(), toAscii(), toLatin1(), toUtf8(), and QTextCodec.

long QString::toLong(bool * ok = 0, int base = 10) const

Returns the string converted to a `long` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;

long hex = str.toLong(&ok, 16);     // hex == 255, ok == true
long dec = str.toLong(&ok, 10);     // dec == 0, ok == false
```

See also number(), toULong(), and toInt().

qlonglong QString::toLongLong(bool * ok = 0, int base = 10) const

Returns the string converted to a `long long` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;
```

```
qint64 hex = str.toLongLong(&ok, 16);      // hex == 255, ok == true
qint64 dec = str.toLongLong(&ok, 10);      // dec == 0, ok == false
```

See also number(), toULongLong(), and toInt().

QString QString::toLower() const

Returns a lowercase copy of the string.

```
QString str = "Qt by NOKIA";
str = str.toLower();        // str == "qt by nokia"
```

The case conversion will always happen in the 'C' locale. For locale dependent case folding use QLocale::toLower()

See also toUpper() and QLocale::toLower().

short QString::toShort(bool * ok = 0, int base = 10) const

Returns the string converted to a `short` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;

short hex = str.toShort(&ok, 16);   // hex == 255, ok == true
short dec = str.toShort(&ok, 10);   // dec == 0, ok == false
```

See also number(), toUShort(), and toInt().

std::string QString::toStdString() const

Returns a std::string object with the data contained in this QString. The Unicode data is converted into 8-bit characters using the toAscii() function.

This operator is mostly useful to pass a QString to a function that accepts a std::string object.

If the QString contains Unicode characters that the QTextCodec::codecForCStrings() codec cannot handle, using this operator can lead to loss of information.

This operator is only available if Qt is configured with STL compatibility enabled.

See also toAscii(), toLatin1(), toUtf8(), and toLocal8Bit().

std::wstring QString::toStdWString() const

Returns a std::wstring object with the data contained in this QString. The std::wstring is encoded in utf16 on platforms where wchar_t is 2 bytes wide (e.g. windows) and in ucs4 on platforms where wchar_t is 4 bytes wide (most Unix systems).

This operator is mostly useful to pass a QString to a function that accepts a std::wstring object.

This operator is only available if Qt is configured with STL compatibility enabled.

See also utf16(), toAscii(), toLatin1(), toUtf8(), and toLocal8Bit().

uint QString::toUInt(bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned int` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;

uint hex = str.toUInt(&ok, 16);     // hex == 255, ok == true
uint dec = str.toUInt(&ok, 10);     // dec == 0, ok == false
```

See also number() and toInt().

ulong QString::toULong(bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned long` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;

ulong hex = str.toULong(&ok, 16);   // hex == 255, ok == true
ulong dec = str.toULong(&ok, 10);   // dec == 0, ok == false
```

See also number().

qulonglong QString::toULongLong(bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned long long` using base base, which is 10 by default and must be between 2 and 36, or 0.

Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;

quint64 hex = str.toULongLong(&ok, 16);    // hex == 255, ok == true
quint64 dec = str.toULongLong(&ok, 10);    // dec == 0, ok == false
```

See also number() and toLongLong().

ushort QString::toUShort(bool * ok = 0, int base = 10) const

Returns the string converted to an `unsigned short` using base base, which is 10 by default and must be between 2 and 36, or 0. Returns 0 if the conversion fails.

If a conversion error occurs, *ok is set to false; otherwise *ok is set to true.

If base is 0, the C language convention is used: If the string begins with "0x", base 16 is used; if the string begins with "0", base 8 is used; otherwise, base 10 is used.

Example:

```
QString str = "FF";
bool ok;

ushort hex = str.toUShort(&ok, 16);     // hex == 255, ok == true
ushort dec = str.toUShort(&ok, 10);     // dec == 0, ok == false
```

See also number() and toShort().

QVector<uint> QString::toUcs4() const

Returns a UCS-4/UTF-32 representation of the string as a QVector<uint>.

UCS-4 is a Unicode codec and is lossless. All characters from this string can be encoded in UCS-4. The vector is not null terminated.

This function was introduced in Qt 4.2.

See also fromUtf8(), toAscii(), toLatin1(), toLocal8Bit(), QTextCodec, fromUcs4(), and toWCharArray().

QString QString::toUpper() const

Returns an uppercase copy of the string.

```
QString str = "TeXt";
str = str.toUpper();        // str == "TEXT"
```

The case conversion will always happen in the 'C' locale. For locale dependent case folding use QLocale::toUpper()

See also toLower() and QLocale::toLower().

QByteArray QString::toUtf8() const

Returns a UTF-8 representation of the string as a QByteArray.

UTF-8 is a Unicode codec and can represent all characters in a Unicode string like QString.

However, in the Unicode range, there are certain codepoints that are not considered characters. The Unicode standard reserves the last two codepoints in each Unicode Plane (U+FFFE, U+FFFF, U+1FFFE, U+1FFFF, U+2FFFE, etc.), as well as 16 codepoints in the range U+FDD0..U+FDDF, inclusive, as non-characters. If any of those appear in the string, they may be discarded and will not appear in the UTF-8 representation, or they may be replaced by one or more replacement characters.

See also fromUtf8(), toAscii(), toLatin1(), toLocal8Bit(), and QTextCodec.

int QString::toWCharArray(wchar_t * array) const

Fills the array with the data contained in this QString object. The array is encoded in utf16 on platforms where wchar_t is 2 bytes wide (e.g. windows) and in ucs4 on platforms where wchar_t is 4 bytes wide (most Unix systems).

array has to be allocated by the caller and contain enough space to hold the complete string (allocating the array with the same length as the string is always sufficient).

returns the actual length of the string in array.

Note: This function does not append a null character to the array.

This function was introduced in Qt 4.2.

See also utf16(), toUcs4(), toAscii(), toLatin1(), toUtf8(), toLocal8Bit(), and toStdWString().

QString QString::trimmed() 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 the ASCII characters '\t', '\n', '\v', '\f', '\r', and ' '.

Example:

```
QString str = "  lots\t of\nwhitespace\r\n ";
str = str.trimmed();
// str == "lots\t of\nwhitespace"
```

Unlike simplified(), trimmed() leaves internal whitespace alone.

See also simplified().

void QString::truncate(int position)

Truncates the string at the given position index.

If the specified position index is beyond the end of the string, nothing happens.

Example:

```
QString str = "Vladivostok";
str.truncate(4);
// str == "Vlad"
```

If position is negative, it is equivalent to passing zero.

See also chop(), resize(), and left().

const QChar * QString::unicode() const

Returns a '\0'-terminated Unicode representation of the string. The result remains valid until the string is modified.

See also setUnicode() and utf16().

const ushort * QString::utf16() const

Returns the QString as a '\0'-terminated array of unsigned shorts. The result remains valid until the string is modified.

The returned string is in host byte order.

See also setUtf16() and unicode().

QString & QString::vsprintf(const char * cformat, va_list ap)

Equivalent method to sprintf(), but takes a va_list ap instead a list of variable arguments. See the sprintf() documentation for an explanation of cformat.

This method does not call the va_end macro, the caller is responsible to call va_end on ap.

See also sprintf().

bool QString::operator!=(const QString & other) const

Returns true if this string is not equal to string other; otherwise returns false.

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 localeAwareCompare().

bool QString::operator!=(const QLatin1String & other) const

This function overloads operator!=().

bool QString::operator!=(const QByteArray & other) const

This function overloads operator!=().

The other byte array is converted to a QString using the fromAscii() function. If any NUL characters ('\0') are embedded in the byte array, they will be included in the transformation.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator!=(const char * other) const

This function overloads operator!=().

The other const char pointer is converted to a QString using the fromAscii() function.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::operator+=(const QString & other)

Appends the string other onto the end of this string and returns a reference to this string.

Example:

```
QString x = "free";
QString y = "dom";
x += y;
// x == "freedom"
```

This operation is typically very fast (constant time), because QString preallocates extra space at the end of the string data so it can grow without reallocating the entire string each time.

See also append() and prepend().

QString & QString::operator+=(const QLatin1String & str)

This function overloads operator+=().

Appends the Latin-1 string str to this string.

QString & QString::operator+=(const QByteArray & ba)

This function overloads operator+=().

Appends the byte array ba to this string. The byte array is converted to Unicode using the fromAscii() function. If any NUL characters ('\0') are embedded in the ba byte array, they will be included in the transformation.

You can disable this function by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

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

This function overloads operator+=().

Appends the string str to this string. The const char pointer is converted to Unicode using the fromAscii() function.

You can disable this function by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::operator+=(const QStringRef & str)

This function overloads operator+=().

Appends the string section referenced by str to this string.

QString & QString::operator+=(char ch)

This function overloads operator+=().

Appends the character ch to this string. The character is converted to Unicode using the fromAscii() function.

You can disable this function by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::operator+=(QChar ch)

This function overloads operator+=().

Appends the character ch to the string.

bool QString::operator<(const QString & other) const

Returns true if this string is lexically less than string other; otherwise returns false.

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 using the QString::localeAwareCompare() function.

bool QString::operator<(const QLatin1String & other) const

This function overloads operator<().

bool QString::operator<(const QByteArray & other) const

This function overloads operator<().

The other byte array is converted to a QString using the fromAscii() function. If any NUL characters ('\0') are embedded in the byte array, they will be included in the transformation.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator<(const char * other) const

This function overloads operator<().

The other const char pointer is converted to a QString using the fromAscii() function.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator<=(const QString & other) const

Returns true if this string is lexically less than or equal to string other; otherwise returns false.

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 localeAwareCompare().

bool QString::operator<=(const QLatin1String & other) const

This function overloads operator<=().

bool QString::operator<=(const QByteArray & other) const

This function overloads operator<=().

The other byte array is converted to a QString using the fromAscii() function. If any NUL characters ('\0') are embedded in the byte array, they will be included in the transformation.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator<=(const char * other) const

This function overloads operator<=().

The other const char pointer is converted to a QString using the fromAscii() function.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::operator=(const QString & other)

Assigns other to this string and returns a reference to this string.

QString & QString::operator=(QString && other)

QString & QString::operator=(const QLatin1String & str)

This function overloads operator=().

Assigns the Latin-1 string str to this string.

QString & QString::operator=(const QByteArray & ba)

This function overloads operator=().

Assigns ba to this string. The byte array is converted to Unicode using the fromAscii() function. This function stops conversion at the first NUL character found, or the end of the ba byte array.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

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

This function overloads operator=().

Assigns str to this string. The const char pointer is converted to Unicode using the fromAscii() function.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::operator=(char ch)

This function overloads operator=().

Assigns character ch to this string. The character is converted to Unicode using the fromAscii() function.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QString & QString::operator=(QChar ch)

This function overloads operator=().

Sets the string to contain the single character ch.

bool QString::operator==(const QString & other) const

Returns true if string other is equal to this string; otherwise returns false.

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 localeAwareCompare().

bool QString::operator==(const QLatin1String & other) const

This function overloads operator==().

bool QString::operator==(const QByteArray & other) const

This function overloads operator==().

The other byte array is converted to a QString using the fromAscii() function. This function stops conversion at the first NUL character found, or the end of the byte array.

You can disable this operator by defining QT_NO_CAST_FROM_ASCII when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator==(const char * other) const

This function overloads operator==().

The other const char pointer is converted to a QString using the fromAscii() function.

You can disable this operator by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator>(const QString & other) const

Returns true if this string is lexically greater than string other; otherwise returns false.

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 localeAwareCompare().

bool QString::operator>(const QLatin1String & other) const

This function overloads operator>().

bool QString::operator>(const QByteArray & other) const

This function overloads operator>().

The other byte array is converted to a QString using the fromAscii() function. If any NUL characters ('\0') are embedded in the byte array, they will be included in the transformation.

You can disable this operator by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator>(const char * other) const

This function overloads operator>().

The other const char pointer is converted to a QString using the fromAscii() function.

You can disable this operator by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator>=(const QString & other) const

Returns true if this string is lexically greater than or equal to string other; otherwise returns false.

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 localeAwareCompare().

bool QString::operator>=(const QLatin1String & other) const

This function overloads operator>=().

This function overloads operator>=().

bool QString::operator>=(const QByteArray & other) const

This function overloads operator>=().

The other byte array is converted to a QString using the fromAscii() function. If any NUL characters ('\0') are embedded in the byte array, they will be included in the transformation.

You can disable this operator by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

bool QString::operator>=(const char * other) const

This function overloads operator>=().

The other const char pointer is converted to a QString using the fromAscii() function.

You can disable this operator by defining `QT_NO_CAST_FROM_ASCII` when you compile your applications. This can be useful if you want to ensure that all user-visible strings go through QObject::tr(), for example.

QCharRef QString::operator[](int position)

Returns the character at the specified position in the string as a modifiable reference.

Example:

```
QString str;

if (str[0] == QChar('?'))
    str[0] = QChar('_');
```

The return value is of type QCharRef, a helper class for QString. When you get an object of type QCharRef, you can use it as if it were a QChar &. If you assign to it, the assignment will apply to the character in the QString from which you got the reference.

See also at().

const QChar QString::operator[](int position) const

This function overloads operator[]().

QCharRef QString::operator[](uint position)

This function overloads operator[]().

Returns the character at the specified position in the string as a modifiable reference. Equivalent to `at(position)`.

const QChar QString::operator[](uint position) const

This function overloads operator[]().

Related Non-Members

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

Returns true if s1 is not equal to s2; otherwise returns false.

For s1 != 0, this is equivalent to `compare( s1, s2 ) != 0`. Note that no string is equal to s1 being 0.

See also QString::compare().

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

Returns a string which is the result of concatenating s1 and s2.

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

Returns a string which is the result of concatenating s1 and s2 (s2 is converted to Unicode using the QString::fromAscii() function).

See also QString::fromAscii().

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

Returns a string which is the result of concatenating s1 and s2 (s1 is converted to Unicode using the QString::fromAscii() function).

See also QString::fromAscii().

const QString operator+(char ch, const QString & s)

Returns a string which is the result of concatenating the character ch and the string s.

const QString operator+(const QString & s, char ch)

Returns a string which is the result of concatenating the string s and the character ch.

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

Returns true if s1 is lexically less than s2; otherwise returns false. For s1 != 0, this is equivalent to `compare(s1, s2) < 0`.

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 using the QString::localeAwareCompare() function.

See also QString::compare().

QDataStream & operator<<(QDataStream & stream, const QString & string)

Writes the given string to the specified stream.

See also Serializing Qt Data Types.

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

Returns true if s1 is lexically less than or equal to s2; otherwise returns false. For s1 != 0, this is equivalent to `compare(s1, s2) <= 0`.

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().

See also QString::compare().

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

This function overloads operator==().

Returns true if s1 is equal to s2; otherwise returns false. Note that no string is equal to s1 being 0.

Equivalent to `s1 != 0 && compare(s1, s2) == 0`.

See also QString::compare().

bool operator>(const char * s1, const QString & s2)

Returns true if s1 is lexically greater than s2; otherwise returns false. Equivalent to `compare(s1, s2) > 0`.

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 using the QString::localeAwareCompare() function.

See also QString::compare().

bool operator>=(const char * s1, const QString & s2)

Returns true if s1 is lexically greater than or equal to s2; otherwise returns false. For s1 != 0, this is equivalent to `compare(s1, s2) >= 0`.

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 using the QString::localeAwareCompare() function.

QDataStream & operator>>(QDataStream & stream, QString & string)

Reads a string from the specified stream into the given string.

See also Serializing Qt Data Types.

Macro Documentation

QT_NO_CAST_FROM_ASCII

Disables automatic conversions from 8-bit strings (char *) to unicode QStrings

disables automatic conversions from 8-bit strings (char *) to unicode QStrings

See also QT_NO_CAST_TO_ASCII and QT_NO_CAST_FROM_BYTEARRAY.

QT_NO_CAST_TO_ASCII

disables automatic conversion from QString to 8-bit strings (char *)

See also QT_NO_CAST_FROM_ASCII and QT_NO_CAST_FROM_BYTEARRAY.