Qt World Summit 2017
October 10-12 | Berlin, Germany

QStringView QStringView everywhere

Marc Mutz, Senior Software Engineer at KDAB

About me

Intro (cont'd)

KDAB

Author of QStringView

Overview

- QStringView
- Using QStringView
- API Patterns For QStringView
- Future Directions
- Heterogeneous Associative Container Lookup
- Technical Deep Dive: Managing Overloads
- Technical Deep Dive: Contracts

QStringView

QStringView (cont'd)

The QStringView class provides a unified view on UTF-16 strings with a read-only subset of the QString API.

- container-agnostic
- UTF-16 like QString, std::u16string, std::wstring (Win only)
- non-mutable
- QString API

QString vs. QStringView

- QString is an *owning* container
  - QStringView is *non-owning*
    - Just (ptr, size)
- QString *copies* the data
  - QStringView *references* the data
- QString ← const char* implicit
  - QStringView can't be constructed from 8-bit
- QString functions (mostly) handle out-of-range
  - QStringView functions assert
- QString::data() is usually NUL-terminated
  - QStringView::data() usually isn't
- QString::data() never returns nullptr
  - QStringView::data() == nullptr ⇔ QStringView::isNull()

My First QStringView Function


```
 1 constexpr bool isValidFirstChar(QChar c)    { ... }
 2 constexpr bool isValidFollowupChar(QChar c) { ... }
 3
 4 constexpr bool isValidIdentifier(QStringView s) noexcept {
 5
 6
 7     if (s.isEmpty())
 8         return false;
 9     if (!isValidFirstChar(s.front()))
10         return false;
11     for (auto c : s.mid(1))
12         if (!isValidFollowupChar(c))
13             return false;
14     return true;
15 }
16
17 int main(int argc, char *argv[]) {
18     if (argc > 1)
19         std::cout << isValidIdentifier(QString::fromLocal8Bit(argv[1]));
20     else
21         std::cout << isValidIdentifier(u"_1"); // cout << true
22     std::cout << std::endl;
23 }
```

Why QStringView?

Efficiency

Why QStringView? (cont'd)

A better QStringRef

Why QStringView? (cont'd)

A better std::basic_string_view<QChar>

Why QStringView? (cont'd)

A better (const QChar*, int)

Expressive

Why QStringView? (cont'd)

Interface Type

Using QStringView

Construction

Construct a QStringView from:

- *Char* string literals
- *Char** begin + end
- NUL-terminated *Char**
- *Char** + size
- QString
- QStringRef
- std::basic_string<*Char*, ...>
- [std::basic_string_view<*Char*, ...>]
- [std::vector/QVector<Char>]
- [QVarLengthArray<Char, N>]
- nullptr

where *Char* ∈ {QChar, ushort, char16_t, wchar_t}

Salient Functions

- data() / utf16()
- size() / length() / isEmpty() / isNull()
- {,c}{,r}{begin,end}()
- at() / operator[]
- {starts,ends}With()
- front() / back()
- left() / right() / mid()
- chop() / chopped() / truncate() [/ trim()] / trimmed()
- toString()
- to{Latin1,Local8Bit,Utf8,Ucs4}()
- relational operators / qHash()

Target for 5.11: All const QString API (within reason)

Intentionally Unsupported

- mutating functions
- split() → [QStringTokenizer]
- number() → [fromNumber()]
- magic

Magic


```
1 QString::mid(int pos, int n = -1)
```

Magic (cont'd)


```
 1 QContainerImplHelper::mid(int originalLength, int &position, int &length)
 2    -> CutResult
 3 {
 4     if (position > originalLength)
 5         return Null;                                          // 1 branch
 6
 7     if (position < 0) {
 8         if (length < 0 || length + position >= originalLength)
 9             return Full;                                      // 3-4 branches
10         if (length + position <= 0)
11             return Null;                                      // 4-5 branches
12         length += position;
13         position = 0;
14     } else if (uint(length) > uint(originalLength - position)) {
15         length = originalLength - position;
16     }
17
18     if (position == 0 && length == originalLength)
19         return Full;                                          // 4-5 branches
20
21     return length > 0 ? Subset : Empty;                       // 5-6 branches
22 }
```

No Magic

```
1 QStringView::mid(qssize_t pos)
2 QStringView::mid(qssize_t pos, qssize_t n)
```

No Magic (cont'd)

```
1 QStringView::mid(qssize_t pos)
2 { return {data() + pos, size() - pos}; }
3 QStringView::mid(qssize_t pos, qssize_t n)
4 { return {data() + pos, n}; }
```

QStringView as an Interface Type

Interface Type

```
1 int countXs(QStringView s)
```

Interface Type (cont'd)

```
1 QStringView pluginName()
```

Interface Type (cont'd)

```
1 QStringView sv = someFunc();
```

Interface Type (cont'd)

```
1 auto r = someFunc();
2 auto sv = QStringView{r};
```

Interface Type (cont'd)

```
1 auto c = countXs(someFunc());
```

Conclusion

Use QStringView as

- function argument ⇒ everything ok
- return argument ⇒ document lifetime
- automatic or member variable ⇒ watch out for lifetime issues

API Patterns For QStringView

Legacy


```
1 void f(const QString &); // legacy
2 void f(QStringView); // never ambiguous
```

Legacy Delegation

```
1 void f(const QString &)
2 { return f(QStringView{s}); }
3 void f(QStringView);
```

Setters

```
1 void f(QString s); // take by value
2 void f(QStringView); // never ambiguous
```

Setters Delegation


```
1 void f(QString s); // take by value
2 void f(QStringView sv) { return f(sv.toString()); }
```

High-Performance

```
1 void f(QStringView);
2 void f(QLatin1String);
3 void f(QChar);
4 void f(const QString &);
```

High-Performance Delegation


```
1 void f(QStringView);
2 void f(QLatin1String);
3 void f(QChar) { return f(QStringView{&c, 1}); }
4 void f(const QString &s) { return f(QStringView{s}); }
```

Getters

```
1 QStringView f();
```

Future Directions

Complete Const QString API (Qt 5.11)

- mainly missing:
  - contains()
  - indexOf()
  - split()
    - WONTFIX (slide 41)
  - number → string → number
    - new API (slide 47)

Remember: QStringView to avoid allocations

QStringTokenizer (cont'd)

QVector<QStringView> split(QStringView sep, ~~~)

QStringTokenizer (cont'd)

QVector<QStringView> split(QStringView sep, ~~~)

QStringTokenizer (cont'd)

QVector<QStringView> split(QStringView sep, ~~~)

- Allocates memory
- Is not cache-friendly

KDAB

Solution: QStringTokenizer

QStringTokenizer (cont'd)

```
1 int countXsInCommaSeparatedList(QStringView s) {
2     int result = 0;
3     for (auto part : QStringTokenizer{s, u","})
4         if (part.trimmed() == QLatin1Char{'X'})
5             ++result;
6     return result;
7 }
```

New Number Conversion API

```
1 QString QString::number(int i, ~~~); // allocates
```

New Number Conversion API (cont'd)


```
1 QFormattedNumber<int> QStringView::number(int i, ~~~);
```

New Number Conversion API (cont'd)


```
1 int QString::toInt(bool *ok, ~~~); // out parameter
```

New Number Conversion API (cont'd)

```
1 bool ok; // init? to true or false?
2 int r = QString::toInt(&ok, ~~~);
3 if (ok) {
4     // use 'r'
```

New Number Conversion API (cont'd)

Hint: ***Return*** *Return Values!*

New Number Conversion API (cont'd)


```
1 std::expected<int> QString::toNumber<int>(~~~);
```

New Number Conversion API (cont'd)


```
1 QResult<int> QString::toNumber<int>(~~~);
```

New Number Conversion API (cont'd)

```
1 if (auto r = QString::toInt(~~~)) {
2     // use *r
```

KDAB

What is QByteArray?

QUtf8String / QByteArrayView (cont'd)

QByteArray semantic overload:

- binary data
- UTF-8-encoded strings

QUtf8String / QByteArrayView (cont'd)

What does f(const QByteArray &) expect?

QUtf8String / QByteArrayView (cont'd)

KDAB

(Partial) Solution: QUtf8String(View?)

QUtf8String / QByteArrayView (cont'd)

QUtf8String: Like QLatin1String, but for UTF-8 char*s

QUtf8String / QByteArrayView (cont'd)

QString ← QUtf8String: implicit
QUtf8String ← {QByteArray, const char*}: explicit

KDAB

Heterogeneous Associative Container Lookup

Heterogeneous Associative Container Lookup (cont'd)

Qt containers lack heterogeneous lookup

Heterogeneous Associative Container Lookup (cont'd)


```
1 QMap<QString, QString> replacements = ~~~;
2 // ~~~
3 auto it1 = replacements.find("$i");                    // allocates
4 auto it2 = replacements.find(QStringView(u"$i"));      // allocates
5 auto it3 = replacements.find(QStringLiteral("$i")); // duplicates
```

Heterogeneous Associative Container Lookup (cont'd)

How many find() functions?

Heterogeneous Associative Container Lookup (cont'd)

STL containers *have* heterogeneous lookup

Heterogeneous Associative Container Lookup (cont'd)


```
1 std::map<QString, QString, std::less<>> replacements = ~~~;
2 // ~~~
3 auto it1 = replacements.find("$i");                  // no alloc
4 auto it2 = replacements.find(QStringView(u"$i"));    // no alloc
5 auto it3 = replacements.find(QStringLiteral("$i")); // no alloc
6 auto it4 = replacements.find("$date");               // no alloc
```

Heterogeneous Associative Container Lookup (cont'd)

How many find() functions?

Heterogeneous Associative Container Lookup (cont'd)

Can we do better?

Heterogeneous Associative Container Lookup (cont'd)

Yes, we can!

Heterogeneous Associative Container Lookup (cont'd)


```
struct Replacement { QString from, to; };
```

Heterogeneous Associative Container Lookup (cont'd)


```
1 const QHash<QStringView, Replacement> replacements = ~~~;
2
3 auto it1 = replacements.find(u"$i");                  // doesn't allo
4 auto it2 = replacements.find(QStringView(u"$i"));     // doesn't allo
5 auto it3 = replacements.find(QStringLiteral("$i")); // doesn't allo
6 auto it4 = replacements.find(u"$date");              // doesn't allo
```

Heterogeneous Associative Container Lookup (cont'd)

How many find() functions?

Heterogeneous Associative Container Lookup (cont'd)


```
1 QHash<QStringView, Replacement> replacements = ~~~;
2
3 auto [name, text] = parseEntity(~~~);
4 auto repl = Replacement{name.toString(), text.toString()};
5 replacement.insert(repl.name, std::move(repl));
```

Heterogeneous Associative Container Lookup (cont'd)

General Pattern, not limited to QStringView

Technical Deep Dive: Managing Overloads

KDAB

Gradual Introduction

KDAB

QString / QStringView overloads

Managing Overloads (cont'd)

No ambiguities

Managing Overloads (cont'd)

QStringView ← T → QString

KDAB

MSVC

Managing Overloads (cont'd)


```
QStringView v = QLatin1String("hello");
```

Managing Overloads (cont'd)

How would you solve:

1. QStringView ← QString implicit
2. But error on QStringView(QLatin1String())

KDAB

Solution: Constrained Templates

Managing Overloads (cont'd)


```
1 template <typename String>
2     requires std::is_same<String, QString>
3 QStringView(const String &s)
4     : QStringView(s.isNull() ? nullptr : s.data(), s.size())
5 {}
```

Managing Overloads (cont'd)

Solution: SFINAE

Managing Overloads (cont'd)


```
1 template <typename String,
2           typename = std::enable_if_t<
3                                 std::is_same<String, QString>>>
4 QStringView(const String &s)
5     : QStringView(s.isNull() ? nullptr : s.data(), s.size())
6 {}
```

KDAB

MSVC

Managing Overloads (cont'd)

Ville: "Try a non-type parameter"

Managing Overloads (cont'd)

me: "a ... what? ... *how*?"

Managing Overloads (cont'd)

```
1 template <typename String>
2 using if_qstring = std::enable_if_t<
3         std::is_same<String, QString>,
4         bool
5     >;
6 template <typename String, if_qstring_like<String> = true>
7 QStringView(const String &s)
8     : QStringView(s.isNull() ? nullptr : s.data(), s.size())
9 {}
```

Technical Deep Dive: Contracts

Contracts Recap

Contracts Recap

Preconditions + Postconditions = Contract

Contracts Recap (cont'd)

Precondition: Predicate *p*

- p() == true ⇒ can call function
- p() == false ⇒ UB to call function

Contracts Recap (cont'd)

KDAB

Precondition Example: std::lower_bound(f, l, v, cmp)

- [f,l) is a valid rage
- [f,l) is sorted according to cmp
- cmp is a Strict Weak Order
- EqualityComparable<decltype(v), ValueType(f)>

Contracts Recap (cont'd)

Calling a function out of contract ⇒ UB

Contracts Recap (cont'd)

What about "valid parameter values"?

Contracts Recap (cont'd)

$$\forall \text{ parameters } P\ p\text{: } p \in Domain(P)$$

Contracts Recap (cont'd)

$\exists$ parameter $P\ p$, $p \notin Domain(P) \Rightarrow$ UB has happened

QStringView vs. (ptr, len)

QStringView(ptr, len)

QStringView vs. (ptr, len) (cont'd)


```
char16_t a[16];
fun(QStringView{a, 16}); // UB: uninitialized
```

QStringView vs. (ptr, len) (cont'd)

```
char16_t a[16];
fun(a, 16); // UB: uninitialized
```

QStringView vs. (ptr, len) (cont'd)

```
char16_t a[16] = {};
fun(a, 17); // UB: off-by-one
```

QStringView vs. (ptr, len) (cont'd)


```
f(QStringView) noexcept // wide contract
f(const QChar*, int); // narrow contract
```

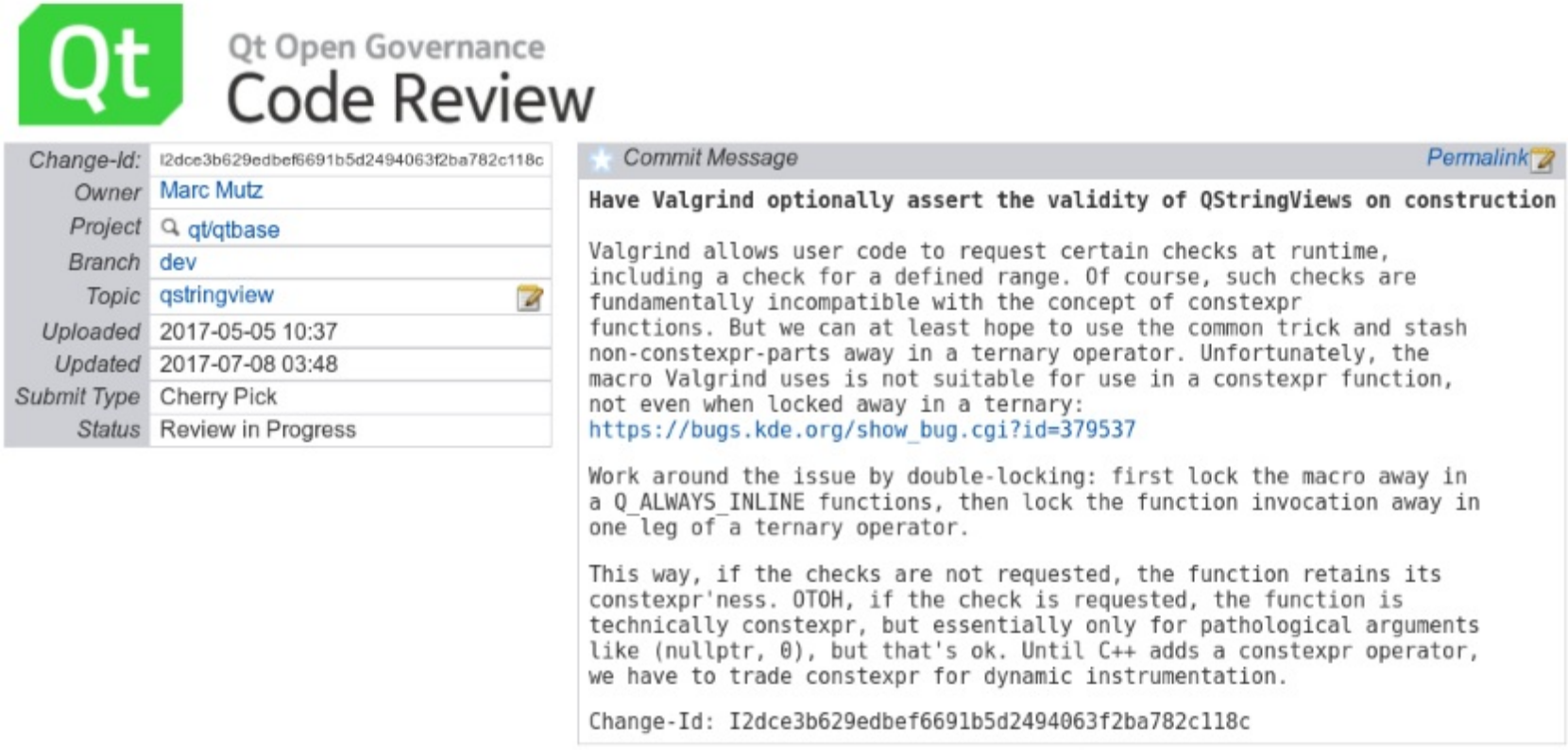
QStringView vs. (ptr, len) (cont'd)

Does it matter?

QStringView vs. (ptr, len) (cont'd)

What if you *could* check (ptr, len) is a valid range?

Valgrind Range Checking



Coming soon to a Qt near you...

Thank you!

www.kdab.com

marc.mutz@kdab.com